3719 Mantell Ave.
Cincinnati, Ohio 45236
(513) 891-4496

SNAPP BASIC

THE ULTIMATE APPLICATION DEVELOPMENT ENVIRONMENT

TABLE OF CONTENTS

Introducing the Snapp BASIC Family

Snapp BASIC is an expanding family of enhancements to Microsoft BASIC designed to accelerate the application development process by providing the programmer with a twentieth century toolbox. In the early days of computing, machines were tremendously expensive, so economic necessity dictated that the needs of the programmers were subservient to the use of the machinery. As a result, programmers would spend many hours 'desk-checking' programs before ever trying them out, and wade through mounds of memory dumps looking for the elusive bugs. Hands-on debugging was just too costly, as were interpretive languages. As machinery has grown in power and plummeted in cost, while people power has remained relatively constant and the cost of good people has risen, the ratio of costs in a computer system has turned totally upside down.

The use of CRT displays and interpretive languages is a laudable first step towards solving this new problem. By unhooking the programmer from his coding sheets and flowcharting templates, and hooking him directly to the target machine, programmer productivity has risen markedly.

We at SNAPP WARE feel that this is just a start toward the solution to the application development process. While we doubt that there will ever be "THE FINAL PROGRAM", which would write programs while chatting with the end user, there are many ways in which we can increase the programmer's productivity.

To allow our users to choose only those functions of the Snapp BASIC family which are appropriate to their needs, we have separated the family into six distinct products. Normally, if you order more than one, you will receive them together on a single disk.

Installation Procedures

There are two distinct kinds of Snapp BASIC disks:
1) The DISTRIBUTION MASTER, which is included in this package.
2) Working copies or TARGET disks.

The distribution master is used only to create working copies of Snapp BASIC and should thereafter be retired to a safe storage location. This procedure is analogous to use of your original LDOS master disks, which also should only be used to make working copies. Working copies of Snapp BASIC can only be created via the installation process described below. While you may certainly make backups of Snapp BASIC disks, the SNAPP software will not function on the backup unless/until the standard installation procedure is completed for the new disk. We have made every effort to preserve the user's interests by allowing creation of an unlimited number of working copies and by establishing a very liberal replacement/upgrade policy for the distribution masters.

This version of Snapp BASIC requires LBASIC 5.1.2. As a convenience to those users who have not yet upgraded to 5.1.2, Logical Systems Incorporated has kindly allowed us to include LBASIC 5.1.2 on the distribution master. This should not preclude updating to 5.1.2, as this upgrade provides many additional enhancements to other portions of LDOS plus complete documentation of these enhancements. We strongly suggest that every user upgrade to LDOS 5.1.2 immediately.

A brief description of the files contained on the SNAPP distribution master:

SNAPPINC	The Snapp BASIC extensions.
LBASIC/CMD	LBASIC 5.1.2 with the Snapp BASIC enabling patches already installed.
LBASIC/OV3	A required LBASIC overlay.
LSNAPPnn/FIX	These files contain optional patches which may be applied at the user's discretion. Descriptions and instructions are contained in each file and may be viewed using the LIST library command.
ASTERON/FIX ASTEROFF/FIX	Applicable only to users of SNAPP SIX, The College Educated Garbage Collector. Descriptions and instructions are contained in each file.
XBOFF1/FIX	Patch to LBASIC to disable the Snapp BASIC extensions.
XBPATCH1/FIX	Patch to LBASIC to enable the Snapp BASIC extensions.
GENERATE/BAS	Applicable only to users of SNAPP FOUR, Extended BASIC Mapping Support. See the documentation on that product for details.

The Snapp BASIC extensions must be installed on an LDOS SYSTEM disk which will be mounted on logical drive Ø during execution of LBASIC. This will normally be a 5" floppy disk, but a special version is available for those using hard disk as their logical drive Ø disk. Refer to the section "HARD DISK INSTALLATION" for instructions concerning this version.

The installation process will create a file named SNAPPINC on the target disk. This file will consist of 72 records with a logical record length of 256. On a standard 5" double density disk, this translates to 12 granules or four cylinders. Since the Snapp BASIC extensions use an overlay scheme to minimize memory requirements, placement of the SNAPPINC file can have a direct bearing on processing efficiency. The preferred placement would be as close as possible to the directory cylinder. It is desirable, but not essential, for the SNAPPINC file to consist of a single extent and begin on a cylinder boundary. It is mandatory that the SNAPPINC file contain no more than four extents. With these factors in mind, you should examine a FREE space map for the target disk and decide where to place the SNAPPINC file before starting the actual installation process. Placement of the SNAPPINC file may be accomplished in one of two ways:
1) If a file named SNAPPINC already exists on the target disk, that starting location will be used.
2) If the SNAPPINC file does not exist, you will be prompted for a starting cylinder number.

The installation sequence for new users is as follows:

1. Place the target LDOS system disk in physical and logical drive Ø. This disk must contain the LBASIC/CMD file and the system overlays SYS2/SYS, SYS3/SYS, SYS8/SYS, and SYS1Ø/SYS. These overlays may be SYSRESed rather than being present on the disk so long as HIGH$ is greater than X'CØØØ'.

2. Place the SNAPP distribution disk in drive 1.
   a. If your target disk DOES NOT contain LBASIC 5.1.2, execute the following two commands:
      COPY LBASIC/CMD.RSØLTØFF:1 :Ø (CLONE)
      COPY LBASIC/OV3.RSØLTØFF:1 :Ø (CLONE)
   b. If your target disk DOES contain LBASIC 5.1.2, execute the following command:
      PATCH LBASIC.RSØLTØFF XBPATCH1

3. Upon receiving LDOS. Ready, place the distribution master in drive Ø and press the RESET button. In a few seconds, the EXTEND BASIC COPY utility will prompt you to place your target disk in drive Ø.

4. If the SNAPPINC file does not exist on your target disk, you will be asked to enter the "Starting Cylinder" for this file. Enter the value previously determined from your examination of the target disk FREE space map.

5. The message "Copy Complete" indicates successful conclusion of the installation process. You will then be prompted to place a system disk in drive Ø and press ENTER. Following a boot sequence to re-establish LDOS control, your Snapp BASIC product is ready to use.

All Snapp BASIC products included in a single order will normally be placed on one distribution master. If, however, you were to purchase different Snapp BASIC products at different times, you would receive a separate distribution master for each order. To simplify the installation process in this situation, we have implemented a "distribution merge" facility which allows you to combine the Snapp BASIC products onto a single distribution

master. In the following MERGE instructions, we will refer to the most recently acquired distribution master as the NEW disk, and the older one as the OLD disk:

1. Place a standard 5" double density LDOS system disk in physical and logical drive Ø and obtain the LDOS Ready prompt.

2. Place the NEW disk in drive Ø and press the RESET button.

3. When the EXTENDED BASIC COPY utility program prompts you for the target disk, place the OLD disk in drive Ø and press ENTER.

4. In a few seconds, the copy utility will prompt you for the distribution disk. Re-mount the NEW disk in drive Ø and press ENTER.

5. The message "Merge Complete" indicates successful conclusion of the merge process. You will then be prompted to place a system disk in drive Ø and press ENTER. Following the boot sequence to re-establish LDOS control, you may proceed to the normal "installation sequence for new users."

When you have completed this merge process, the OLD disk is no longer a distribution master. You may discard it or use it for another purpose. The NEW disk is now the DISTRIBUTION MASTER for all Snapp BASIC products that you have purchased.

HARD DISK INSTALLATION:

The special hard disk version can ONLY be installed on hard disk. The target must be an LDOS system disk, normally used as logical drive Ø, and must contain the LBASIC/CMD and system overlay files specified in item 1 of the normal installation sequence.

1. Perform one of the actions specified in item 2 of the normal installation sequence to place a patched copy of LBASIC 5.1.2 on the target hard disk.

2. Arrange the logical drive configurations such that physical and logical drive Ø contains a 5" double density LDOS system disk. Logical drive 1 should reference the target hard disk. Execute the DEVICE library command to verify these assignments.

3. Proceed to item 3 in the normal installation sequence. You will NOT be prompted for the target disk as it is assumed to be at logical drive 1. In item 5 of the normal installation sequence, at the prompt for a system disk in drive Ø, insert your standard LDOS 5" disk which automatically switches logical drive Ø to the hard disk.

INSTALLATION OF A SUBSET OF SNAPP PRODUCTS:

A facility is provided which will permit you to install some, but not all of the Snapp BASIC products which are included on your distribution master. This facility is engaged by pressing the CLEAR key in response to the prompt 'Place TARGET diskette in drive Ø, then press ENTER" in step 3 of the normal installation process. Then a series of prompts will appear, for those modules contained on the distribution diskette:

```
Disable SNAPP II?
Disable SNAPP III?
```

and so forth. To these prompts, a response of the CLEAR key will disable the specified module, while a response of the ENTER key will leave the module enabled.

Introducing the Trial Package

Good software is not cheap. While our products carry a high price tag, our customers agree that the value contained therein greatly exceeds the cost.

Since most microcomputer software is sold by mail order, a problem is created by (appropriately) high priced items. The prospect of purchasing a $19.95 game sight unseen would not faze most users, but spending several hundred dollars certainly will.

Our answer to this dilemma is the trial package. The purchaser of this package receives, for a very low investment, the opportunity to fully evaluate our most popular products in his own environment.

The distribution master included with the trial package may be used to apply these program products to ONLY ONE LDOS diskette. Additionally, the single working copy created therefrom will last for only a limited period of time.

Memory Requirements

Other versions of Snapp BASIC are able to provide the additional features with NO decrease in memory available to the user. This is due to certain coding practices in Disk BASIC which left "holes" that Snapp BASIC can use for its resident code.

The clean and efficient code of LBASIC, however, does not provide us with this luxury. In this system, the Snapp BASIC resident control routines occupy 574 bytes of user memory. This space is automatically reserved during LBASIC initialization in the area immediately following LBASIC itself.

Once the resident control routines are in place, most of the functions of the Snapp BASIC family are accomplished via overlay techniques, and will not require further dedication of memory. Two of the facilities, however, will require modest amounts of dedicated memory to achieve a reasonable performance level. Details are in the descriptions of SNAPP FOUR and SNAPP SIX.

Notation

Generally, the following conventions will be used in this document:

CAPITAL LETTERS
indicate material which must be entered exactly as shown.

lowercase letters
represent words or values you supply from the acceptable values for a particular command.

... (ellipsis)
indicates that the preceding items may be repeated.

[] (square brackets)
indicate that the material contained therein is optional.

<key>
specifies that the indicated key or keys are to be pressed.

Ø
indicates a required blank.

SNAPP TWO - EXTENDED BASIC
Program Development Aids

GENERAL INFORMATION:

SNAPP-II is a group of extensions to Model III LBASIC which extend the function and operational ease of this interpreter. The system is written entirely in machine language for SUPER FAST execution of your commands! The system consists of six modules, all of which are invoked directly from the keyboard. The resident control routines for the entire Snapp BASIC family occupy 574 bytes of memory. This space is automatically reserved during LBASIC initialization in the area immediately following the LBASIC program.

The six modules, and their major function:

XBASIC: General keyboard functions.
XREF: Variable/line number cross reference.
XDUMP: Dynamic dump of the values of variables.
XFIND: A cross reference facility for strings and keywords.
XRENUM: A better renumbering facility.
XCOMPRESS: A facility to significantly reduce the memory requirements of a program.

SINGLE STEP TRACE

An interesting feature of this product, not particularly related to the six main modules, is a single step trace facility.

SYSTEM TRON [lineno] enables Single Step Trace. Just before each line is executed, the line number is displayed in the upper right corner of the screen and execution is suspended until keyboard input is detected. Press ANY key to continue. If the BREAK key is pressed, it will be handled normally, but the line number in the Break message will be that of the LAST line executed, while the Trace display will show the line number of the NEXT line which would have been executed. When any key is held down, lines will be executed at the normal keyboard repeat rate. The optional line number parameter will cause the Trace operation to be initiated only when the NEXT line to be executed has a line number equal to that specified. The line numbers will be displayed on the second line of the screen so as not to conflict with the clock display.

SYSTEM TROFF disables Single Step Trace.

Each of the above commands may be entered either as program statements or as immediate commands.

THE SAMPLE PROGRAM:

We have set up the following program to demonstrate the functions performed by the various processing modules in SNAPP-II. Don't try to 'understand' this program, because it just doesn't do anything! The statements were chosen simply to highlight the utility functions available. Please note that the example includes a command from one of our other products, EXTENDED BUILTIN FUNCTIONS, and that unless you have also purchased that product, this (FN FILES) will not function separately. Furthermore, XREF and XCOMPRESS handling of the FN FILES statement will depend on the presence or absence of the EXTENDED BUILTIN FUNCTIONS program product.

```
1Ø AB = 4
2Ø HN = &HFFFF
3Ø 'THIS IS A REMARK
4Ø PRINT FN FILES
5Ø B$ = "THIS IS A CHARACTER STRING"
6Ø CD = 5ØØØØØØ
7Ø IF AB > CD THEN GOTO 15Ø
8Ø DIM J(5)
9Ø FOR I% = 1 TO 5
1ØØ J(I%) = I%^2
11Ø NEXT I%
12Ø GOTO 15Ø
13Ø A$ = "THIS IS A CHARACTER STRING" + ", TOO!"
14Ø YZ = 198Ø
15Ø PRINT "END"
```

XBASIC:
GENERAL KEYBOARD FUNCTIONS

The functions performed by XBASIC itself are abbreviations for commonly used commands, of which twelve are implemented.

A	AUTO
C	CLS
D	DELETE
E	EDIT
K	KILL
L	LIST
L"	LOAD"
M	MERGE
N	NEW
P	LLIST
S	SYSTEM
V	SAVE

The above abbreviations, when they appear as the first character of a command line, will function as if the full word had been keyed in. As with the full command name, blanks following the abbreviation may be included or omitted.

Additionally, XBASIC supports an 'UN-NEW' function which will restore the resident BASIC program following an unintentional NEW, an accidental CMD"S", or even a system RESET. This function is invoked using the letter 'U', and MUST BE THE FIRST COMMAND ISSUED FOLLOWING WHATEVER DISASTER HAS OCCURRED. If BASIC has been re-entered following a CMD"S" or a re-boot, the number of files MUST be the same as it was before. Completion of the UN-NEW function will be followed by an automatic LIST of the recovered material so that you may visually verify the results. Please note that certain types of failures, such as defective memory, can cause total garbage to be produced.

XREF:
THE CROSS REFERENCE FACILITY

The purpose of this routine is to produce a listing of any/all variables/line numbers used within the resident BASIC program, with information about each location within the program where the variable/line number is used. Most programmers find a cross reference facility to be indispensable, and consider the cross reference printout to be a standard part of a program listing.

This facility is more comprehensive than the CMD"X" function provided with LBASIC, in that it distinguishes between simple references and updates to a variable. This can be very helpful when debugging in that if you are trying to locate the cause of a corrupted variable, you may limit your search to those lines which modify that variable. Additionally, this facility is significantly faster than CMD"X". Most users will want to free up disk space by killing LBASIC/OV2.

XREF is invoked using the abbreviation 'X', usually followed by parameters. There are nine options (spacing between the 'X' and any parameters is optional):

```
X .          List all references to screen.
X ,          List all references to printer.
X .vv        List references starting with vv to screen.
X .nnnnn     List references starting with nnnnn to screen.
X ,vv        List references starting with vv to printer.
X ,nnnnn     List references starting with nnnnn to printer.
X vv         List only references to vv to screen.
X nnnnn      List only references to nnnnn to screen.
X            List 'next' program line containing a reference to the last 'X
vv' or 'X nnnnn' command.
```

In the above formats, the symbol vv represents any variable name, and may be either one or two characters. The symbol nnnnn represents any (line) number, and may be from one to five characters.

FORMAT OF THE LISTINGS:
Each listed variable will show the variable name at the left margin, followed by one or more reference entries. A reference entry is as follows:

<*>nnnnn</(><%$!#><nn>

where the *, if present, means, for a number, that it is a line number rather than an integer constant, and for a variable, that it is modified in the referenced line, nnnnn is the line number in which the reference(s) occur, the /, if present is simply a separator, the (if present, indicates that this was an array reference, the % or $ or ! or #, if present was the typing character found, and nn, if present is the number of references in this line (if nn is not present, there was only one reference in this line).

The standard LBASIC Pause and Break key processing is utilized, therefore:
1) Press <SHIFT @> to temporarily suspend the listing.
2) Press any key except <SHIFT @> to resume a suspended listing.

3) Press the BREAK key at any time to terminate the listing and return to the "Ready" prompt.
4) The "LBASIC SINGLE STEPPING" feature may be used.
5) Pressing either <SHIFT @> or BREAK keys will remove all characters from the typeahead buffer.

AN EXAMPLE:

If the command X , is given with our sample program resident, the following printout will be produced:

```
    1  9Ø
    2  1ØØ
    4  1Ø
    5  8Ø 9Ø
  15Ø  *7Ø *12Ø
 198Ø  14Ø
"FILES  4Ø
&HFFFF  2Ø
   A   *13Ø/$
   AB  *1Ø 7Ø
   B   *5Ø/$
   CD  *6Ø 7Ø
   HN  *2Ø
   I   *9Ø/%
   J   8Ø( *1ØØ(
   YZ  *14Ø
```

Note that the special 'SNAPP keywords' from our EXTENDED BUILTIN FUNCTIONS product will be referenced following the number references. Individual reference listings of 'SNAPP keywords' may also be obtained by preceding the keyword with a quote.

Hex and/or Octal constants will appear in the listing following listings of 'SNAPP keywords', if any.

AN EXAMPLE:

If the command Z , is given with the sample program resident and RUN, the following printout will be produced:

```
A
AB !  4
B  $  "THIS IS A CHARACTER STRING"
CD !  5E+Ø6
HN !  -1
I  %  6
J  !  (Ø) Ø  (1) 1  (2) 4  (3) 9  (4) 16  (5) 25
YZ
```

Note that variables which have never been initialized are shown without typing characters or values.

XDUMP:
THE DYNAMIC VARIABLE PRINT FACILITY

The purpose of this routine is to allow the programmer to easily list to the video or printer all variables used in the program, ALONG WITH THEIR CURRENT VALUES. This routine can greatly simplify debugging.

XDUMP is invoked using the 'Z' command, usually followed by parameters. Six parameter formats are supported:

```
Z .     List all variables to video screen.
Z ,     List all variables to printer.
Z vv    List only the named variable to video screen.
Z .vv   List beginning with the named variable to the video screen.
Z ,vv   List beginning with the named variable to the printer.
Z       With no parameters, same as Z .
```

Non displayable characters contained within string variables will appear as a period. Array variables will have each member listed separately. Arrays up to 1Ø dimensions are supported.

FORMAT OF LISTING:

Each variable will be listed starting on a new line, followed by the type character (% = integer, $ = string, ! = single float, # = double float), followed by its current value. Array members will be listed the way they are stored in memory, column major order, with the left most subscript varying most frequently.

The standard LBASIC Pause and Break key processing is utilized, therefore:
1) Press <SHIFT @> to temporarily suspend the listing.
2) Press any key except <SHIFT @> to resume a suspended listing.
3) Press the BREAK key at any time to terminate the listing and return to the "Ready" prompt.
4) The "LBASIC SINGLE STEPPING" feature may be used.
5) Pressing either <SHIFT @> or BREAK keys will remove all characters from the typeahead buffer.

XRENUM:
THE ENHANCED RENUMBERING FACILITY

This is an expanded program line renumbering facility which provides significant advantages over the CMD"N" feature of LBASIC. Most users will want to free up disk space by killing LBASIC/OV1. The specific enhancements provided by XRENUM are:

1) Allows relocation of blocks of program code.
2) Provides the capability of duplicating blocks of code.
3) Modifies line number references in ERL statements which follow the AND or OR operators.
4) Correctly modifies line number references in LIST and DELETE statements which have the form .-nnnnn.
5) Modifies line number references in LLIST statements.
6) Correctly updates the current line number pointer to the resequenced value.
7) Displays informational messages showing the number of program lines and size of program text before and after updates.
8) The speed of the renumbering is much better than that of CMD"N", and the improvement ratio seems to grow geometrically with the size of the program. For very large programs, this function may run as much as 2Ø times as fast as CMD"N".

XRENUM is invoked using the abbreviation 'R', usually followed by parameters. There are four optional forms of invocation (note that spacing between the 'R' and any parameters is optional):

1) R U Scans the program text for undefined line numbers or other errors in statements which reference line numbers. The program text is not changed. Errors encountered are displayed in the following format:

nnnnn/U - Line number nnnnn is referenced but does not exist in the program.
nnnnn/X - Line number nnnnn contains a statement which requires reference to a line number, but that line number reference was not found.
nnnnn/S - Line number nnnnn contains a statement which references a line number but that referenced number is not a valid line number. (Not within the range 1-65529)

2) R newline,increment,startline,endline Causes all program lines with a line number >= "startline" and <= "endline" to be assigned new line numbers beginning with "newline" with subsequent line numbers generated by adding "increment". Each parameter must be a number in the range 1-65529. The default value for "startline" is Ø; the default value for "endline" is 65529; and the default values for "newline" and "increment" are 1Ø. The range of newly generated line numbers must not encompass any old text lines that are not part of the resequence range "startline" - "endline" inclusive. So long as this rule is observed, the newly generated line number range may be placed anywhere in the program. The renumbered block of text will be moved to the proper location after references to the renumbered program lines have been

altered. If any error of the type outlined in option 1 is encountered before the text is altered, this command reverts to option 1 and the program text is not changed. Note that this form of the command can easily be used to perform a "destructive copy" of a block of program text, that is to say the code is moved to the new location and deleted from the old location.

3) R I newline,increment,startline,endline Duplicates the block of program text with line numbers >= "startline" and <= "endline", assigning line numbers beginning with "newline" and incremented by "increment". This option differs from option 2 in that the old block of program text is not renumbered or moved and no changes are made to line number references. Note that this form of the command is used to perform a "nondestructive copy" of a block of program text, that is to say the code is moved to a new location but also left in the old location.

4) R X newline,increment,startline,endline Exactly the same as option 2, with the exception that undefined line errors will not prevent the completion of the renumbering operation. This option is particularly useful when working on a program which references subroutines which have not yet been written.

Please note that if 'startline' and 'endline' are identical in any of the above formats, that the desired operation will be performed on the indicated single line.

If you wish to suppress the appearance of the information messages which normally appear on the display during the renumbering process, you may insert the letter 'M' after the 'R' and before any other options, if present.

Error conditions which may be encountered during R processing, and their messages:
Syntax Error - An invalid parameter has been entered.
Can't process line Ø - A program containing line number Ø cannot be processed by XRENUM.
Seq # overflow - The range of newly generated line numbers either encompasses a line number in the original program or exceeds the maximum valid number of 65529.
Program text error - The "next line address" contained within each program line does not agree with the actual end of the program line. Might be caused by bad memory.
FATAL ERROR - TEXT NOW BAD - An error has been encountered from which no recovery is possible. An exit to LDOS is made to ensure that the erroneous text is not used. Almost certainly caused by bad memory.
Out of memory - There is insufficient free memory to process the request; reducing the amount of string space will increase the available free memory.
No program - There is no program in the text buffer.
Error lines: - Identifies the set of program text errors outlined in option 1.

AN EXAMPLE:

If the command R I 122,2,9Ø,11Ø is given with the sample program resident, the following program will result:

```
1Ø AB = 4
2Ø HN = &HFFFF
3Ø 'THIS IS A REMARK
4Ø PRINT FN FILES
5Ø B$ = "THIS IS A CHARACTER STRING"
6Ø CD = 5ØØØØØ
7Ø IF AB > CD THEN GOTO 15Ø
8Ø DIM J(5)
9Ø FOR I% = 1 TO 5
1ØØ J(I%) = I%^2
11Ø NEXT I%
12Ø GOTO 15Ø
122 FOR I% = 1 TO 5
124 J(I%) = I%^2
126 NEXT I%
13Ø A$ = "THIS IS A CHARACTER STRING" + ", TOO!"
14Ø YZ = 198Ø
15Ø PRINT "END"
```

XFIND:
THE STRING/KEYWORD CROSS REFERENCE FACILITY

The purpose of this routine is to produce a listing of any/all strings and/or keywords used within the resident program. The functions provided are identical to those performed for variables and integer constants (line numbers) by XREF.

XFIND is invoked using the 'F' command, usually followed by parameters. In the format specifications, 'kw' refers to a BASIC keyword (like GET or LPRINT), and 'ss' refers to any user specified character string or substring (like 'NOW IS THE TIME'). When constructing a search string, the commercial at sign (@) character is used as a "wild card" or "don't care" character, and will logically compare equal with any character appearing in that relative position within the text string. There are eleven options (note that spacing between the 'F' and any parameters is optional):

```
F .          List all keyword references to screen.
F ,          List all keyword references to printer.
F ."kw       List all keyword references starting with 'kw' to screen.
F ,"kw       List all keyword references starting with 'kw' to printer.
F ."ss"      List all string references starting with 'ss' to screen.
F ,"ss"      List all string references starting with 'ss' to printer.
F "ss"       List all string references which CONTAIN 'ss' to screen.
F "kw        List all references to 'kw' to screen.
F "kw1&kw2   List all references in which kw1 and kw2 appear in the same pro-
gram line to screen.
F "kw1,kw2   List all references to kw1 to screen and REPLACE each occurrence
of kw1 with kw2.
F            List 'next' program line containing a reference to the last 'F
"ss"' or 'F "kw' command.
```

In the command formats, the presence or absence of a trailing quote mark distinguishes a keyword from a string reference. Both need a leading quote mark, and string references must have a trailing quote mark, while keyword references must not. In the reference listing, the apostrophe (') abbreviation for :REM will not identified separately, but combined with REM in the listing.

The standard LBASIC Pause and Break key processing is utilized, therefore:
1) Press <SHIFT @> to temporarily suspend the listing.
2) Press any key except <SHIFT @> to resume a suspended listing.
3) Press the BREAK key at any time to terminate the listing and return to the "Ready" prompt.
4) The "LBASIC SINGLE STEPPING" feature may be used.
5) Pressing either <SHIFT @> or BREAK keys will remove all characters from the typeahead buffer.

EXAMPLES:

If the command F , is given while the sample program is resident, the following printout will result:

```
+         13Ø
=         1Ø 2Ø 5Ø 6Ø 9Ø 1ØØ 13Ø 14Ø
>         7Ø
DIM       8Ø
FN        4Ø
FOR       9Ø
GOTO      7Ø 12Ø
IF        7Ø
NEXT      11Ø
PRINT     4Ø 15Ø
REM       3Ø
TO        9Ø
^         1ØØ
```

While the command F ,"" will produce this printout:

```
", TOO!"  13Ø
"END"  15Ø
"THIS IS A CHARACTER STRING"  5Ø 13Ø
```

XCOMPRESS:
THE PROGRAM OPTIMIZATION FACILITY

The purpose of this routine is to reduce to an absolute minimum the size of the resident BASIC program for optimal execution using the Model III interpreter. Programs which have been so compressed typically occupy 3Ø to 4Ø% less memory space, and run 7 to 1Ø% faster.

Optimization consists of several phases, most of which are optional or chosen entirely by the user's specifications:

1) Removal of remarks.
2) Removal of irrelevant blanks.
3) Removal of irrelevant tab characters.
4) Removal of extraneous colons.
5) Removal of the LET keyword.
6) Removal of quote marks at the end of a line.
7) Removal of GOTO in the sequences 'THEN GOTO' and 'ELSE GOTO'.
8) Removal of non-significant characters from variable names when the length of the variable name exceeds two.
9) Removal of completely non-executable code.
1Ø) Removal of variable typing characters (%,$,!,#) when a previous DEFINT, DEFSTR, DEFSNG, or DEFDBL statement makes such explicit typing redundant.
11) Merging multiple statements into single lines.
12) Renumbering the program on a 1 x 1 basis to make line number references as small as possible.
13) Removal of specific variable identifiers from NEXT statements.

Numbers 4, 6, and 7 above are not optional. All of the other functions are controlled by the options. Unless otherwise specified, item 13 is OFF, and all other options are ON.

Although item number 13 normally saves a few bytes, the primary objective in removing the variable identifiers is to improve the performance of the program. Our benchmarks show that FOR - NEXT loops execute up to 25% faster with the variable identifiers omitted.

XCOMPRESS is invoked using the abbreviation 'H', optionally followed by one or more parameters to specify your requirements for compression. Multiple options, if present, must be separated by commas.

The available options:
NC - Specifies that you do not want multiple lines (#11) merged together. Also inhibits removal of non-executable code (#9).
NR - Specifies that you do not want the compressed program renumbered on a 1 x 1 basis (#12).
LB - Specifies that you do not want blanks (#2) and tabs (#3) removed. Also inhibits shortening long variable names (#8) and removal of LET keywords (#5). Under certain circumstances, a very few blanks may be removed, even with this option set.
LR - Specifies that you do not want remarks (#1) removed. Also inhibits removal of non-executable code (#9).

LT - Specifies that you do not want variable typing characters (#1Ø) removed. In order for this option to function correctly, the DEFxxx (where xxx = INT, STR, SNG, or DBL) statements must appear in a physical order corresponding to their logical order. If DEFxxx statements appear in subroutines which are referenced by GOSUB statements, those lines which appear in the program physically before the DEFxxx statements will be treated as if the DEFxxx statement was not in effect. We suggest that you place your DEFxxx statements very near the beginning of the program to take maximum advantage of this feature.
RV - Specifies that you wish the specific variable identifiers removed from NEXT statements. In most cases, the only difference that will be noticed from the use of this option will be faster execution. If, however, your program executes a GOTO statement referencing a NEXT with a named variable which is the control variable of an 'outer' loop, and depends upon the implied 'abnormal termination' of an un-named inner loop, the program will no longer function identically. We have never seen a 'real-world' situation where this occurred, but be aware of the theoretical problem.

If you wish to suppress the appearance of the information messages which normally appear on the display during the compression process, you may insert the letter 'M' after the 'H' and before any other options, if present.

A WORD OF CAUTION:
You should retain your uncompressed code for two very good reasons:
1) A fully compressed program is hardly readable, and performing maintenance on it would be very difficult.
2) Because of the manner in which the BASIC interpreter tokenizes statements, under certain circumstances EDITing or SAVEing in ASCII can change the syntax of a compressed program, and introduce syntax errors. As long as you don't do either of these, the compressed program will run quite nicely, but be aware of the hazards. An example of the type of code which will introduce this error follows:

Your program contained:

```
    1Ø IF S = T AND U = V THEN W = X
```

The compression routine changed it to:

```
    1Ø IFS=TANDU=VTHENW=X
```

Which did not cause any problem until you, for example, save the line in ASCII. When you reload it, BASIC sees (blanks and ?? added for clarity): IF S = TAN ?? DU = V THEN W = X. As you can see, BASIC will misinterpret this line, believing that the keyword TAN is incorrectly used.

To summarize the above caution:
DON'T *FIDDLE* WITH A PROGRAM AFTER IT HAS BEEN COMPRESSED. WE CAN NOT GUARANTEE THE RESULTS, BUT THEY WILL PROBABLY BE BAD.

AN EXAMPLE:

If the command H is issued with the sample program resident, the following program will result:

```
1 AB=4:HN=&HFFFF:PRINTFNFILES:B$="THIS IS A CHARACTER STRING":CD=5ØØØØØØ:IFAB
>CDTHEN3
2 DIMJ(5):FORI%=1TO5:J(I%)=I%^2:NEXT:GOTO3
3 PRINT"END
```

Note that the previous contents of lines 13Ø and 14Ø have been completely removed under rule #9.

SNAPP THREE - EXTENDED BUILTIN FUNCTIONS (XBIF)
A Set of Language Extensions for the Model I/III BASIC Interpreter

GENERAL INFORMATION:

This product is a collection of much needed additions to the Model III interpreter which will greatly extend its convenience and utility. These features, when installed, become a part of your BASIC language.

The most important component of this product is a SUPER FAST in-memory sort routine. We have benchmarked this against everything on the market, and beat them all hands down. In addition, our sort is far and away the EASIEST TO USE and MOST GENERALIZED of anything available.

Many of the facilities provided are invoked as FUNCTIONS, that is to say, their use is preceded by the FN keyword, most of which require a parenthesized ARGUMENT LIST, and they RETURN A RESULT. Unlike user defined functions, however, you will not be required (in fact must not) issue a DEF FN statement to establish the existence of the functions. Hence the name BUILTIN FUNCTIONS.

The general format for invocation of a BUILTIN FUNCTION is:

```
variable = FN builtin-function-name (argument...)
```

where 'variable' is any variable of your choice (of the appropriate type, string or numeric, depending upon the function) to receive the RESULT, and the arguments are either STRING EXPRESSIONS or NUMERIC EXPRESSIONS (again, depending upon the function). In keeping with the philosophy of the interpreter, blanks may be included or omitted. Please note that BUILTIN FUNCTIONS may not be nested within a single statement. Generally, any attempt to use two BUILTIN FUNCTIONS within the same statement will produce an Illegal function call error.

The remaining facilities are implemented as VERBS. We have chosen the SYSTEM verb for this purpose. The general format for invocation of *a SYSTEM COMMAND* is:

```
SYSTEM command [operator-1[,operator-2]...]
```

where 'command' is either a previously defined BASIC keyword or a string literal SNAPP keyword, and operators are defined by the specific COMMAND.

If you don't already have a clear understanding of an EXPRESSION, we suggest you read pp 1/29 & 1/3Ø of the Model III BASIC REFERENCE MANUAL. From this point, when we refer to 'stexp', we mean a STRING EXPRESSION, and when we refer to 'nmexp', we mean a NUMERIC EXPRESSION. The judicious use of EXPRESSIONS can greatly simplify your programming chores.

FN PEKW

Extract Two Bytes (LSB,MSB) From A Specified Memory Location

SYNTAX:

PEKW(nmexp)

RETURNS:

number

'nmexp' is evaluated as an INTEGER and the value contained in the WORD (LSB,MSB format) is returned.

EXAMPLE:

```
II = FN PEKW (&H4411)
```

Places into II the address of the higest memory location not used by LDOS. This value is usually called HIGH$.

POSSIBLE ERRORS:

Overflow - The argument could not be converted to INTEGER.
Syntax Error - Required punctuation not found in the expected place.
Type mismatch - The argument is not a number.

SYSTEM POKE

Replace The Contents Of A Specified Memory Area With The Supplied Value

SYNTAX:

SYSTEM POKE nmexp1,exp2

RETURNS:

nothing. This is a verb, not a function.

'nmexp1' is the starting address of the memory area to be modified and is evaluated as an INTEGER. If exp2 is a string expression, POKE the string to the specified address. If exp2 is a numeric expression, convert it to an integer value *and* POKE the word in LSB/MSB format.

EXAMPLE:

```
SYSTEM POKE &H42ØØ,STRING$(255,Ø)
```

Will ABSOLUTELY DESTROY LDOS.

POSSIBLE ERRORS:
Overflow - Exp2 was numeric, but could not be converted to INTEGER.
Syntax Error - Required punctuation not found in the expected place.
Type mismatch - Nmexp1 is not a number.

FN ETIM$

Calculate The Difference Between Two Times

SYNTAX:

ETIM$(stexp1[,stexp2])

RETURNS:

string

'stexp1', and if present, 'stexp2' are 17 byte character strings containing a date/time as returned by TIME$. If 'stexp2' is provided, 'stexp1' is the STARTING TIME and 'stexp2' is the ENDING TIME. If 'stexp2' is omitted, then 'stexp1' is the STARTING TIME, and the current time of day is the ENDING TIME. In either case, the second value MUST be greater than the first. This function is designed to calculate the elapsed time for only a relatively short period (less than 48 hours); therefore, if the date portions of the two strings are not equal, the function assumes that the second time is ONE day later than the first and adds 24 hours to the second time before calculating the difference. The difference between the STARTING TIME and the ENDING TIME is calculated, and returned as an 8 byte character string in the format of the right-most part of the arguments.

EXAMPLE:

```
PRINT FN ETIM$("Ø8/2Ø/81 Ø9:ØØ:ØØ","Ø8/2Ø/81 11:ØØ:ØØ")
```

Will print 'Ø2:ØØ:ØØ'.

POSSIBLE ERRORS:

Syntax Error - Required punctuation not found in the expected place.
Type mismatch - The arguments are not strings.
Illegal function call - Probably your argument(s) contain an invalid time.

SAMPLE PROGRAM:

```
1Ø ST$ = TIME$
2Ø FOR I = 1 TO 1ØØØØ
3Ø NEXT I
4Ø PRINT FN ETIM$(ST$)
5Ø END
```

Calculates the time required to do 1ØØØØ FOR-NEXT iterations with a single precision loop variable (it printed 'ØØ:ØØ:28').

FN FILES

Return The Number Of File Blocks Currently Allocated

SYNTAX:

FILES

RETURNS:

number

The number of files which may be opened concurrently is returned. If the Blocked file mode (BLK=ON) was specified (the default), the number is indicated as a negative value.

EXAMPLE:

```
PRINT FN FILES
```

Will display the number of file blocks allocated. If the returned value is negative, the files may be opened with a user-specified LRL.

SYSTEM "SORT"

Sort One Or More Arrays Into Specified Sequence

SYNTAX:

SYSTEM "SORT",stexp[,nmexp1[,nmexp2]]

RETURNS:

nothing. This is a VERB, not a function.

'stexp' is a string expression containing from 1 to 32 iterations of the sequence "[+-] [*]array name[,...]" The + or - indicates that (1) this array is to participate in the 'sort key', which means that helps determine the final order of the results, and (2) indicates whether the sort logic for this array is to be ascending or descending. The first array name which is not prefixed by a sign terminates key construction, and that and all subsequently named arrays become 'tagalongs' to the sorting process; which means that they will be sorted, but will not participate in the decision process. The *, if present indicates 'special sequence mode' for that array only. Special sequence mode is meaningful only for strings and integers. For strings, special sequence mode means that a shorter string is to be logically padded on the right with blanks when making the comparison to longer strings. Without this special sequence mode, "X" will precede "X ", according to BASIC's standard comparison algorithm. With special sequencing, they will compare equally. For integers, special sequence means to treat the numbers as unsigned binary, rather than signed binary. This will be most useful when sorting machine addresses.

'nmexp1', if present, specifies the lowest numbered element of the arrays to participate in the sort. If you omit this specification, it will default to 1. If you use the zero elements of your arrays, be sure to specify this value, or the zero elements won't get sorted.

'nmexp2', if present specifies the highest numbered element of the arrays to participate in the sort. If you omit this specification, it will default to the minimum size of the arrays listed in 'stexp'.
The arrays to participate in the sort may be multi dimensioned arrays, but they will be treated by the sort as if they were singly dimensioned arrays.

EXAMPLE:

```
SYSTEM "SORT","+A%"
```

Will cause the array A% to be sorted in ascending sequence, with all elements of the array participating except the zero element.

POSSIBLE ERRORS:
Syntax Error - Required punctuation not found in the expected place.
Type mismatch - 'stexp' is not a string expression, or 'nmexp1' or 'nmexp2' are not numeric expressions.
Overflow - 'nmexp1' or 'nmexp2' could not convert to integer.
Illegal function call -

1) 'nmexp1' or 'nmexp2' is negative.
2) 'nmexp2' is greater than the size of the smallest array specified in 'stexp'.
3) 'nmexp1' is greater than 'nmexp2'.
4) 'stexp' contains no array names, or an invalid array name.
5) 'stexp' contains more than 32 array names.

A SAMPLE PROGRAM:
Is in the section 'SOME REAL-WORLD SAMPLES' of this document.

SYSTEM CLEAR

This command allows you to specify the number of file blocks to be allocated at the same time that you allocate string space. An additional optional parameter allows you to to change the high memory address used by BASIC.

SYNTAX:

SYSTEM CLEAR [nmexp1] [,nmexp2] [,nmexp3]

RETURNS:

nothing. This is a verb, not a function.

Nmexp1 is the new value for number of file blocks. A positive value sets the BLK=OFF file mode while a negative value sets BLK=ON (this is the default setting on entry to LBASIC). The value of nmexp1 may range from -15 to 15.

Nmexp2 is the string space specification as in the BASIC CLEAR statement.

Nmexp3 provides a means for changing the high memory address (HIMEM) used by BASIC. Three different change methods are provided, depending upon the parameter value:

1) If nmexp3 = Ø, set BASIC HIMEM = LDOS HIGH$.
2) If nmexp3 <= &HØØØ, subtract that value from the current BASIC HIMEM and use the result as the new BASIC HIMEM.
3) If nmexp3 > &HØØØ, set BASIC HIMEM to the specified value minus one, subject to the checks that there is sufficient memory for the new value and that it does not exceed LDOS HIGH$.

A null parameter will be left unchanged; therefore the statement SYSTEM CLEAR Ø would change only the number of file blocks, while the statement SYSTEM CLEAR ,,&HEØØ would change only BASIC's HIMEM.

This statement will always CLOSE all files and CLEAR all variables. As a result, it will normally be placed at or near the beginning of a program.

To facilitate 'location independence', small values (less than &HØØØ) for nmexp3 are interpreted as requests to reduce BASIC's HIMEM by that amount. This use can make the BASIC application less dependent upon changes in LDOS drivers & filters. A very practical example might be to have a 'starter' program which is executed only once as BASIC is initialized that will invoke this command specifying &H24Ø as nmexp3 (space required by the College Educated Garbage Collector), invoke CEGC, then again invoke this command specifying &H38Ø as nmexp3 (space required for Extended BASIC Mapping Support).

In processing this statement, all files are closed, nmexp1 is evaluated, a test is made for sufficient memory to support the new file buffer areas, the file address table is reset, the entire program is moved to its new location, and the text line pointer is updated to reflect the new program location. Finally, nmexp2 and nmexp3 are processed, if present.

EXAMPLE:

SYSTEM CLEAR -5,1ØØØ,&HF8ØØ

Resets the number of file buffers to 5, enables blocked file mode, allocates

1ØØØ bytes of string space, and sets BASIC's HIMEM to &HF7FF.

POSSIBLE ERRORS:
Overflow - One of the arguments could not be converted to INTEGER.
Syntax Error - Required punctuation not found in the expected place.
Type mismatch - One of the arguments is not a number.
Illegal function call - Nmexpl is not within the range -15 to 15 or nmexp3 exceeds LDOS HIGH$.
Out of memory - Insufficient memory available to satisfy the request.
Missing operand - At least one of the three parameters must be specified.

SYSTEM "ERASE"

Remove any or all arrays from the array table, thereby freeing the space used and permitting them to be re-dimensioned.

SYNTAX:
SYSTEM "ERASE" [,array1,array2...]
RETURNS:
nothing. This is a verb, not a function.
If no array names are specified, all arrays are deleted, otherwise the listed arrays are removed from the array table.

FN ID$

Read disk names.

SYNTAX:
ID$(nmexp)
RETURNS:
string.
Nmexp specifies the drive number and must be in the range Ø-7. The function will return an 8 byte string containing the DISKID of the disk currently in the specified drive.

EXAMPLE:

```
XX$ = FN ID$ (Ø)
```

Will return an 8 byte string to XX$, containing the ID of the disk mounted in drive Ø.

POSSIBLE ERRORS:
Illegal function call - Nmexp not within required range.
Syntax error - Right parenthesis absent.
Type mismatch - Argument not a numeric expression.
Internal error - Error in attempting to read disk name.

FN PEK$

Extract multiple characters from a specified memory location.

SYNTAX:
    PEK$(nmexp1,nmexp2)
RETURNS:
    string.
Nmexp1 is the peek address. Nmexp2 is the number of bytes to PEEK into your string.

EXAMPLE:

```
PRINT FN PEK$ (&H4225,64)
```

Will display the LDOS command buffer. This will normally contain LBASIC (with options), with trailing garbage.

POSSIBLE ERRORS:
Illegal function call - Nmexp2 not in range 1-255.
Syntax error - Required punctuation not found where expected.
Type mismatch - One of the arguments is not a numeric expression.

FN UC$

Convert string to upper case.

SYNTAX:
    UC$(stexp)
RETURNS:
    string.
Each lower case alphabetic byte contained in the string is converted to upper case, by ANDing it with &HDF, and the result is returned.

EXAMPLE:

```
PRINT FN UC$ ("abcd")
```

Will print ABCD.

POSSIBLE ERRORS:
Type mismatch - Argument not a string expression.
Syntax error - Required punctuation not found where expected.

FN LC$

Convert string to lower case.
SYNTAX:
    LC$(stexp)
RETURNS:
    string.
Each upper case alphabetic byte contained in the string is converted to lower case, by ORing it with &H2Ø, and the result is returned.

EXAMPLE:
```
    PRINT FN LC$ ("ABCD")
```
Will print abcd.

Possible errors are as for UC$

FN MAX

Return the largest value from a user supplied list.

SYNTAX:
    MAX (nmexpl [,nmexp2]...)
RETURNS:
    number
each argument is converted to double precision, then the largest of them is selected, and re-converted to the numeric type of the returned variable.

EXAMPLE:
```
    PRINT FN MAX (2,7,4,9,1,8)
```
Will print the value 9.

POSSIBLE ERRORS:
Syntax error - required punctuation not found in the expected place.
Type mismatch - one of the arguments is not a number.
Overflow - The value selected by the function exceeds the limits of the return variable type.

FN MIN

Return the smallest value from a user supplied list.

SYNTAX:
    MIN (nmexp1 [,nmexp2]...)
RETURNS:
    number
processing identical to MAX.

POSSIBLE ERRORS:
As for MAX.

FN FMT$

Arrange data into a String variable, as with PRINT USING.

SYNTAX:
    FMT$ (stexp;item-list)
RETURNS:
    string.
'stexp' is the FORMAT SPECIFICATION, and item-list is the list of variables to be inserted into the FORMAT SPECIFICATION, as described in the BASIC manual pp 3/4 - 3/8. Two restrictions are imposed on the use of FN FMT$: 1) The generated string may not exceed 255 bytes. 2) The closing paren MUST be the last non-blank byte before the end of the statement (which is defined as end of line, or colon).

EXAMPLE:
    X$ = FN FMT$("HELLO ####";49;)
Will cause X$ to contain "HELLO   49".

Use your imagination with this function. It is probably the most powerful addition ever made to Microsoft BASIC.

POSSIBLE ERRORS:
Syntax error - Incorrect punctuation, or closing right paren is not immediately followed by end of statement.
Illegal function call - The portion of the statement following the FMT$ function contains another BUILTIN FUNCTION reference.
String too long - The generated string exceeds the 255 byte maximum string size.
Other errors may be generated by our implicit use of the PRINT USING routine.

Please note that unless the item list is terminated with a semicolon, the generated string will be terminated by a carriage return.

SYSTEM DELETE

Delete statements from the resident BASIC program.
SYNTAX:
SYSTEM DELETE nmexp1 [-[nmexp2]]
RETURNS:
nothing. This is a verb, not a function.
This verb allows you to delete statements from an executing BASIC program without a return to the "Ready" prompt. This might be used, for example, to delete DATA statements after they have been READ, thereby freeing up memory for other purposes. Parameters following the DELETE keyword are identical to those normally allowed for the DELETE statement, but the requirement that the end range line number be present in the program has been removed; therefore, SYSTEM DELETE 1ØØØ- will function as expected. Processing steps are as follows:

a. The parameters are evaluated and a Syntax error is generated if they are incorrect.
b. If the delete block end address is <= start address, an Illegal function call error is generated.
c. If the SYSTEM DELETE line is above the range of lines being deleted, the line pointer is reduced by the delete block size. If the SYSTEM DELETE line is below the range of lines being deleted, the line pointer is not changed. If the SYSTEM DELETE line is within the range of lines being deleted, the line pointer is reset to the first line following the delete block; therefore, the SYSTEM DELETE statement may delete itself.
d. The Variable Table, Array Table, and Free Space addresses are reduced by the delete block size.
e. The program and table data are moved down into the block of line areas being deleted.
f. The line address chain pointers are reset.

A Caution:
There is no examination of addresses held in the variable and array tables; it is therefore the user's responsibility that those addresses are not affected by the movement of program text down into the delete block area. There are at least three cases where addresses within the program area are contained in the variable or array tables: 1) String literals contained in the program. 2) The Function Name variable table entry contains the address of the DEF FN line which defines the Function Name. 3) The address of the statement which is the target of an ON ERROR GOTO statement is contained as control information within the interpreter.

A Suggestion:
Because of this characteristic, you would be well advised to place lines which contain DEF FN statements, or those which execute LET statements with string literals EARLY in the program, BEFORE the range of lines to be DELETED. If this is not desirable, then you must ensure that these lines are executed AFTER the DELETE operation. If, for example, a DEF FN statement is AFTER the delete block, the function will not correctly execute following the delete operation, unless the DEF FN statement is re-executed. Furthermore, if the statement which is to be the target of an ON ERROR GOTO statement will be moved upward by the DELETE operation, the ON ERROR GOTO statement must be

executed (or re-executed) AFTER the DELETE operation.

SYSTEM "SWAP"

Exchange the values of two named variables. Either or both of the variables may be elements of arrays.

SYNTAX:
    SYSTEM "SWAP",variable1,variable2
RETURNS:
    nothing. This is a VERB, not a function.

EXAMPLE:

    SYSTEM "SWAP",F1#,F2#

The contents of F2# are placed into F1#, and the contents of F1# are placed into F2#.

POSSIBLE ERRORS:
Type mismatch - The specified variables are not of the same type.
Illegal function call - The second variable is a non-array variable which has not been assigned a value.

FN ROW

Return a number from Ø to 15 indicating the current vertical position of the cursor on the display.

SYNTAX:
    ROW
RETURNS:
    number.

FN PW

Calculate an encoded password, or extract the encoded master password from any mounted disk.

SYNTAX:
    PW (expression)
RETURNS:
    number.
If expression is numeric, it must be in the range Ø-7, and specifies a drive number from which the encode of the master password will be retrieved. If expression is a string, the function will return the password encode value calculated from the first 8 bytes of the string value.

POSSIBLE ERRORS:
Illegal function call - The argument is numeric, but not in the range Ø-7.
Internal error - Error in attempting to read the disk master password.

FN HEX$

Convert a numeric expression to its hexadecimal equivalent.

SYNTAX:
    HEX$ (nmexp)
RETURNS:
    string.
'nmexp' is evaluated as an integer, and the character string representation of the hex equivalent is returned.

POSSIBLE ERRORS:
Type mismatch - The argument is not numeric.
Overflow - Argument could not be converted to integer type.

FN PK$

'Pack' a string into a compressed internal representation.

SYNTAX:

PK$(stexp)

RETURNS:

string.

The source string is compressed, using a 4 byte -> 3 byte algorithm. The source string must have a length less than or equal to 252 bytes. Characters in the source string must be in the range 2ØH to 5FH, which precludes the lower case alphabet. For each group of 4 characters in the source string, 3 characters will be generated in the result. If the length of the source string is not an even multiple of four, it will be effectively be 'padded' with blanks to even it out. This function will be useful in building large disk files, when disk space is at a premium .

EXAMPLE:

```
B$ = FN PK$("ABCDEFGH")
PRINT LEN(B$)
```

Will cause an encoded form of A$ to be stored in B$, with a length of 6.

POSSIBLE ERRORS:

Type mismatch - Argument is not a string.

Illegal function call - 1) Source string contains illegal characters, or 2) Length of source string is Ø or greater than 252.

FN UPK$

Decode strings compressed by the PK$ function.

SYNTAX:

UPK$(stexp)

RETURNS:

string.

The compression logic of the PK$ function is reversed, and the string is returned to ASCII format.

EXAMPLE:

```
PRINT FN UPK$(B$)
```

If used with the B$ created in the example for PK$, will print 'ABCDEFGH'.

POSSIBLE ERRORS:

Illegal function call - Length of source string not evenly divisible by three, or not in the range 3-189.

Type mismatch - Argument not a string.

FN PDAT

'Pack' a date into a two byte internal representation.

SYNTAX:

PDAT(stexp)

RETURNS:

number.

'stexp' is evaluated with the assumption that it contains a date in the format MM/DD/YY, as returned by the TIME$ function. If the string is longer than eight bytes, only the first eight will be used. The date is converted into an integer representing 'relative day of century', with a date of Ø1/Ø1/5Ø returning a value of zero. Dates prior to the mid-century date will return a negative value, while dates subsequent to the mid-century date will return a positive value.

EXAMPLE:

```
B% = FN PDAT ("Ø9/13/81")
```

Will return the value 11578.

POSSIBLE ERRORS:

Type mismatch - Argument is not a string.
Illegal function call - Invalid date string.

APPLICATION NOTES:

This function will be useful in two ways: 1) By converting dates into integer format, they may be stored more efficiently in files and arrays when memory usage or disk storage space may be of critical concern. 2) By representing dates as numbers, arithmetic may be performed thereon, facilitating computations in business applications (e.g. NET 3Ø).

FN UDAT$

Decode dates compressed by the PDAT function.

SYNTAX:

UDAT$(nmexp)

RETURNS:

string.

The compression logic of the PDAT function is reversed, and the number is converted to an ASCII format date.

EXAMPLE:

```
PRINT FN UDAT$(B%)
```

If used with the B% created in the example for PDAT, will print '09/13/81'.

POSSIBLE ERRORS:

Overflow - Argument could not be converted to integer.

Illegal function call - Numeric value was outside the range which could be converted to a valid date.

SNAPP FOUR - EXTENDED BASIC MAPPING SUPPORT (XBMS)
Automated Video Display Management for the Model I/III BASIC Interpreter

GENERAL INFORMATION:

This product is designed to automate for the BASIC programmer the tasks of presenting information on the video display and accepting information from the keyboard operator. Programs which use this facility will normally use much less memory, will be coded and debugged much more quickly, and will execute more quickly when doing screen and keyboard I/O. The facility, when installed, becomes part of your BASIC interpreter.

The system consists of two major components: 1) The OFF-LINE COMPONENT, which is a conversational BASIC program named GENERATE/BAS. You will utilize this component to describe to the system the screen formats you wish to use. 2) The ON-LINE COMPONENT, which becomes a part of your BASIC interpreter at installation time. You will issue commands to the ON-LINE COMPONENT from within your BASIC program to variously initialize a screen, send data to the video display, and receive data from the keyboard operator.

The interface between your BASIC program and the ON-LINE COMPONENT is via a verb. We have chosen the SYSTEM verb this purpose. The general format of a SYSTEM-COMMAND is:

```
SYSTEM command [argument [,argument]...]
```

where command is a string expression defined by the particular SYSTEM-COMMAND.

SCREENS, FIELDS, and ATTRIBUTES:

A SCREEN is the image of how you wish information displayed on the video to appear. It consists of one or more FIELDS, which are the elemental data items being sent to the display and received from the keyboard. In this system, a screen may contain up to 99 fields. Each field has a number of ATTRIBUTES which tell the system precisely how to handle it.

The first attribute is POSITION, which describes where on the display this particular field is to appear. When you define the position attribute, you specify a ROW, which is the vertical position indicator (Ø-15), and a COLUMN, which is the horizontal position indicator (Ø-63).
Another attribute is CAPTION, which is like a 'tag' for a field. An example of a caption might be "LAST NAME", if your screen called for name & address data. A caption is a character string which will appear on the display field, and may range from Ø to 63 characters.
Another attribute is FIELD LENGTH, which defines the maximum number of characters to be displayed on the video or accepted from the keyboard. The data component of the field may range from 1 to 24Ø in length (numeric fields have smaller maximum field lengths, depending upon their data type).
The data component also possesses a POSITION attribute, which it will normally 'inherit' from the position of the caption, but this may be overridden.

A field may be PROTECTED, which specifies that it is to be displayed only, and that the keyboard operator may not change it. The most common use of the protection attribute is probably for error messages.
The data component of the field has a VARIABLE NAME associated with it. This defines for the system what variable to find in your program when displaying the screen, and what variable to update after the operator has overkeyed the field. The variable name may be a simple variable, such as NL$ or AD#, or it may be a subscripted variable, such as NL$(3) or AD#(5). It even may contain an expression as the subscript, such as NL$(I) or AD#(I+J-K). The variable name may not exceed 13 positions in length.
If the variable is a number (rather than a string), it may be POSITIVE ONLY, which will prevent the operator from entering a negative value into this data field.
Single and double precision variables also have the attribute NUMBER OF DECIMAL PLACES, which controls both the way information is displayed on the screen and what the operator is permitted to enter.

Our handling of numeric fields can relieve you of many burdensome chores in editing numeric data. If, for example, you have defined a seven position field, specified that it is to contain two decimal places (as in dollars and cents type information), and that it may contain negative numbers, then the system logically looks at the field as Snnn.nn, where 'S' is the spot for the minus sign, if used, 'n' is a logical digit spot, and '.' is the position of the express or implied decimal point. THE SYSTEM WILL ABSOLUTELY ASSURE THAT DATA ENTERED INTO THIS FIELD WILL MATCH THIS LOGICAL VIEWPOINT OF THE FIELD. In this example, you can be assured that you will always get back a number N, such that -999.99 <= N <= 999.99, and that it will never contain more than the specified two fractional digits.

THE OFF-LINE COMPONENT:

The off-line component is contained in a BASIC program named GENERATE/BAS. When you RUN this program, it will first prompt you for a SCREEN NAME. You may chose any name, up to eight characters, which is a valid FILE NAME. You may specify a drive number, separated by a colon, but you may not specify any extension, as this program will create one or two output files, and will assign his own extensions to keep them separate. If the SCREEN NAME you enter is a previously existing screen, it will be fetched from disk, and the image displayed for you. Otherwise, a blank screen with only our command prompt line will be displayed. The command prompt line shows the following options:
<N>ext - positions the cursor at the 'next' field on the screen for editing or insertion. This function is duplicated by the TAB key.
<P>revious - positions the cursor at the 'previous' field on the screen for editing or insertion. This function is duplicated by the ESC key.
<A>dd - is a request to construct a NEW FIELD which is to be positioned on the screen AFTER all existing fields. This is the only valid option for a brand new screen.
<I>nsert - is a request to construct a NEW FIELD which is to be positioned just before the field on which the cursor is now placed.
<E>dit - is a request to change some attributes of an existing field on which the cursor is now placed. If the only attributes you wish to change are those of POSITION, you may also use the four arrow keys for this purpose.

<D>elete - is a request to discard the field on which the cursor is now placed.
<S>ave - indicates that you are done constructing and/or editing this screen. You will be prompted for information as to whether or not you want a documentation file created, and whether or not you want a printed copy of the documentation.
The four arrow keys may be used to 'move' the field upon which the cursor is currently placed in the direction indicated by the arrow itself. If you wish to move a field more than one or two positions, you will probably choose to type in a number before pressing the arrow, which will cause the field to move the indicated number of spaces without the nuisance of refreshing the entire screen for each move.

When you specify any of the commands A, I, or E, you will be moved from command mode to field specification mode, where you specify or change the FIELD ATTRIBUTES. Each prompt will ask you for an attribute, and display what will result if you supply only <ENTER>. The first two prompts are for the POSITION ATTRIBUTE, and will call for ROW and COLUMN. Don't worry about being a little off from where you really want the field, you can easily change these later with the arrow keys. The next prompt will call for the CAPTION. The next few prompts relate to the DATA COMPONENT of the field. You will be asked for the length and whether or not you want to override the position inherited from that of the caption. The next prompt will query whether this is to be a PROTECTED data field. You then be asked for the VARIABLE NAME. The variable name must be EXPLICITLY typed. That is to say, it MUST contain a %, #, !, or $. This explicit typing requirement ONLY applies during the generation of a screen image. In your BASIC program which uses the screen, you may still use implicit typing if you wish to do so. If the variable is numeric, you will be asked if negative numbers are to be accepted at entry time. If the variable is single or double precision, you will be asked for the NUMBER OF DECIMAL PLACES. Upon answering each of these prompts, the screen will be re-displayed with the new or revised field, and you will be returned to command mode.

In the event that you exceed the screen size limit, you will receive a message to that effect, and you will need to either reduce the number of fields or shorten a variable name. (see SYSTEM RESTRICTIONS)

When you are finished working on the screen, and you issue the Save command, a file will be created containing information to be used by the ON-LINE COMPONENT. This will be named "screenname/MAP", and will contain all information about the attributes of the fields. If you have been updating a previously existing screen definition, the previous file will be destroyed. You might be well advised to make a copy of the /MAP file if you anticipate major changes thereto.

When you request a <S>ave of your screen definition, you will first be asked if you want the fields sorted. In order for the TAB, BACK-TAB, EXPRESS TAB, AND EXPRESS BACK-TAB functions to work as described, the fields must be sorted. If, however, you have a special requirement (e.g. you want to arrange the screen into logical vertical columns, with data entry proceeding down the screen rather than left to right), you may specify that the fields NOT be sorted. Most applications will work best with a sorted screen, but you are

welcome to experiment with this. After the sort question, you will be asked if you want a documentation file to be created, and if you want a hard copy of the documentation. This documentation will save you a lot of time when creating your application.

If you request that a documentation file be created, it will be named "screenname/REM", and will consist of REM statements, starting with line number 64ØØØ, which you may merge into your program to give you reminders about the screen while you are writing your application. If you request a printout of the documentation , you will get a printed copy of the same information as that in the "/REM" file.

Since GENERATE/BAS does extensive processing when LOADing and SAVEing your screen information, and is an interpreted BASIC program itself, it is rather slow. Most users will not find this very distressing, because this program is only used infrequently during application development, and the ON-LINE component, which the end-user sees, is very quick. In any event, we have written GENERATE/BAS so it may be compiled using Microsoft's BASIC Compiler (BASCOM). If you wish to compile GENERATE/BAS, the only change necessary is to remove the CLEAR statement, which is in the first line of the program.

THE ON-LINE COMPONENT:

This is the easy part. There are only three primary commands. These three are to initialize a screen, to send a screen to the display with your data, and to receive the updated information. The remaining commands allow you to selectively override the protected / unprotected specifications you made at GENERATE time. You relax, and we'll do all the hard work. The logic structure of a simple application should be the model for your own, more complex applications. A very simple model looks like:

```
1Ø  INITIALIZE SCREEN
2Ø  SEND SCREEN
3Ø  RECEIVE SCREEN
4Ø  IF ERRORS, PRINT INDICATIVE MESSAGE
5Ø  IF MORE TO DO THEN GOTO 2Ø
```

The syntax of the initialization command is:

```
SYSTEM "INIT",string-expression [,numeric-expression]
```

where string-expression may be a literal, a string variable, or anything that will resolve to the SCREEN NAME you assigned when you GENERATED this screen. Please note that while most applications might use only one screen, that you are not limited in any way as to the number of different screens you wish to use. Initialization, however, takes a few (not more than 3.5) seconds, and must be done each time you wish to use a DIFFERENT screen. A well designed application, therefore, will choose to limit the number of times the screen CHANGES to the least required to accomplish the desired function. During the initialization, the CAPTIONS from the '/MAP' file will appear on the screen. The data areas will not appear until the first SEND, which will normally follow immediately.

Numeric-expression, if present, specifies the start of an 896 byte (&H38Ø) memory block in which to load the '/MAP' file. This memory block must have been protected upon entry to BASIC, or subsequently via the expanded CLEAR statement from our BUILTIN FUNCTIONS product. The start of this block must be > BASIC's HIMEM, and the end of this block must be <= LDOS HIGH$. If numeric-expression is omitted, the memory block start address will default to BASIC's HIMEM plus one.

The easiest way to position this block, given the variable nature of LDOS HIGH$, is to place the following statements at the start of your program:

```
SYSTEM CLEAR ,,Ø 'Move BASIC's HIMEM up to the limit.
SYSTEM CLEAR ,,&H38Ø 'Make room for the /MAP file.
SYSTEM "INIT","screen" 'Intialize XBMS screen at the current HIMEM+1.
```

If you will be using CEGC concurrently with BASIC mapping support, the following statements will do the job:

```
SYSTEM "CEGC",Ø 'Turn off the Garbage Collector, if he was on.
SYSTEM CLEAR ,,Ø 'Move BASIC's HIMEM up to the limit.
SYSTEM CLEAR ,,&H24Ø 'Make room for CEGC.
SYSTEM "CEGC" 'Enable CEGC at the current BASIC HIMEM+1.
SYSTEM CLEAR ,,&H38Ø 'Make room for the /MAP file.
SYSTEM "INIT","screen" 'Intialize XBMS screen at the current HIMEM+1.
```

POSSIBLE ERRORS:
Type mismatch - The argument is not a string expression.
Syntax Error - Either SNAPP-IV is not installed, or you completely omitted the argument, which is required.
Out of memory - The named screen is too large. You must have ignored an error message when you GENERATED this screen. Go back to GENERATE and try again.
File not found - The named screen was not found on any resident disk. Either you don't have the right disks mounted, or you spelled the SCREEN NAME incorrectly.
Internal error - The /MAP file is not in the correct format.
Illegal function call - The memory block specified or defaulted is not properly protected, or overlaps the range currently in use by the College Educated Garbage Collector.

The syntax of the send command is:

```
SYSTEM "SEND"
```

Note that there are no arguments. The system 'remembers' what screen you are currently using from the preceding initialize command. The system will, in response to this command, search your program for the variables named in the generation of the screen, and if they contain values, format them into the data areas of the screen. What do you do? 1) Remember to DIM any arrays which are needed by this screen. 2) Relax and let us do the driving!

POSSIBLE ERRORS:
Internal error - The screen table stored in high memory has been corrupted. Either you have loaded something on top of it (we warned you not to do that), or your memory is defective.

The syntax of the receive command is:

SYSTEM "RECEIVE" [,numeric-expression]

where numeric-expression, if present, may be an integer constant, a variable, or anything which will resolve to a FIELD NUMBER, which specifies the field in which you wish the cursor placed for the operator to commence entering information into the screen. If you omit numeric-expression, or specify an invalid field number, the cursor will be placed in the first unprotected field on the screen, otherwise it will be placed in the field you specify. The system will, in response to this command, TAKE CONTROL of the Model III, and work with the operator until the operator presses ENTER, ctl<E>, or ctl<Q>. At that time, control will return to your BASIC program, anything the operator keyed on the screen will be assigned back to the corresponding variables in your program, and a newly reserved variable ZZ$ will contain the value of the key which caused a return. That is to say, ASC(ZZ$) will always be 5, 13, or 17. In this way you can issue special instructions to the operator like "Press ctl<E> to EXIT, ctl<Q> to QUIT, or ENTER to update this record", and you will be able to determine what the operator wanted. In addition, another reserved variable, ZZ%, will contain the FIELD NUMBER of the field in which the cursor was positioned at the time that the operator returned control to your program.

POSSIBLE ERRORS:

Out of string space - We told you we were going to update your variables, including the string variables. Don't be so chintzy on string space. By the way, we won't fragment string space like a BASIC routine would, so you won't see the 'garbage collection' pauses nearly so much, if at all, with this product.

Internal error - as for receive.

Illegal function call - numeric expression greater than 255.

The final two commands allow you to override the protected / unprotected attributes assigned to fields at GENERATE time. This can give you additional flexibility where, for example, the great majority of the time a field is not to be updated, but occasional special requirements dictate that it may require modification.

The syntax of the command which allows you to force a field to be protected is:

SYSTEM "PROT",nmexpl [,nmexp2...]

where nmexpl, etc. specify field numbers. Invalid field numbers are ignored, but a field number expression outside the range Ø-255 will produce an Illegal function call.

The syntax of the command which allows you to force a field to be unprotected is:

SYSTEM "UPROT",nmexpl [,nmexp2...]

with the same format as the PROT command.

Note that the effects of these commands will be delayed until the next RECEIVE command is issued.

NOTES:

The SEND command will turn off the cursor, to avoid its annoying flicker while placing your data on the screen. The RECEIVE command will also turn the cursor off, turning if on only when actually soliciting operator input, and immediately after obtaining the desired keystroke, turn it right back off. All of this produces a more attractive screen for the operator. If you are using SNAPP-IV for ALL video/keyboard I/O, this will never be a problem. If however, you wish to do some of you own I/O, and wish the cursor to be displayed, you will have to turn it back on yourself. Just PRINT CHR$(14).

SYSTEM RESTRICTIONS:

This system requires that you reserve 896 bytes of memory to accommodate the /MAP information during execution. While the default location for this information is immediately above BASIC's HIMEM, you may override this in the INIT command to prevent conflicts with machine language routines from SNAPP or other software vendors.

The formula for the memory space requirement for the screen information is as follows: 6 * the number of defined fields, plus the summation of the lengths of all the variable names, plus one. If this exceeds 896, you are in trouble. If the average length of your variable names is three or less, you will be able to use the system defined maximum of 99 fields, and never run out of space here. If you ever run into this problem, check to see if you have any blanks in subscript expressions in your variable names. That is the first thing to get rid of. Otherwise, you will have to re-design the screen for a few less fields. Remember that, if necessary, you can PRINT additional information on the screen providing that you do so AFTER the INIT statement.

TECHNICAL INFORMATION:

The information contained within this section is NOT necessary for use of this product. It is provided primarily for those programmers who might wish to understand the format of our screen files, perhaps with the idea of writing their own programs to manipulate them.

We treat the '/MAP' file as two logical files, one of which contains the captions, the other of which contains all other information relating to your screen definition. We make them one physical file to reduce OPEN/CLOSE overhead processing time. Each of the logical files contains 'variable length' logical records, one for each field on the screen, and is terminated by a HEX 'FF' as a file delimiter, with each logical file starting at a sector boundary.

Within the first section of the file, which contains the captions, a logical record appears as follows:

REC+Ø Total length of this logical record.
REC+1
   BIT 7 Reserved.
   BITS 6-Ø Row of caption.
REC+2 Column of caption.
REC+3 The caption itself. This element varies in size, and its size can be calculated as (REC+Ø)-3.

Within the second section of the file, which contains all remaining information about your screen definition, a logical record appears as follows:

REC+Ø Total length of this logical record.
REC+1 Row of data field.
REC+2 Column of data field.
REC+3 Data field length.
REC+4 Number of decimal places.
REC+5 A series of bits:
   BIT 7 Protected field if on.
   BIT 6 Negative numbers o.k. if on.
   BIT 5 Reserved.
   BIT 4 Not used.
   BIT 3 Double precision if on.
   BIT 2 Single precision if on.
   BIT 1 Integer if on.
   BIT Ø String if on.
REC+6 Variable name. This element varies in size, and its size can be calculated as (REC+Ø)-6.

OPERATOR'S GUIDE:

The system you will be using was designed using an automated screen management facility called EXTENDED BASIC MAPPING SUPPORT. As all of the programs in this system will be using this facility, there are many keyboard/screen features and options which will be identical throughout the system.

When the application program wants to display some information to you, it will appear as a series of FIELDS on the screen. Most fields will have a CAPTION AREA and a DATA AREA. The CAPTION AREA is just a 'tag' to describe the purpose of this field. The DATA AREA is usually a place where you may fill in new information or change existing information. If the program is sending data to you in a field, it will be displayed right justified in the DATA AREA if it is numeric information, or left justified if it is character information. If the program does not send data to you for a particular field, that DATA AREA will appear as a series of small blocks to show you that this is a 'fill in the blanks' place on the screen.

Normally, immediately after sending you a screen, with or without information filled in, the program will expect you to fill in one or more fields with new or updated information. When this occurs, the CURSOR, a blinking block, will appear in the first position of the DATA AREA chosen by the programmer. You may then begin entering information as required by the particular application.

A number of SPECIAL KEYS have been defined to make your job of filling in the information easier. In the following discussion, ctl<LETTER> refers to the combination Shift, Down-Arrow, and the indicated letter.

The Down-Arrow key, on the left side of the keyboard, functions as a TAB key, in that it moves the CURSOR to the first position of the field following the one in which it is currently sitting.
The Up-Arrow key, right above the Down-Arrow key, acts as a BACK-TAB, in that it moves the CURSOR to the first position of the field preceding the one in which it it currently sitting.
Both of these keys will 'wrap around' from the last field on the screen back to the first , or vice versa.
Ctl<F>is an EXPRESS TAB, in that it moves the cursor to the first field on a lower line on the screen.
Ctl<B>is similar, except it performs an EXPRESS BACK-TAB function.
Shift-Left-Arrow and Right-Arrow without shift keys move the cursor within a field to allow you to EDIT information already there. These two keys are not functional in numeric fields.
The Left-Arrow key, in the upper right of the keyboard, erases the last character you typed in, and is functional for all field types. This key is NOT functional when you are in the middle of a field, having positioned yourself there using the shift-left-arrow or right-arrow keys.
Ctl<R> ERASES the field in which the cursor is currently located.
Ctl<T> will function as an 'ERASE TO END OF FIELD KEY', which might save you time rather than typing blanks to the end of the field.
The ENTER key, the ctl<E> key, and the ctl<Q> key are used as signals to the system that you are ALL DONE WITH THIS SCREEN. Don't press any of these keys until you are completed with your entries. Your programmer will advise you

which of these to use for what function, but in most programs, you will normally press the enter key.

All remaining keys on the keyboard function as standard typewriter keys. Normally, pressing one of these keys will cause the corresponding character to be displayed at the current cursor location, and the cursor to advance to the next location. If you fill a field up, the cursor will automatically advance to the next field, but if you enter your information into a field and there are extra spaces left, you will need to hit the down-arrow key to advance to the next field.

NUMERIC FIELDS:
Some special rules and features apply to fields that the programmer has defined as NUMERIC ONLY. For example, if the programmer has specified that the number is to contain two digits after the decimal point, it will not be necessary for you to enter a decimal point, as it will be automatically inserted for you in the appropriate position. It will not, however be displayed immediately on the screen, but only when you 'exit' that field and move to the next field. You may insert your own decimal points, if you wish (although it is probably easier not to), but if you do, you will not be able to enter more digits after the decimal point than specified by the programmer when the programmer defined the field. If the programmer has specified that the field may contain negative numbers, you may enter them, but you must put the minus sign in first. Of course, you will not be able to enter any letters, etc. into a numeric field.

SNAPP FIVE - EXTENDED FILE MAPPING SUPPORT (XFMS)

Automated Disk Input/Output Management for the Model I/III BASIC Interpreter

GENERAL INFORMATION:

This product is designed to automate for the BASIC programmer the task of moving data elements to and from a direct file. In Microsoft BASIC, this is a clumsy chore, as the FIELDed variables may not be directly be referenced by user logic. Programs which use this facility will normally use less memory, will be coded and debugged more quickly, and will execute more quickly when doing disk input/output. The facility, when installed, becomes part of your BASIC interpreter.

The interface between your BASIC program and this product is via a verb. We have chosen the SYSTEM verb for this purpose. The general format for invocation of a SYSTEM COMMAND is:

```
SYSTEM command [argument[,argument]...]
```

where 'command', for the purpose of this product, will be either GET or PUT, and the arguments are defined in the separate discussions of the commands.

WHAT'S WRONG WITH THE WAY IT IS NOW:

Direct file processing in BASIC was an afterthought to what started life as a simplistic problem solving language. Unfortunately, the program level implementation survives what was initially a 'band-aid' solution to an immediate problem. In most high-level languages, a programmer may define a direct correlation between his program variables and a direct access file buffer, with any necessary conversions normally performed automatically by the high-level language. In this implementation of BASIC, however, the programmer is limited to an overlay definition of the file buffer (using the FIELD statement) which consists ONLY of string variables. This imposes upon the programmer the requirement of EXPLICITLY moving all his variables to and from the file buffer, and specifying the necessary conversions.

Furthermore, since the memory address of normal string variables is subject to dynamic alteration, and the FIELDed string variables are only loosely connected to the file buffer (no checking is done by the interpreter to insure that the program does not inadvertently move them away from the file buffer), the programmer must be very careful when updating the FIELDed variables not to do so in a way that can cause their location to be moved. Failure to follow this convention will result in particularly hard to shoot 'bugs', as subsequent references to the FIELDed variable will, in fact, reference a newly created variable in the normal string area, and not the file buffer.

An additional penalty paid by the BASIC programmer in the current implementation is that two variables must be normally used for each data element, one in the field buffer, and one for normal program use. This wastes a limited resource, and leads to additional confusion.

To further complicate programming direct files, string variables are padded with blanks to fill out the FIELD length, so that equal comparisons may not

be made after a GET unless special coding is provided to account for the trailing blanks.

WHAT WE ARE GOING TO DO ABOUT IT:

First, we are going to eliminate the FIELD statement, eliminating the duplicate variable set and the question of where in the world the FIELDed variable is.

Next, we are going to add a third argument to the GET and PUT verbs. This argument will be a string expression defining the program variables moving to and from the file. In processing the new format GET and PUT statements, all necessary conversions will be performed automatically, eliminating all the <LET> statements which would normally be executed before a PUT (including the LSETS and the MKI$, MKS$, and MKD$ function references), or after a GET (including the CVI, CVS, and CVD function references).

A fringe benefit of the above is that it is no longer necessary to be concerned about keeping the buffer numbers in the FIELD statement in sync with those used in the GET and PUT statements.

In order to solve the problem of handling the trailing blanks from FIELDed variables, we shall simply remove them during GET processing.

Finally, we are going to define a new data type, supplementing the existing types of string, integer, single, and double. We will call the new data type ONE BYTE INTEGER. This data type may be used to store positive integers in the range Ø-255, and only occupies one position in the dataset. You may already have been using ONE BYTE INTEGER, if you have ever written code like this:

```
FIELD n, . . . 1 AS ZZ$, . . .
GET n,n
ZZ% = ASC(ZZ$)

LSET ZZ$ = CHR$(ZZ%)
PUT n,n
```

In addition to eliminating the above mentioned function references, the ASC and CHR$ function references will be performed automatically for this new data type.

SAMPLE PROGRAM THE OLD WAY:

Let's imagine a very simple record layout. Please keep in mind that for more realistic applications, the 'confusion factor' using the old way may seem to grow exponentially, while the straightforward code in the XFMS example will remain essentially constant. Our imaginary record looks like:

PROGRAM # OF

DESCRIPTION	PROGRAM VARIABLE	# OF BYTES
Last name	NL$	18
First name	NF$	18
Address	AD$	18
City	CT$	14
State	ST$	2
Zip Code	ZP!	4
Status Code	ST%	1
(above is ONE BYTE INTEGER)		
# of orders this year	OD%	2
Credit Limit	CL#	8
Current balance due	CB#	8
Month to date sales	MS!	4
Year to date sales	YS#	8
Salesman number	SM%	2
Month to date commissions	MC!	4
Year to date commissions	YC!	4

Our first task is to construct a FIELD statement describing this data record. Remembering that all of the FIELDed variables must be strings, and may not be directly referenced by our application program, we will have to select some variables which will not be used elsewhere in the program (and make sure we remember that!). For this example, our FIELD statement might look like:

```
FIELD 3, 18 AS ZA$, 18 AS ZB$, 18 AS ZC$, 14 AS ZD$, 2 AS ZE$, 4 AS ZF$, 1 AS
ZG$, 2 AS ZH$, 8 AS ZI$, 8 AS ZJ$, 4 AS ZK$, 8 AS ZL$, 2 AS ZM$, 4 AS ZN$, 4
AS ZO$
```

Note that when using the ZA, ZB, etc. naming convention for our 'dummy' variables, we must be constantly alert not to 'collide' with program variables. In the above example, had we used one more variable, we might easily have made the mistake of naming it ZP which, as you can see, is already in use in the program.

Now let's construct the code required to retrieve a record from the dataset and convert the FIELDed variables to the program variables (imagine that RN% has been initialized with the desired record number):

```
GET 3,RN%
NL$ = ZA$
NF$ = ZB$
AD$ = ZC$
CT$ = ZD$
ST$ = ZE$
ZP! = CVS(ZF$)
ST% = ASC(ZG$)
OD% = CVI(ZH$)
CL# = CVD(ZI$)
CB# = CVD(ZJ$)
MS! = CVS(ZK$)
```

```
YS# = CVD(ZL$)
SM% = CVI(ZM$)
MC! = CVS(ZN$)
YC! = CVS(ZO$)
```

And if that isn't bad enough, now look at the code required to replace a record on the dataset, updating the FIELDed variables with the new values as determined by the application program:

```
LSET ZA$ = NL$
LSET ZB$ = NF$
LSET ZC$ = AD$
LSET ZD$ = CT$
LSET ZE$ = ST$
LSET ZF$ = MKS$(ZP!)
LSET ZG$ = CHR$(ST%)
LSET ZH$ = MKI$(OD%)
LSET ZI$ = MKD$(CL#)
LSET ZJ$ = MKD$(CB#)
LSET ZK$ = MKS$(MS!)
LSET ZL$ = MKD$(YS#)
LSET ZM$ = MKI$(SM%)
LSET ZN$ = MKS$(MC!)
LSET ZO$ = MKS$(YC!)
PUT 3,RN%
```

At this point, we are reminded that direct file processing is usually the hardest topic for a new Microsoft BASIC programmer to master. No wonder!

SAMPLE PROGRAM THE NEW WAY:

First, we will construct a string variable describing the logical record and the variables we wish to associate with that record. If you refer to COMMAND SYNTAX, you will see that the syntax calls for a 'string expression', which means also that the descriptor could have been coded directly into the GET and PUT statements as a string constant, but making it a variable will usually simplify life, as the specification only need be made once. Note that the list of variables must be terminated with a colon.

```
FV$ = "(18)NL$, (18)NF$, (18)AD$, (14)CT$, (2)ST$, ZP!, (1)ST%, OD%, CL#,
CB#, MS!, YS#, SM%, MC!, YC! :"
```

Now the code to retrieve the record and make all the conversions:

```
SYSTEM GET FV$,3,RN%
```

Now the code to replace the record, updating all the data elements in the dataset:

```
SYSTEM PUT FV$,3,RN%
```

Hand from the back of the room:
"That seems too easy. What else to we have to do?"
Hmm.. RELAX, and leave the driving to us!

COMMAND SYNTAX:

SYSTEM GET
Retrieve specified record from specified file, updating specified program variables from the file buffer.

SYNTAX:

SYSTEM GET [*] stexp,nmexp1,nmexp2

The numeric expressions are identical to the standard BASIC GET verb. 'Stexp' is a string expression which, when evaluated, contains a series of 'Variable Identifiers', separated by commas. For numeric variables, the variable identifier simply consists of the variable name. For string variables, the variable identifier consists of this sequence: left paren, numeric expression, right paren, and variable name (again explicitly typed). Consistent with the philosophy of the BASIC interpreter, blanks are ignored throughout the entire expression. The normal operation of SYSTEM GET is to remove trailing space characters from string variables. The optional asterisk may be specified to override this action and leave the trailing spaces in place. Note that the list of variables must be terminated with a colon.

NOTES:

A) The one byte integer format is specified by use of the length specification of one preceding an integer typed variable.

B) The parenthesized length expression may be present without an associated variable, in which case it acts like a COBOL FILLER in skipping the specified number of bytes in the file buffer.

C) Explicit typing of the variables is not required in the string expression, although it might be a good idea to include them, as incorrect typing will produce difficult to troubleshoot errors.

D) Array members may be included in the list of variables. If you wish to include multiple, adjacent array members, you may do so by specifying the lowest member and the highest member, separated by a semicolon rather than a comma. As an example, "A%(1);A%(1ØØ)" specifies that one hundred array members are to participate in this I/O operation. If you use string arrays, specify the length attribute ONLY on the lowest member. The specified length will propagate through all the array elements.

E) To optimize the performance of the GET commands, we have chosen to 'leave' the string variables in the file buffer, rather than impose the overhead of moving each one to the string pool. For most applications, this will be transparent. If, however, your application issues consecutive GET commands with different format descriptors (this includes the situation where the subscript value of an array variable changes), and you need to preserve the values retrieved by preceding GET commands, you must explicitly move the string variables into the string pool. The easiest way to accomplish this is to concatenate a null string onto the string in question. E.G. A$ = A$ + "".

AN EXAMPLE:
is presented on the preceding page.

POSSIBLE ERRORS:

Exactly as for the normal GET command, plus
Type mismatch - first argument not a string expression.
Syntax error - required punctuation not found in the expected place.
Illegal function call - string expression has length of zero or invalid length specification for a string variable.

SYSTEM PUT
Place specified record in specified file (may be a replacement for an existing record, or a new record), using specified program variables.

SYNTAX, EXAMPLE, ERRORS:
As for SYSTEM GET except that the optional asterisk, if specified, will have no effect.

SNAPP SIX - THE COLLEGE EDUCATED GARBAGE COLLECTOR (CEGC)
A Performance Enhancement for the Model I/III BASIC Interpreter

GENERAL INFORMATION:

This product provides a substantial performance improvement to BASIC application programs which use strings extensively, by replacing a particularly time consuming section of code in the BASIC interpreter.

In implementing this BASIC interpreter, Microsoft chose to use a 'variable length string' approach, which has both advantages and disadvantages. The primary benefit of this approach is the simplicity from the programmer's standpoint of using strings. The primary drawback is that strings are constantly being relocated in a 'string pool' each time they gain a new value, and periodically the 'string pool' must be reorganized to condense the various small free areas into a single contiguous area.

This interpreter was originally conceived to be run on very limited-memory machines, before RAM and relatively high-speed disk storage became so cost effective. As a result of the space limitations on the target design machinery, the approach that the developers were forced to take in performing this string space reclamation function was rather crude and time consuming.

When this reclamation is called for, the system seems to 'lock up', and will not respond to the operator at all until the process is complete.

As available memory and peripheral storage devices have come to be relatively inexpensive for microcomputers, system designers have been calling upon these small machines to perform tasks that were never imagined by the early designers of this interpreter. We have found that the time required to perform string space reclamation (also known as 'garbage collection') is roughly proportional to the square of the number of active strings in the resident program. When dealing, for example, with a 4k RAM system, it would be unusual to have more than a handful of strings, and the time to collect the garbage would be only a minor nuisance. When you expand the system to the point where you may have, for example, 1ØØØ active strings, the garbage collection time becomes more than two and one half minutes!

We have developed a more modern solution to this problem which takes advantage of the fact that auxiliary memory is available, when needed, as a work area. Our enhanced garbage collection system requires 4 bytes per active string as a work area, and when this amount of space is available as 'free storage', will temporarily borrow that space, and return it to the free storage pool when completed. We will refer to this as the 'memory mode' of garbage collection. If the required storage is not available, our system will temporarily transfer out to disk enough of the BASIC interpreter and perhaps part or all of the resident BASIC program to make room for our work area, and return the 'paged out' information to its correct location when completed. We will refer to this as the 'disk mode' of garbage collection.

In almost every conceivable situation, our method produces significantly enhanced performance when compared to the Microsoft approach. The ONLY situa-

tion where this is not true is when A) The program uses only a very few strings, AND B) There is hardly any free memory at all. Under these circumstances, we will simply return control to the Microsoft code. In our own testing, we have never seen an application which did not benefit from the new method.

Actual benchmark times for the various methods follow:
These timings were made on a Model III. Model I times will be slightly greater. Times are in seconds.

#Strings	Microsoft	CECG/memory	CEGC/disk
125	2.7Ø	.35	1.1Ø
25Ø	1Ø.6	.8Ø	1.92
5ØØ	4Ø.8	1.9Ø	3.4Ø
1ØØØ	159	4.4Ø	7.2Ø
2ØØØ	631	1Ø.5Ø	14.2Ø
4ØØØ	2513	24.2Ø	29.7Ø

As you can see, in the last example, the enhanced routine is more than one hundred times as fast as the original approach. While the 1ØØ:1 ratio is based upon a 'laboratory' situation which will probably never occur in real world applications, it remains that many applications will gain significantly enhanced performance from the use of this intelligent processing function.

OPERATION:

Before using the enhanced collector, you must create a workfile for it to use in case insufficient memory is available for memory mode collection. The following BASIC program will create the necessary workfile:

```
1Ø OPEN "RN",1,"BASIC/SAV"
2Ø PUT 1,9Ø
3Ø CLOSE
4Ø END
```

After BASIC has initialized, but before running your application programs, issue the command SYSTEM "CEGC". This command may, if desired, be placed within your BASIC program, or entered from the keyboard. See the section 'Command Format' for details and options for this command. The improved collection routine will remain in control until BASIC is exited. If your systems make a habit of re-initializing BASIC, you will need to reload the collector to keep it going. If the reason you are re-initializing BASIC is to change the number of file buffers, please see our EXTENDED BUILTIN FUNCTIONS program product and the facilities included therein for changing the number of file buffers.

COMMAND FORMAT:

The complete syntax of the command to invoke the enhanced collection routine is as follows:

SYSTEM "CEGC" [,nmexp1 [,nmexp2]]

Nmexp1, if present, specifies the starting address of a 576 (&H24Ø) byte block in which the collection logic is to be loaded. If not specified, the address will default to BASIC's HIMEM plus one. The module start address must be > BASIC's HIMEM, and its end address must be <= LDOS HIGH$, or an Illegal function call error will be generated in response to the request. Additionally, the memory area must not overlap that currently in use by Extended Basic Mapping Support. A value of zero for nmexp1 is interpreted as a request to disable the enhanced collection routine. Nmexp2, if present, specifies the 'trigger level' at which point collection will be invoked, and will default to a value of 255.

APPLICATION NOTES:

The easiest way to position this block, given the variable nature of LDOS HIGH$, is to place the following statements at the start of your program:

```
SYSTEM "CEGC",Ø  'Turn off the Garbage Collector, if he was on.
SYSTEM CLEAR ,,Ø 'Move BASIC's HIMEM up to the limit.
SYSTEM CLEAR ,,&H24Ø 'Make room for CEGC.
SYSTEM "CEGC" 'Enable CEGC at the current BASIC HIMEM+1.
```

If you will be using CEGC concurrently with BASIC Mapping Support, follow the above statements with:

```
SYSTEM CLEAR ,,&H38Ø 'Make room for XBMS /MAP file.
SYSTEM "INIT","screen" 'Initialize XBMS screen at the current HIMEM+1.
```

LIMITATIONS:

As with all good things, there are some limits. For any given set of circumstances, there is a certain maximum number of strings which can be processed by this routine. We do not believe that any real world applications will come even close, but mention this for your information. The exact value of this maximum number is calculated by a fairly complex formula, but it will always be at least 128Ø, and will never exceed 576Ø. If the maximum is exceeded, we will simply turn control over to the Microsoft routine.

Because the Microsoft garbage collection logic is burned into a ROM, and cannot be patched, we are forced to perform collection on an 'anticipatory' basis. The enhanced collection routine will examine unused string space following the execution of each BASIC statement, and if the computed value falls below the 'trigger level', will commence collection immediately. We have chosen the default of 255 as the 'trigger level' value because ordinary assignment (LET) statements can never use more than that amount of string space. It is possible, however, for a single statement to use many times that amount. This is the reason that we support a user defined value for this level. If you perceive that your BASIC application is collecting without our assistance, you would be advised to increase the value from the default of 255. Be aware that UNNECESSARILY increasing this value, however, will cause collection to occur more frequently.

HINTS & TIPS:

As always, there are techniques which will reduce the frequency of garbage collection, and these techniques are still valid and wise to use: We have included in your documentation package an article entitled ". . . But the GARBAGE COLLECTOR will ring several times!" which will provide you with several suggestions in this area.

Using the Microsoft garbage collection routine, the best overall results were obtained by CLEARING as much string space as possible. Using the new one, this is not always true. IF, by reducing the CLEARED space, you can leave enough FREE MEMORY such that there are 4 bytes free for each active string, the benefit of memory mode garbage collection will accrue.

To assist you in optimizing the CLEAR value for any given BASIC program, we have included on your distribution disk a program named OPTIMIZE/BAS. Before you run this program you should execute the BASIC program which is of concern, and BREAK it at a point when activity is at its peak. OPTIMIZE/BAS will need four pieces of information: 1) The results from a PRINT MEM. 2) The results from a PRINT FRE(A$). 3) The number of bytes your program CLEARed. 4) The number of strings in use by your program.

In some applications, there are points where it is especially important not to permit garbage collection to occur. Previously, the way to avoid this was to force a garbage collection with a statement such as VV = FRE(A$). Note that even with CEGC installed, this statement will force a collection with the Microsoft logic. To force a collection via CEGC, use the statement SYSTEM FRE [VV%], where VV%, if present is any INTEGER variable. You might find it helpful to do this before any extensive operator input, so that the operator will not have to suffer a collection while in the middle of typing in a word.

POSSIBLE USER MODIFICATIONS:

This version of the collection routine will flash an asterisk in the upper right hand corner of the video display when collection starts, and blank it out when the procedure ends. If you do not care for this feature, the file named ASTEROFF/FIX contains a patch to disable it. The file named ASTERON/FIX contains a patch to re-enable this asterisk display. Both these /FIX files contain patch installation instructions.

SNAPP SEVEN - REVERSE COMPRESSION

GENERAL INFORMATION

The purpose of this enhancement to the LBASIC interpreter is to make BASIC programs as 'readable' as possible to facilitate understanding and/or maintenance.

It is a sad fact that the characteristics of interpretation dictate that the programs which are most easily read and understood by human beings are the least efficient from the standpoint of the machine, and vice versa. As a result, programs tend to become 'compressed' for increased execution efficiency. Several products, including one of ours, are on the market to perform this compression automatically.

A very tightly compressed program becomes extremely difficult to understand and modify, even if you wrote the program yourself. If someone else wrote the program and compressed it, the task becomes nearly insurmountable.

SNAPP-VII addresses this very real problem by offering a facility whereby a program may be 'cleaned up' such that reading and understanding is facilitated. Essentially, it does everything you could imagine to improve the readability of a BASIC program EXCEPT put the REMs back in (if someone can figure out how to do that, please call collect).

SNAPP-VII will:

Insert spaces into the program code wherever appropriate.
Separate multiple statement lines into separate lines where possible.
When a multiple statement line can not be separated, linefeeds and tabs will be inserted to improve readability.
Insert missing typing characters corresponding to the defaults in effect.
Indent statements contained within FOR / NEXT loops.

OPERATION

SNAPP-VII is invoked from the LBASIC Ready prompt by keying the letter 'I', possibly followed by optional parameters. The operation is performed upon the memory resident program.
Optional parameters are:
'M', which if present must be the first option, and requests suppression of the informational messages about the size of the program in lines and bytes before and after the operation.
'E', which requests the insertion of spaces following left parentheses and before right parentheses. Some people like spaces here, some hate them. No one seems to be indifferent on this topic. Try it both ways and pick your favorite.
'I', which suppresses the indentation of statements contained within FOR / NEXT loops.
'L', which requests indentation of lines which were split apart. This option will provide a 'stair-step' visual connection of statements which are executed as a single unit.

SOME REAL-WORLD SAMPLES

The following are intended more for illustrative purposes than as functional programs.

Print sorted mailing list:

A file named "TEST/DAT" is stored on disk. The file contains names and addresses. It is desired to print a list of the contents of the file in alphabetical order by last name. When the last names are identical, they should be listed in alphabetical order by first name within last name.

The record is arranged as follows:

Description	Size	Variable
Last name	18	NL$
First name	18	NF$
Street address	18	SA$
City	18	CT$
State	2	ST$
First five digits of ZIP	4	Z1!
Last four digits of ZIP	2	Z2%

The code might look like:

```
1ØØ  SYSTEM CLEAR -1,1ØØØØ 'Make sure we have a file buffer
11Ø  OPEN "RO",1,"TEST/DAT",8Ø 'Make sure the file is there
12Ø  DEFINT A-Z 'For convenience
13Ø  FD$ = "(18)NL$,(18)NF$,(18)SA$,(18)CT$,(2)ST$,Z1!,Z2%" 'Describe the
       record
14Ø  DIM NL$(LOF(1)),NF$(LOF(1)),RR%(LOF(1)) 'Set up arrays. They could have
       been named anything.
15Ø  FOR I = 1 TO LOF(1) 'The bounds of the file
16Ø   SYSTEM GET FD$,1,I 'Go getem
17Ø   NL$(I) = NL$ 'Save in the array
18Ø   NF$(I) = NF$ 'Ditto
19Ø   RR%(I) = I   'Remember where this person was in the file
2ØØ  NEXT 'I
21Ø  SYSTEM "SORT","+NL$,+NF$,RR%" 'Sort first name within last name, record
       numbers tag along
22Ø  SYSTEM "ERASE" NL$,NF$ 'We don't need these anymore. Getting rid of
       them will reduce garbage collection
23Ø  'At this point, the array RR% contains the list of disk record numbers
       in the desired alphabetical order.
24Ø  FOR I = 1 TO LOF(1)
25Ø   SYSTEM GET FD$,1,RR%(I) 'Note that we use the array as the index.
26Ø   ZP$ = FN FMT$("##### ####";Z1!,Z2%;) 'Working on the zip code
27Ø   MID$(ZP$,6,1) = "-" 'Post office likes this dash
28Ø   FOR J = 1 TO LEN(ZP$)
29Ø    IF MID$(ZP$,J,1) = " " THEN MID$(ZP$,J,1) = "Ø" 'Post office insists
         upon these zeros.
3ØØ   NEXT 'J
```

```
31Ø   LPRINT USING "%                % %                % %                %
        %                % %% %        %";NF$,NL$,SA$,CT$,ST$,ZP$
32Ø  NEXT 'I
33Ø  CLOSE
34Ø  END
```

Convert mailing list to upper/lower case:

The following is a program I wrote to convert my own customer database, stored on PROFILE II+, from all capitals to upper/lower case. It doesn't do the job perfectly, but it accurately fixed about 99%, and is a good example of the use of Extended File Mapping Support and FN LC$.

```
11Ø  SYSTEM CLEAR -3,1ØØØØ 'Acquire files & string space.
13Ø  DIM PR$(43)
14Ø  OPEN "R",1,"CUSTOMER/KEY" 'This is old-style blocked records.
15Ø  OPEN "R",2,"CUSTOMER/DAT",85 'This is blocked file mode.
16Ø  OPEN "R",3,"CUSTOMER/DA2",85 'As is this one.
17Ø  FK$ = "(OF%),(2Ø)NL$,(2Ø)CO$,(2)ST$,(1)PR$(1);PR$(43)"
18Ø  FD$ = "(16)NF$,(2Ø)AD$,(16)CT$,(14)CY$,(5)Z1$,(4)Z2$,(8)DT$,(2)RT$"
19Ø  F2$ = "(5)FP$,(2Ø)CC$,(5)ED$,(4Ø)RM$,(12)PH$"
2ØØ  FOR LI% = 1 TO LOF(3)
21Ø    OF% = 85*(LI%-(3*(INT((LI%-1)/3)))-1) 'calculate offset into record.
22Ø    SYSTEM GET FK$,1,(LI%+2)/3
23Ø    SYSTEM GET FD$,2,LI%
24Ø    SYSTEM GET F2$,3,LI%
25Ø    A$ = NL$ : GOSUB 38Ø : NL$ = A$ 'Convert last name
26Ø    A$ = CO$ : GOSUB 38Ø : CO$ = A$ 'And company name
27Ø    A$ = NF$ : GOSUB 38Ø : NF$ = A$ 'And first name
28Ø    A$ = AD$ : GOSUB 38Ø : AD$ = A$ 'And street address
29Ø    A$ = CT$ : GOSUB 38Ø : CT$ = A$ 'And city name
3ØØ    A$ = CY$ : GOSUB 38Ø : CY$ = A$ 'And country name
32Ø    SYSTEM PUT FK$,1,(LI%+2)/3
33Ø    SYSTEM PUT FD$,2,LI%
34Ø    SYSTEM PUT F2$,3,LI%
35Ø  NEXT
36Ø  CLOSE
37Ø  END
38Ø  IF LEN(A$) = Ø THEN RETURN 'Nothing to convert
39Ø  C% = -1 'If C% is true, capitalize next letter.
4ØØ  FOR I% = 1 TO LEN(A$) 'Step through the string.
41Ø    IF NOT C% THEN 43Ø 'This should not be capitalized.
42Ø    IF MID$(A$,I%,1) >= "A" AND MID$(A$,I%,1) <= "Z" THEN C%=Ø : GOTO 47Ø
       'Found a letter to capitalize : now stop capitalizing
43Ø    MID$(A$,I%,1) = FN LC$(MID$(A$,I%,1)) 'Bring this one down to L/C.
44Ø    IF MID$(A$,I%,1) = " " OR MID$(A$,I%,1) = "." OR MID$(A$,I%,1) = "-"
       THEN C% = -1 'End of word.  Start capitalizing again.
45Ø    IF I% > 1 THEN IF MID$(A$,I%-1,2) = "Mc" THEN C% = -1 'Special case
46Ø    IF I% > 2 THEN IF MID$(A$,I%-2,3) = "Mac" THEN C% = -1 'Ditto
47Ø  NEXT
48Ø  RETURN
```

A TUTORIAL ON GARBAGE COLLECTION

. . . But the Garbage Collector will ring several times!

By Bob Snapp

Most of you who have spent any time working with TRS-8Ø BASIC have observed the situation where the system will appear to 'freeze up' for some period of time, then (amazingly?) come back to life. You may have deduced that it has something to do with strings, or been told that it was 'reorganizing string space'. Want to know more about what's happening? Read on . . .

Variable length character strings are a real boon to the high-level language programmer. As a very simple example, consider printing an employee's name on a paycheck. If the employee's name, for example, were John Jones, you would not normally wish it to appear on the check as John Jones, but to have the last name appear immediately adjacent to the first name.. Using fixed-length strings, each of the name fields would normally be as long as the longest practical name, and would be filled out with blanks to this maximum length. The programmer in this case would have to tediously count out the number of trailing blanks in the first name field, and subtract that from the maximum length of the field to determine how to construct the print line for an attractive appearance. We should be grateful that Microsoft chose the variable length approach such that we can LPRINT NF$;" ";NL$.

If you still don't appreciate the value of this approach, ask a COBOL programmer what he would have to go through to get that paycheck the way we want it. He'd probably still be coding this time next week!

Given that the implementors of a language have chosen to use variable length strings, another question is immediately brought to mind: shall the storage for the strings be 'statically' or 'dynamically' allocated? Suppose, for example, that the application program needed to construct an array of names in which the longest name is 3Ø characters, but the average length is only 12. If storage were to be statically allocated, an array of 1ØØØ such names would immediately gobble up 3Ø,ØØØ bytes of memory. This is the way that North Star BASIC and IBM's VSBASIC handle strings. While memory has become relatively inexpensive these days, our current generation processors can still only address 65,536 bytes of memory, so the question normally is not one of 'Can I justify the cost of more memory?', but rather 'What would I do with more memory if I had it?'

Again, Microsoft has made the wiser choice, by dynamically allocating memory for strings on an 'as required' basis. In the example above, only 12,ØØØ bytes of memory would be required to implement the string array.

Experience should teach, however, that good things are not free, and this is no exception. In exchange for a very flexible and memory efficient string handling system, we pay the price of a rather complex overhead to 'manage' the strings.

Figure 1 is a 'map' of memory allocation for Microsoft BASIC, so you can get a feel for where the string space fits into the overall picture. Two of the areas on the map are completely dynamic: the stack, which moves downward, and the variable/array tables, which move upward. Stack requirements are normally minimal, but deeply nested GOSUB's or FOR/NEXT loops can make it quite large. When the stack is about to bump into the variable/array tables, or vice versa, the dreaded 'OM' error results.

This article, however, is about strings, so let us direct our attention to the string space. A 'pool' of storage for strings is set aside immediately below reserved memory (if any). The size of this area is determined by the CLEAR nnnn statement, or defaults to either 5Ø or 1ØØ bytes, depending upon the TRS-8Ø model.

Within the variable table, a string always occupies exactly three bytes of storage: one byte for the current length of the string (Ø-255), and two bytes for the location of the string data. The string data can be in one of four places: 1) The string was named in a FIELD statement, in which case the data is in the file buffer area; 2) The string was created by a LET or READ statement which assigned to it the value of a string literal, in which case the data is contained within the BASIC program; 3) The string has a length of zero, in which case there is no string data; or 4) Any other situation, in which case the string data is located somewhere in the string pool.

Space in the string pool is allocated to active strings starting from the top. Pointers are maintained by BASIC to 'next available string location' (initially right below reserved memory), and to 'top of stack'. When a string is to be stored in the pool, the number of bytes required is subtracted from 'next available', and the result is compared to 'top of stack'. When the comparison shows that space is available, the string is stored, 'next available' is updated, and processing continues on its merry way. When the test fails . . . but that's what we are here to learn about!

Let's examine here a very simple BASIC program and a diagram of what the string pool will look like in Figure 2.

```
1Ø CLEAR 12
2Ø A$ = STRING$(3," ")
3Ø B$ = STRING$(3," ")
4Ø C$ = A$
5Ø C$ = B$
6Ø A$ = C$
```

We will use the letter F to indicate a free position in the pool. Following the execution of line 1Ø, all 12 positions are free. When line 2Ø is executed, the top three positions are allocated to A$. In lines 3Ø and 4Ø, space is further allocated to B$ and C$. Note carefully the results, however, of line 5Ø. When C$ is assigned a second value, additional space was taken from the pool for the new value, and the old space occupied by C$ is ABANDONED! This space is marked with the letter G (for garbage).

The problem arises during the execution of line 6Ø. The interpreter determines that three more bytes of string storage are needed for the new value of A$, but no space is available.

S T O P E V E R Y T H I N G!

Alarms go off all over the place! After quickly consulting his 'emergency procedures' book, the interpreter calls the GARBAGE COLLECTOR: "Please rush right over, I gotta have at least three bytes for A$". The garbage collector arrives, and notices a 'hole' in the middle of the pool. He then pushes the value of C$ upward into the 'hole', leaving a new free area at the bottom of the pool. Satisfied that the job is done, the garbage collector departs, leaving his bill behind. In this case, however, his invoice is not for money, but for TIME.

In this simple example, the garbage collector acts quickly, and if you blinked, you would probably never notice that he had come and gone. In the 'real world', however, when many strings are active, his job can become rather formidable! In fact, he normally surveys the task before starting, and if it looks like a big job, he will think long and hard about doing that much work (he's naturally a little lazy).

He always comes through, but sometimes his bill is very large. I have transcribed his 'operations guide' into a PL/I-like pseudocode, in Listing 1. A careful examination will show that the amount of work he has to do grows EXPONENTIALLY with the number of strings. The main loop, "DO UNTIL HIGH-STRING-LOCATION = Ø", will be executed once for each string in the string pool, while the hottest subroutine, "EVALUATE-STRING-LOCATION", will be executed (for each iteration of the main loop) once for every string.

From a disassembly of the garbage collection routine, I have derived the following formula for calculating the time for the garbage collector to do his work:

```
CT = ( ( ( 149 * SV  (use 129 for Mod II)
         + 271 * AC  (use 251 for Mod II)
         +  92 * AE
         +  53 * NS
         + 219 * NN  (use 199 for Mod II)
         +  21 * SS
         + 258      ) * PS ) + ( 83 * BC ) ) / CS
```

Where: CT is collection time in seconds,
SV is the number of simple (non-array) variables, including string variables.
AC is the number of arrays,
AE is the number of string array elements,
NS is the number of null (zero length) string elements,
NN is the number of non-null string elements, including those located outside the string pool,
SS is the number of simple (non-array) string elements,
PS is the number of non-null string elements located in the string pool,

BC is the total number of bytes used for string data,
CS is the Z-8Ø clock speed, in cycles per second.
unmodified TRS-8Ø clock speeds are:
Model I - 1774Ø83
Model II - 4ØØØØØØ
Model III - 2Ø2752Ø

That's a pretty complex formula, and normally many of the factors are not terribly significant in determining the final result. A good approximation can usually be found from:

CT = 31Ø * PS^2 / CS (use 3ØØ for Model II)

The second formula was derived from experimentation, rather than calculations, and is in fact often more accurate than the first (proving only that my calculations were not perfect).

The most important thing to observe about the formulae is that the time required to garbage collect is roughly proportional to the SQUARE of the number of strings. This means, quite simply, that if you double the number of strings, you multiply the garbage collection time by 4!

Let's take a real world example: A Model III program with 5ØØ active strings. Using the second formula, we get:

31Ø * 5ØØ^2 / 2Ø2752Ø

or 38.224 seconds! During this time, the machine will seemingly 'lock up', and will not respond even to the BREAK key. And all this time you suspected your machine was malfunctioning?

To take the matter to its ridiculous extreme, the program shown below will produce a garbage collection time (again on a Model III) of 3 hours, 26 minutes, 24 seconds, again using the second formula.

```
1Ø CLEAR 91ØØ
2Ø DIM A$(9ØØØ)
3Ø FOR I% = 1 TO 9ØØØ
4Ø A$(I%) = CHR$(32)
5Ø NEXT
6Ø PRINT TIME$
7Ø PRINT FRE(A$)
8Ø PRINT TIME$
```

Now that we know just how bad that this can get, let's see what we can do to reduce its impact on us.

Our efforts in reducing the impact of garbage collection can be divided into two main groups: A) Reducing the frequency of garbage collection; and B) Reducing the time required for garbage collection when it occurs.

Reducing the frequency of garbage collection is normally the simpler of the two areas, and we shall start there.

Without changing the code in the program at all, simply increasing the value used in the CLEAR statement will cause garbage collection to occur less frequently, sometimes very dramatically. Garbage collection will occur in inverse proportion to the amount of unused string space (i.e. the results of FRE("")). If your program actually uses 1ØØØ bytes of string space and CLEARs, for example, 11ØØ bytes as a string pool, then garbage collection will occur at some general rate. Let us call this rate N. If, however, you were to CLEAR 6ØØØ bytes, the unused string space would be 5Ø times as large and garbage collection will occur at the rate N/5Ø. Your best choice, then, is to CLEAR the largest possible value.

Hand from the back of the room:

"How do we know what is the largest possible value?"

Good question. You have to determine this by trial and error. Just keep increasing the CLEARed value until you get OM errors, then reduce it until the OM errors go away. To be on the safe side, you might reduce it by a few hundred bytes more than seems absolutely necessary. By the way, programs tend to require changes, and you don't want to have to keep re-doing the trial and error, so I suggest that you use a 'reverse logic CLEAR'. The space required for variables and stack tends to change very slowly when modifications are made to the program, so use the following technique: 1) Determine through the trial and error process the largest practical amount to CLEAR. Call this value X. 2) Enter the following command: CLEAR Ø : PRINT MEM. Call this value Y. 3) The expression Y - X represents the space needed for variables and stack. Call this value Z. 4) Replace the CLEAR statement in the program with this: CLEAR Ø : CLEAR MEM - Z, plugging in the number derived above. This technique has been shown to reduce the need to go back and rework the CLEAR requirements.

Now that we know how much string space to CLEAR, let's learn how to treat our string space very gently. The first thing to understand is that each time a string variable appears on the left hand side of an assignment statement (statement which contains an equal sign), the old value of the string variable (if any) is abandoned UNLESS the string variable is used with LSET, RSET, or MID$. The easiest way to benefit from this knowledge is to prevent the abandonment of a string when its value, but not its length is to change. If, for example, A$ and B$ both have a length of 15, the execution of

```
A$ = B$
```

will cause the previous space occupied by A$ to be abandoned, contributing to the fragmentation which causes the calls to the garbage collector. The execution of

```
MID$(A$,1,15) = B$,
```

however, will NOT contribute to the fragmentation.

Multiple concatenation is another villain which tends to beat up the string space. If A$, B$, C$, D$, and E$ all have, for example, a length of 5, and we wish to construct Z$ with the five other strings 'strung together', the execution of

```
Z$ = A$ + B$ + C$ + D$ + E$
```

will really play havoc with the string space. BASIC will actually execute that statement as if you had entered

```
T1$ = A$ + B$
```

```
T2$ = T1$ + C$
T3$ = T2$ + D$
Z$  = T3$ + E$.
```

A much less damaging set of code would be

```
Z$ = STRING$(25,Ø)
MID$(Z$,1,5) = A$
MID$(Z$,6,5) = B$
MID$(Z$,11,5) = C$
MID$(Z$,16,5) = D$
MID$(Z$,21,5) = E$.
```

And the first statement in this sequence could be omitted if Z$ already had a length of 25.

A very typical set of code commonly found in an INKEY$ routine might look like:

```
1ØØ W$ = ""
11Ø I$ = INKEY$ : IF I$ = "" THEN 11Ø
12Ø IF ASC(I$) = 13 THEN RETURN
13Ø W$ = W$ + I$
14Ø IF LEN(W$) = N THEN RETURN ELSE 11Ø
```

A much less damaging set of code would be:

```
1ØØ W$ = STRING$(N," ")
11Ø I$ = INKEY$ : IF I$ = "" THEN 11Ø
12Ø IF ASC(I$) = 13 THEN 15Ø
13Ø K = K + 1 : MID$(W$,K,1) = I$
14Ø IF K = N THEN RETURN
15Ø FOR J = N TO 1 STEP -1
16Ø IF MID$(W$,J,1) <> " " THEN P = J : J = 1
17Ø NEXT
18Ø W$ = LEFT$(W$,P) : RETURN.
```

In the first set of code, a string will be abandoned once per character input. In the second example, strings will be abandoned only twice.

The final example of reducing the frequency of garbage collection deals with the exchange of the values of two strings. I won't go into great detail on this, because the need to exchange the values of two strings occurs primarily in sorting applications, and I feel that sorting should be done with a machine language routine. Suffice it to say that on the Model II, the SWAP verb can be used to exchange two values without leaving a trail of garbage behind. On the Model I or III, the following combination of VARPTR, PEEK, and POKE may be used to simulate the SWAP verb.

```
FOR I% = Ø TO 2
    T% = PEEK(VARPTR(A$)+I%)
    POKE (VARPTR(A$)+I%),PEEK(VARPTR(B$)+I%)
    POKE (VARPTR(B$)+I%),T%
NEXT
```

Now that we have learned some techniques that can reduce the frequency of garbage collection, how can we reduce its duration? Recall that the collection

time is roughly proportional to the square of the number of strings. While this means that doubling the number of strings will multiply the time by 4, it also provides a fertile ground for time savings, as a reduction of only 3Ø% in the number of strings will cut the collection time in half!

This, therefore, will be the area to attack. Reducing the number of strings is the only effective way to reduce the collection time.

A good example of string reduction might be a list of names and addresses. If a table of 1ØØ were required, you might be tempted to DIM, for example, separate arrays for last name, first name, address, city, state, and zip code. Resist the temptation at all costs! Using that technique would create 6ØØ strings, which definitely puts you in the RED ZONE. By merging each of the data items into a single string (with some sort of home-grown delimiter), you would cut the 6ØØ to 1ØØ, which is in the GREEN ZONE. Keep your eyes open, and any time you see more than one string array with the same dimension, it is probably a prime opportunity to cut down on the number of strings.

A more subtle example might be found in building an index to some kind of large data file. Suppose that the data file has 1ØØØ records, each uniquely identified by a 1Ø character string. If you wanted to be able to gain speedy access to each record by its identifier string, you might take the following approach. Pass through the file once, building parallel arrays of the identifier (string) and the record number (integer). Sort the arrays on the identifier, with the record number 'tagging-along'. When the user requests a record by identifier, binary search the string array, using the record number corresponding to the located string as the key for direct retrieval of the complete data record desired. A sketch of this approach will be found in Listing 2.

This routine will perform VERY quickly, as the binary search will go in almost no time, and only a single disk access is required for the fetch of the data record. Unfortunately, the routine is operating well into the RED ZONE for the garbage collector, and collection times in the 15Ø second range will be observed on a Mod III.

A slight modification to the technique will eliminate the garbage collection problem, with some additional overhead in disk accesses. After sorting the arrays, GET RID of the string array! On the Model II, use the ERASE command. On the Model I or III, pass through the string array and set all the strings to nulls. Then use the array of record numbers as a key to do the very same binary search ON DISK! Binary searching 1ØØØ records will never take more than 1Ø probes. Obviously, the individual searches will be slower, but because we have escaped the terrific garbage collection problem, they will be consistent. A sketch of this approach will be found in Listing 3.

You will find that a user will prefer, for example, a 4 second response to a request (all the time) to a 1 1/2 second response (most of the time) mixed with a 15Ø second response (occasionally). Even in the cases where the TOTAL time for a group of requests might be less, if the computer operator has to sit there and wait for a long garbage collection, he/she will become quite impatient with your program.

To summarize this final suggestion: In many cases you will be better off to implement a large table of strings on disk, rather than put up with long collection times.

LISTING 1

```
GARBAGE-COLLECTOR: PROCEDURE;
  IF PACKED-INDICATOR = TRUE THEN SIGNAL ERROR (OUT-OF-STRING-SPACE);
  PACKED-INDICATOR = TRUE; /* caller sets back to false */
  NEXT-AVAILABLE-STRING-LOCATION = TOP-OF-STRING-SPACE;
  DO UNTIL (HIGH-STRING-LOCATION = Ø);
    HIGH-STRING-LOCATION = Ø;
    POSTION = START-OF-WORKSPACE;
    DO WHILE (POSITION < END-OF-WORKSPACE);
      CALL EVALUATE-STRING-LOCATION;
      INCREMENT POSITION TO NEXT WORKSPACE ENTRY;
    END; /* do while (position < end-of-workspace) */
    POSITION = START-OF-VARIABLE-TABLE;
    DO WHILE (POSITION < END-OF-VARIABLE-TABLE);
      IF VARIABLE-TYPE = STRING THEN CALL EVALUATE-STRING-LOCATION;
      INCREMENT POSITION TO NEXT VARIABLE ENTRY;
    END; /* do while (position < end-of-variable-table) */
    POSITION = START-OF-ARRAY-TABLE;
    DO WHILE (POSITION < END-OF-ARRAY-TABLE);
      IF ARRAY-TYPE = STRING THEN DO;
        CALCULATE SIZE OF ARRAY AND POINT POSITION AT FIRST ELEMENT;
        DO WHILE (MORE ELEMENTS IN THIS ARRAY);
          CALL EVALUATE-STRING-LOCATION;
          INCREMENT POSITION TO NEXT ELEMENT;
        END; /* more elements in this array */
      END; /* array type = string */
      INCREMENT POSTION TO NEXT ARRAY;
    END; /* position < end-of-array-table */
    CALL PACK-SELECTED-STRING;
  END; /* until high-string-location = Ø */
  RETURN; /* that's all, folks! */
```

```
  EVALUATE-STRING-LOCATION: PROCEDURE;
    IF STRING-LENGTH = Ø THEN RETURN;
      /* don't bother with null strings */
    IF STRING-DATA-ADDRESS > NEXT-AVAILABLE-STRING-LOCATION THEN RETURN;
      /* above test indicates this string already packed */
    IF STRING-DATA-ADDRESS < BOTTOM-OF-STRING-SPACE THEN RETURN;
      /* above test indicates this string not in string space */
    IF STRING-DATA-ADDRESS < HIGH-STRING-LOCATION THEN RETURN;
      /* this means that this is not the highest string data */
    HIGH-STRING-LOCATION = STRING-DATA-ADDRESS;
    HIGH-STRING-DESCRIPTOR-ADDRESS = CURRENT-STRING-DESCRIPTOR-ADDRESS;
    RETURN;
  END; /* EVALUATE-STRING-LOCATION */
  PACK-SELECTED-STRING: PROCEDURE;
      /* references here relative to high-string-descriptor-address */
    MOVE STRING DATA TO (NEXT-AVAILABLE-STRING-LOCATION - STRING-LENGTH + 1);
    STRING-DATA-ADDRESS = (NEXT-AVAILABLE-STRING-LOCATION
      - STRING-LENGTH + 1);
    NEXT-AVAILABLE-STRING-LOCATION =
      NEXT-AVAILABLE-STRING-LOCATION - STRING-LENGTH;
    RETURN;
  END; /* PACK-SELECTED-STRING */
END; /* GARBAGE-COLLECTOR */
```

LISTING 2

```
1ØØ CLEAR 12ØØØ
11Ø OPEN "R",1,"DATAFILE"
12Ø FIELD 1, 1Ø AS FK$, 9Ø AS DT$
13Ø DIM RR%(LOF(1)), KT$(LOF(1))
14Ø FOR I% = 1 TO LOF(1)
15Ø     GET 1,I%
16Ø     KT$(I%) = FK$
17Ø     RR%(I%) = I%
19Ø NEXT
2ØØ 'INVOKE A MACHINE LANGUAGE SORT HERE.  SORT KT$ and RR%
21Ø 'USING KT$ AS KEY, RR% TAGGING ALONG.
22Ø LINE INPUT "NAME";SA$ : IF LEN(SA$) <> 1Ø THEN 22Ø
23Ø GOSUB 1ØØØØ
24Ø IF ER% THEN PRINT "NOT FOUND" ELSE PRINT DT$
25Ø GOTO 22Ø
1ØØØØ 'BINARY SEARCH OF STRING ARRAY,
        RECORD RETRIEVED IF FOUND,
        ER% SET IF NOT.
1ØØ1Ø ER% = Ø
        : ZB% = 1
        : ZD% = LOF(1) + 1
        : ZC% = (ZB% + ZD%) / 2
1ØØ2Ø IF SA$ = KT$(ZC%) THEN GET 1,RR%(ZC%)
                        : RETURN
1ØØ3Ø IF KT$(ZC%) > SA$ THEN ZD% = ZC%
                        : ZC% = (ZC% + ZB) / 2
                        : IF ZC% = ZD% THEN ER% = -1
                                : RETURN
                                ELSE 1ØØ2Ø
1ØØ4Ø ZB% = ZC%
        : ZC% = (ZC% + ZD%) / 2
        : IF ZC% = ZB% THEN ER% = -1
                                : RETURN
                        ELSE 1ØØ2Ø
```

LISTING 3

```
100 CLEAR 12000
110 OPEN "R",1,"DATAFILE"
120 FIELD 1, 10 AS FK$, 90 AS DT$
130 DIM RR%(LOF(1)), KT$(LOF(1))
140 FOR I% = 1 TO LOF(1)
150  GET 1,I%
160  KT$(I%) = FK$
170  RR%(I%) = I%
190 NEXT
200 'INVOKE A MACHINE LANGUAGE SORT HERE.  SORT KT$ and RR%
210 'USING KT$ AS KEY, RR% TAGGING ALONG.
211 FOR I% = 1 TO LOF(1)
212 KT$(I%) = ""
213 NEXT
220 LINE INPUT "NAME";SA$ : IF LEN(SA$) <> 10 THEN 220
230 GOSUB 10000
240 IF ER% THEN PRINT "NOT FOUND" ELSE PRINT DT$
250 GOTO 220
10000 'BINARY SEARCH OF DATAFILE,
RECORD RETRIEVED IF FOUND,
ER% SET IF NOT.
10010 ER% = 0
: ZB% = 1
: ZD% = LOF(1) + 1
: ZC% = (ZB% + ZD%) / 2
10015 GET 1,RR%(ZC%)
10020 IF SA$ = FK$ THEN RETURN
10030 IF FK$ > SA$ THEN ZD% = ZC%
: ZC% = (ZC% + ZB) / 2
: IF ZC% = ZD% THEN ER% = -1
: RETURN
ELSE 10015
10040 ZB% = ZC%
: ZC% = (ZC% + ZD%) / 2
: IF ZC% = ZB% THEN ER% = -1
: RETURN
ELSE 10015
```

FIGURE 1

Microsoft BASIC memory map

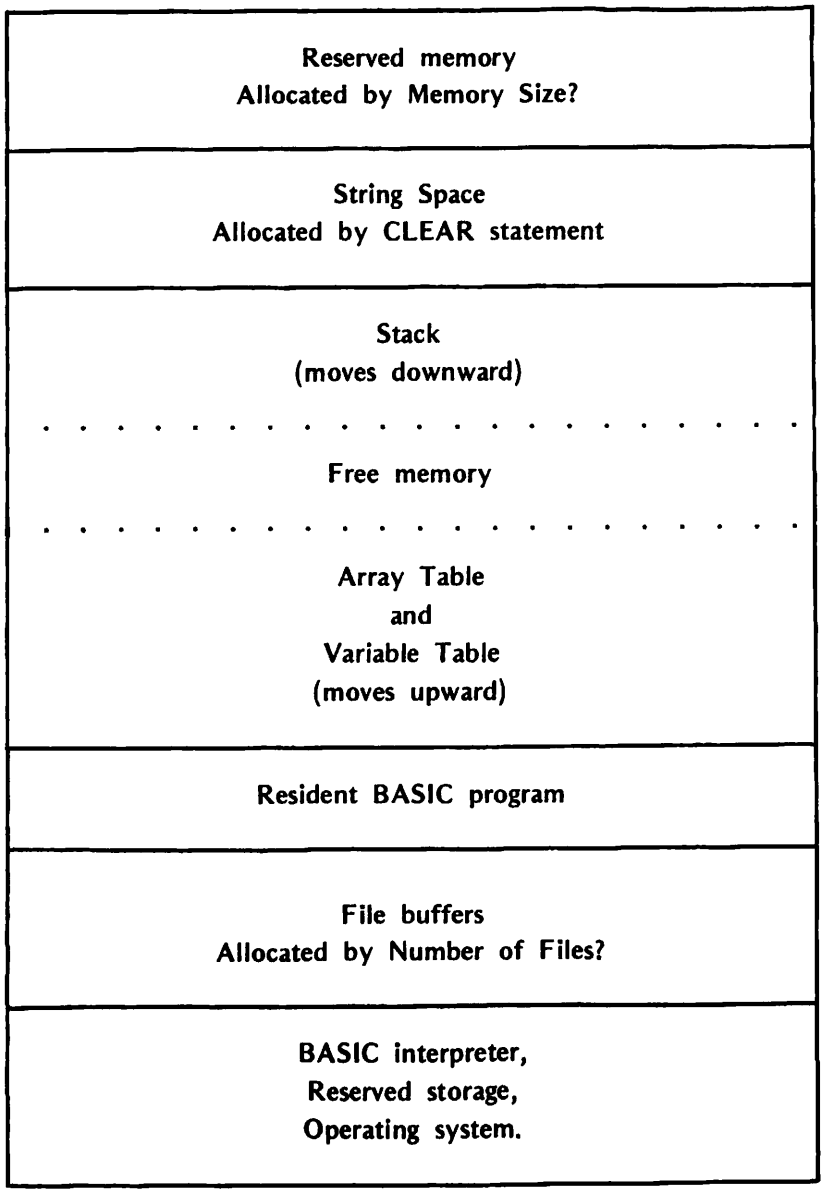

Figure 2

After 10	F	F	F	F	F	F	F	F	F	F	F	F
After 20	F	F	F	F	F	F	F	F	F	A$	A$	A$
After 30	F	F	F	F	F	F	B$	B$	B$	A$	A$	A$
After 40	F	F	F	C$	C$	C$	B$	B$	B$	A$	A$	A$
After 50	C$	C$	C$	G	G	G	B$	B$	B$	A$	A$	A$
During 60	F	F	F	C$	C$	C$	B$	B$	B$	A$	A$	A$
After 60	A$	A$	A$	C$	C$	C$	B$	B$	B$	G	G	G