S'MORE

BASIC

CARTRIDGE MEMORY EXPANDER FOR THE COMMODORE 64

cardco, inc.
300 S. Topeka • Wichita, KS 67201

S'more Basic - INSTRUCTION MANUAL

CARDCO, Inc. - (316) 267-6525

GUARANTEE

For as long as this product is owned by its original owner, CARDCO, Inc. will repair or replace any defective parts or the entire unit if it should become inoperative due to a defect in manufacture or materials, providing the unit is returned to CARDCO, Inc. in undamaged condition with proof of purchase (purchase receipt).

NOTICE: The Demo Diskette that is included with this cartridge is **NOT** covered by the above guarantee. See the appendix of this manual under the heading S'MORE DEMO DISKETTE for the diskette limited guarantee.

This product is distributed by:

CARDCO, Inc.
300 S. Topeka
Wichita, KS.
67202

S'more Basic - INSTRUCTION MANUAL

CARDCO, Inc. - (316) 267-6525

FIRST Printing - FIRST Revision

INSTALLATION OF THE S'MORE CARTRIDGE

When used properly your S'more Basic cartridge will provide a lifetime of satisfactory service. The following rules should be followed to assure trouble free operation:

1. Always turn your computer OFF when inserting or removing the S'more (or any other) cartridge.

2. Install the S'more cartridge with the S'more name plate facing up.

WARNING: Installing or removing the S'more cartridge upside down, or without turning your computer off can cause damage to both your S'more cartridge and your computer, and will invalidate your Guarantee.

HARDWARE COMPATIBILITY

Your S'more cartridge will work with both the Commodore 64 and the Commodore SX-64 computers, as well as with all other Commodore (or fully Commodore compatible) Disk Driver, Printers and Printer Interfaces. S'more may not be fully compatible with some other accessories like multi-slot expansion boards and screen expanders.

NOTE: If your S'more cartridge fails to power-up or operates erratically, please have the power supply of your computer tested before accusing S'more of not working. Commodore 64 computers have a history of inadequate and/or deteriorating (getting worse with heat and time) power supply capacity. The chances of power supply deterioration are many times greater than those of a defective S'more cartridge.

SOFTWARE COMPATIBILITY

S'more Basic is compatible with all Commodore 64 Basic commands. For more information on software compatibility refer to the following sections of this manual:

S'MORE BASIC VS. COMMODORE 64 BASIC
PEEKS AND POKES
MACHINE LANGUAGE WITH S'MORE BASIC
MEMORY MAP

S'more Basic - INSTRUCTION MANUAL

CARDCO, Inc. - (316) 267-6525

TABLE OF CONTENTS

INTRODUCTION

Thank you for purchasing S'more. We hope the cartridge and this manual will allow you to write better programs and make better use of your previously written programs. If you require additional help or feel you might have a problem with your S'more cartridge please call or write our customer service department between 9AM and 5PM (Central Time Zone) Monday through Friday. The phone number is at the top of every page of this manual and the address is:

CARDCO, Inc.
3ØØ S. Topeka
Wichita, KS.
672Ø2

About This Manual

Within this manual all of the new commands provided by S'more are divided into groups of similar commands. We recommend a complete reading and understanding of the first two sections of this manual (CONVENTIONS and S'MORE BASIC VS. CBM BASIC) before trying to go on to use your new S'more cartridge.

GENERAL CONVENTIONS

Please read these important notes on the standard conventions (notation system) used in this manual.

Within this manual you will encounter specific formats (called conventions) which are used consistently to describe certain situations. For example whenever you see <RETURN> that means you should press the return key. For example, if you saw a command listed as:

```
NEW <RETURN>
```

you would type the command NEW and then press the return key.

You will also see <SHIFT> and it tells you that you should hold down the shift key. The <SHIFT> is usually used in conjunction with another key like the return key or one of the function keys, for example:

```
<SHIFT/RETURN>
```

would mean you should hold down the shift key while pressing the return key.

```
<SHIFT/F7>
```

would mean you should hold down the shift key while pressing the F7 function key.

FORMAT CONVENTIONS

When each new KEYWORD is given you will be given the following information:

KEYWORD:

ALTERNATE:

FORMAT:

MODE:

This short capsule of information will give most programmers all the information they need to use each new command or function. The text following the heading will provide additional information about the KEYWORD. Finally you will be provided with examples of the use of each keyword so you can see how they are actually used.

KEYWORD - This is the fully spelled out form of the command or function. Some examples of standard Commodore Basic **KEYWORDs** would be RUN, LIST, PRINT and GOSUB.

ALTERNATE - This is the shortened form of the KEYWORD. Most KEYWORDs have shortened forms. For example the short forms of the Commodore Basic KEYWORDs listed above would be RUN - R<SHIFT/U>, LIST - L<SHIFT/I>, PRINT - ? and GOSUB - GO<SHIFT/S>.

FORMAT - This will be an example of the format that should be used for the KEYWORD being explained. Some statements require additional information (usually called parameters) for proper execution of the command. For example if a format given was:

FORMAT: NEW

no additional information needs to be supplied, so no extra parameter formats are given. But if the format were:

FORMAT: TAB(x)

the lower case x enclosed within parentheses indicates that additional information is required.

Different letter/number combinations will be used throughout this manual to signify specific parameter requirements. The table below lists the conventions used:

* Upper case letters must be entered exactly as shown in the FORMAT example.

* Commas, colons and parentheses must be entered exactly as shown in the FORMAT example.

* Items surrounded by square brackets [] are optional and need only be entered as desired. In some cases, if optional items are not entered, values are automatically assigned to them. These values are called the **default** values. Default values will be given where they apply for each KEYWORD.

* A lower case letter signifies a numeric value is required. The limits on the value will be specified following the FORMAT in parentheses. Unless otherwise stated, the value requirement can be fulfilled with either a number or a variable. For example:

```
A=1:TAB(A)
   or
 TAB(1)
```

* A lower case letter followed by a dollar sign (ie: a$, b$ etc.) will signify that a string of characters is required. If specific requirements are placed on the string required they will be defined following the FORMAT. Unless otherwise stated the requirement can be fulfilled with either a string of characters (which **must** be contained in quotation marks) or a string variable. For example:

```
PRINT "HELLO"
      or
A$="HELLO" : PRINT A$
```

MODE - As with the regular Commodore Basic commands and functions, S'more Basic commands and functions may be used in either the DIRECT mode or the PROGRAM mode, and some may be used in both. This line will let you know in which **MODE** each command may be used.

The **DIRECT** (or immediate) mode refers to executing (typing in) a command **DIRECT**ly from the keyboard as opposed to the **PROGRAM** mode, which would be using the command within a program. For example when you type **LOAD "PROGRAM"** you are telling your computer to LOAD a program now. A more complete explanation of the **DIRECT** and **PROGRAM** modes may be found in your Commodore instruction manual.

S'MORE BASIC VS. COMMODORE 64 BASIC

S'more Basic understands and works with all Commodore 64 Basic KEYWORDS. That means that basic programs written in normal Commodore 64 Basic will run without changing any of the basic program commands. **BUT**, (why is there always a but) in order to expand the memory to use the full 64K available, some of the memory locations had to be changed. What that will mean to you will depend on several things.

First, if your program uses PEEKs and POKEs, refer to the section called PEEKS AND POKES to see what you must change. (We have provided a neat little program on the disk that comes with the S'more cartridge (LOAD"PEEKER") that will identify potential PEEK and POKE problems, and even make some of the corrections for you.)

Second, if your program includes machine language sub-routines, refer to the memory map in the appendix of this book for proper locations and kernal calls.

Third, if your program uses the function keys, you will have to turn off the special function key definitions that are provided by S'more. (See: **KEY OFF**) Turning off the function key definitions can be done by placing the statement **KEY OFF** as one of the first statements in your program.

Last, to take advantage of the additional memory available when running under S'more Basic, you may wish to check for possible array variables that can be increased in size. The **FIND** function that is one of the new commands provided by S'more will help you expedite this process.

S'more Basic has some other nice features built into it that aren't even listed as new commands because they are completely invisible to you as a programmer. These handy features just do their job without any prompting required on your part.

*** UP/DOWN SCROLLING**

When listing a long program, stopping on the line you want to see can be a real pain with a normal Commodore 64. Well, with S'more Basic all you need to do is list any part of the program to the screen and then simply use the cursor up and cursor down keys to scroll up or down to see the rest of the program. That should cut substantially on your aggravation level.

* FAST GARBAGE COLLECTION

If you have ever had your computer seem to lock-up for several minutes while you were entering data or working on some other program that used a lot of string manipulation then you have experienced the frustration of the "GARBAGE COLLECTION" waiting game. Garbage collection is the term used to describe what happens when a computer runs out of temporary storage space and must clear out the garbage data to make room for new data.

The additional memory provided by S'more will cut down the chances of using all of the memory and incurring the wrath of the Garbage Collector. If you do happen to need all the memory S'more makes available, it will comfort you to know that S'more provides an Intelligent Garbage Collector who works hundreds of times faster (smarter?) than the Garbage Collector Commodore puts in the Commodore 64. Waits of up to 15 minutes are possible with Commodore's Garbage Collector, but you will never have to wait more than a couple of seconds for your Intelligent Garbage Collector.

* NULL STRING FIX

A very common complaint about Commodore Basic is that a null (empty) string received from any I/O device (disk drive, keyboard, etc.) does not equal CHR$(Ø). With S'more Basic any attempt to get the CHR$ value of a null string will return a value of CHR$(Ø).

Commodore Basic:

```
1Ø GET A$ : IF A$ = "" THEN A$ = CHR$(Ø)
2Ø A = ASC(A$)
```

OR

```
1Ø GET A$ : A$ = A$ + CHR$(Ø)
2Ø A = ASC(A$)
```

S'more Basic:

```
1Ø GET A$ : A = ASC(A$)
```

* COMMODORE BASIC COMMAND ENHANCEMENTS

Other Commodore Basic Commands have been enhanced. The enhancements have been done in such a way that the normal use of the command is the same as it was under Commodore Basic, so that existing programs will not need to be modified. Each of these enhanced commands is treated as a new command in this manual. (See: IF...THEN, INPUT, LIST, LOAD, PRINT, MID$, RESUME, RUN, SAVE, STOP, and VERIFY)

PART ONE

YOUR TOOL KIT

The first set of commands explained will be tools that you as a programmer can use to speed up your programming, program editing and de-bugging. These commands will be used most often in the DIRECT mode, but some may be useful in the PROGRAM mode as well. The area of programming aids was the area that we felt was most lacking in Commodore Basic. Using these commands will make programming a lot easier.

KEYWORD: **HELP**

ALTERNATE: **NONE**

FORMAT: HELP a$

(a$ must be either ON or OFF)
(NOTE: No quotation marks needed)

MODE: DIRECT

The HELP function is automatically turned on during power-up of S'more. When the HELP function is ON and an error is encountered during the running of a program, S'more will do the following:

1. Stop program execution.

2. Print to the screen the standard Commodore Basic error message.

   For example:
   SYNTAX ERROR LINE 125

3. Print to the screen the complete program line containing the error.

4. Place the cursor on the first character of the program statement that caused the error to occur.

When the HELP function is OFF and an error is encountered during the running of a program, S'more will react like Commodore Basic and print to the screen the standard Commodore Basic error message.

EXAMPLE # 1

```
                HELP OFF  <RETURN>

SYNTAX ERROR LINE 148
```

EXAMPLE # 2

```
                 HELP ON  <RETURN>

SYNTAX ERROR LINE 148

148 FOR I = Ø TO 65 : IBPUT A$(I) : NEXT
                      ^
                      Cursor Shows Here
```

KEYWORD: **' (Apostrophe)**

ALTERNATE: **NONE**

FORMAT: ' this is a remark:

MODE: DIRECT and PROGRAM

The apostrophe (also (') or the <SHIFT/7> key) serves the same function as the REM statement in basic with one minor exception, the apostrophe tells the program to skip to the next colon (as opposed to skip to the next line as caused by the REM statement) and continue program execution. This allows the programmer to put one or more program notes in the middle of a line at the exact program statement that the note refers to.

EXAMPLE # 1

```
10 A=Ø:'SET A TO ZERO:OPEN4,4,7:'SET PRINTER
   TO UPPER/LOWER CASE:GOTO5Ø
```

KEYWORD: **AUTO**

ALTERNATE: **A<SHIFT/U>**

FORMAT: AUTO [x]
(x is any number from Ø to 63998)
Default: x=Ø

MODE: DIRECT

To use the automatic line numbering function you must enable the function with the KEYWORD 'AUTO' followed by a number greater than Ø (ZERO). The number will be the increment between line numbers. For example to number in increments of 1Ø use the command:

```
AUTO 1Ø  <RETURN>
```

NOTE: The space between AUTO the increment number is optional.

NOTE: The space following every KEYWORD in this manual is optional and shown for clarity only.

Once the function is enabled whenever you enter a program line S'more will automatically print the next line number on the screen and place your cursor in the proper position to start typing in the next program statement. If you were using the 'AUTO 1Ø' command as listed above and entered a line numbered 112, S'more would print 122 as the next (112 + 1Ø) line number. If you are done entering program lines, use <SHIFT/RETURN> to put the cursor on an new un-numbered line. To turn off the AUTO function, type:

```
AUTO Ø  <RETURN>
```

EXAMPLE # 1

Start AUTO function with increment of 2Ø

```
AUTO 2Ø  <RETURN>
```

EXAMPLE # 2

To turn off the AUTO function

```
AUTO Ø  <RETURN>
      or
 AUTO  <RETURN>
```

NOTE: AUTO <RETURN> works here because the default value is Ø (ZERO).

KEYWORD: **LIST**

ALTERNATE: **L<SHIFT/I>**

FORMAT: LIST [a] [-b]

(a is the starting line number)
(b is the ending line number)
Default: a=Ø : b=63999

MODE: DIRECT and PROGRAM

The LIST command is an enhanced version of the Commodore Basic command. While you can use the LIST command in a program in standard Commodore Basic, when the command is executed it terminates the program. With S'more Basic you can, from within a program, list the program, or any part of it, and continue on with the program.

EXAMPLE # 1

```
LIST 1ØØ-15Ø  <RETURN>
```

EXAMPLE # 2

```
1ØØ GET A$ : IF A$ = "" THEN GOTO 1ØØ
11Ø IF A$ <> "L" THEN GOTO 13Ø
12Ø LIST 1ØØ-2ØØ : A$ = "R" : GOTO 11Ø
13Ø ' CONTINUE PROGRAM OPERATION
```

KEYWORD: **DELETE**

ALTERNATE: **DE<SHIFT/L>**

FORMAT: DELETE [a] [-b]

(a is the line number to delete or start deleting from)
(b is the highest line number to be deleted)
Default: a=Ø : b=63999

MODE: DIRECT

The format of the DELETE command is exactly like that of the LIST command. But, while the DELETE command allows you to easily delete a line or group of lines from your program, it may **NOT** be used in the PROGRAM mode.

EXAMPLE # 1

This command will delete line number 1Ø from your program if it exists.

```
DELETE 1Ø   <RETURN>
```

EXAMPLE # 2

This command will delete all program lines greater than 99 from your program.

```
DELETE 1ØØ-   <RETURN>
```

EXAMPLE # 3

This command will delete all program lines less then 4Ø1 from your program.

```
DELETE -4ØØ  <RETURN>
```

EXAMPLE # 4

This command will delete all program line numbers from 15Ø through 16Ø inclusive from your program.

```
DELETE 15Ø-16Ø  <RETURN>
```

EXAMPLE # 5

This command will delete all program line numbers from your program.

```
DELETE -  <RETURN>
```

NOTE: This way of deleting a program from memory is not the same as the NEW command. The OLD command will not find a program deleted in this manner.

KEYWORD: **FIND**

ALTERNATE: **F<SHIFT/I>**

FORMAT: FIND a$ [,[x]-[y]]

(a$ is any group of characters that may or may not be enclosed in quotation marks. The comma is not allowed as a character because it is used to separate the string to be found from the next parameter.
(x is the optional starting line)
(y is the optional ending line)
Defaults: x=Ø : y=63999

MODE: DIRECT

The FIND command allows you to quickly locate strings, variables, commands and functions in your program. For example if you wanted to find every line in your program that contained a print statement, you could easily do this with the FIND command. The FIND command will search through your program and list to the screen or to your printer every line that contains the item you want to find.

Be sure to be aware of these important rules:

RULE 1: All KEYWORDS may be used in their alternate form (ie: ? for PRINT or D<SHIFT/A> for DATA) with this command.

RULE 2: When searching for items contained within quotation marks, always use quotation marks around the item to find.

RULE 3: A search for a partial keyword will not find the keyword. (ie: FIND PRI <RETURN> will not find PRINT or PRINT#8 etc.)

RULE 4: When finding a variable like a$, S'more will also find arrays like a$(i) etc.

RULE 5: A search for the variable a will also find all variables beginning with a like a$, ab, a(i) etc.

RULE 6: When finding a text string it is sometimes wise to include spaces or punctuation to eliminate un-wanted matches.

```
FIND "AT"  <RETURN>
```

will also find "ATTENTION", "WHAT" and "MATTER", while

```
FIND " AT "  <RETURN>
```

wouldn't find those words by mistake, but would miss " AT." because of the trailing period.

EXAMPLE # 1

```
FIND A  <RETURN>
```

EXAMPLE # 2

```
FIND "At "  <RETURN>
```

EXAMPLE # 3

```
FIND OPEN,300-  <RETURN>
```

EXAMPLE # 4

```
FIND PRINT,140-240  <RETURN>
```

EXAMPLE # 5

```
FIND FORX=,-40  <RETURN>
```

EXAMPLE # 6

```
FIND A$(,-600  <RETURN>
```

KEYWORD: **CHANGE**

ALTERNATE: **CH<SHIFT/A>**

FORMAT: CHANGE a$,b$ [,[x]-[y]]

(a$ is any group of characters that may or may not be enclosed in quotation marks. The comma is not allowed as a character because it is used to separate the string to be replaced from the replacement string.)

(b$ is any group of characters that may or may not be enclosed in quotation marks. The comma is not allowed as a character because it is used to separate the replacement string from the line number limiters.)
(x is the optional starting line)
(y is the optional ending line)
Defaults: x=Ø : y=63999

MODE: DIRECT

If you think the CHANGE command looks like the FIND command with something added, you're right. The CHANGE command allows you to find and change things in your program without re-typing everything. For example if you wanted to change a file number from 8 to 9, you could easily do this with CHANGE.

EXAMPLE # 1

This command will cause PRINT#8 to be changed to PRINT#9 throughout the entire program.

```
CHANGE PRINT#8,PRINT#9  <RETURN>
```

EXAMPLE # 2

This command will cause "New" to be changed to "Old" throughout the program starting at line number 1ØØ (inclusive).

```
CHANGE "New","Old",1ØØ-  <RETURN>
```

EXAMPLE # 3

This command will cause "HOT" to be changed to "COLD" starting at line number Ø and stopping at line number 1ØØØ (inclusive).

```
CHANGE "HOT","COLD",-1ØØØ  <RETURN>
```

EXAMPLE # 4

This command will cause "A$(" to be changed to "AB$(" starting at line number 2ØØ and stopping at line number 3ØØ (inclusive).

```
CHANGE "A$(","AB$(",2ØØ-3ØØ  <RETURN>
```

NOTE: All the notes that apply to the FIND command also apply to the CHANGE command.

KEYWORD: **NUMBER**

ALTERNATE: **N<SHIFT/U>**

FORMAT: NUMBER [a] [,b] [,c]

(a is the number to be used as the lowest line number.
(b is the increment between line numbers.
(c is the program line number to start renumbering at.
Default: a=1Ø : b=1Ø : c=Ø)

MODE: DIRECT

The NUMBER command allows you to renumber your entire program or part of your program. Number can be used to give you more room between line numbers of a program as well as to make your programs look more professional.

The NUMBER command will automatically renumber all references (ie: GOTO's, GOSUB's, etc.) to specific lines. If a reference to a non-existent line is found, S'more will assign it a line number of 63999. So after you renumber any program you should use the FIND command to locate any occurrences of the number 63999 (FIND 63999 <RETURN>) so you can correct your mistake before it causes a program error.

EXAMPLE # 1

This will renumber all program lines. After this function is completed the first line number in your program will be 1Ø, the next 2Ø, the next 3Ø and so on...

```
NUMBER  <RETURN>
              or
NUMBER 1Ø  <RETURN>
              or
NUMBER ,1Ø  <RETURN>
              or
NUMBER 1Ø,1Ø  <RETURN>
              or
NUMBER ,,Ø  <RETURN>
              or
NUMBER 1Ø,,Ø  <RETURN>
              or
NUMBER ,1Ø,Ø  <RETURN>
              or
NUMBER 1Ø,1Ø,Ø  <RETURN>
```

NOTE: All of the above commands will have exactly the same net effect, because of the default values established by S'more.

EXAMPLE # 2

This will renumber all program lines. After this function is completed the first line number in your program will be 1ØØØ, the next 1Ø1Ø, the next 1Ø2Ø and so on...

```
NUMBER 1ØØØ  <RETURN>
            or
NUMBER 1ØØØ,1Ø  <RETURN>
            or
NUMBER 1ØØØ,1Ø,Ø  <RETURN>
```

EXAMPLE # 3

This will renumber **only the lines of your program that are greater than 1ØØØ,** starting at 1ØØØ in increments of 1ØØ.

```
NUMBER 1ØØØ,1ØØ,1ØØØ  <RETURN>
```

NOTE: After you have finished with a program you can compact it so the program takes up less memory by renumbering it starting at 1 with increments of 1. (NUMBER1,1 <RETURN>) Try this with a large program and use the FRE(Ø) function to see just how much memory line numbers can waste.

KEYWORD: **DUMP**

ALTERNATE: **D<SHIFT/U>**

FORMAT: DUMP

MODE: DIRECT

When the DUMP command is used S'more will list all of the non-array variables and their values that are active as of that point in your program. The DUMP command must be used before editing any program lines, because editing clears all variables.

DUMP is a very handy tool to help determine what went wrong in a program that uses multiple nested for/next loops, or other multiple variable problems. Dump will list variables to the screen or, as shown below, to the printer.

EXAMPLE # 1

To DUMP to the screen

```
DUMP  <RETURN>
```

EXAMPLE # 2

To DUMP to the printer

```
OPEN4,4 : CMD4 : DUMP  <RETURN>
```

KEYWORD: **TRACE**

ALTERNATE: **TRA<SHIFT/C>**

FORMAT: TRACE a$

(a$ must be either ON or OFF)
(NOTE: No quotation marks needed)

MODE: DIRECT and PROGRAM

The TRACE function allows you to track the execution of your program line by line. As each program statement is executed, the line number containing the program statement is printed on the screen. TRACE does slow down program execution so S'more allows you to use the trace command as a program statement so that you can turn TRACE on or off as desired.

EXAMPLE # 1

```
TRACE ON  <RETURN>
```

EXAMPLE # 2

```
1010 FOR I = 0 TO 999
1020 GOSUB 300 : IF X = 60 THEN TRACE ON
1030 NEXT : TRACE OFF
```

KEYWORD: **OLD**

ALTERNATE: **NONE**

FORMAT: OLD

MODE: DIRECT

Although you may not use this command very often, if you ever need it you will be very glad you've got it. The simplest explanation of what the OLD command does is to say that it is the opposite of the Commodore Basic NEW command. The OLD command will restore a program that has been erased from memory by the NEW command.

SPECIAL NOTE: The OLD command will also restore a program after a system reset. Some expansion devices provide a hardware system reset button, usually the OLD command will restore a program that has been interrupted by this type of reset.

SPECIAL SPECIAL NOTE: Occasionally a power surge or static discharge will cause your computer to lock-up or restart. OLD will sometimes restore a program after even this total devastation. (To clear a system lock-up you will need a hardware system reset as mentioned above.) Type: OLD <RETURN> as soon as you have a cursor, **before** trying to do anything else.

KEYWORD: **DEC**

ALTERNATE: **NONE**

FORMAT: DEC(a$)

(a$ is a hexadecimal number less than or equal to FFFF)

MODE: DIRECT and PROGRAM

This command will help you to easily convert hexadecimal to decimal numbers. The hexadecimal number must be a string variable or enclosed in quotation marks.

EXAMPLE # 1

```
PRINT DEC("FE44")  <RETURN>
```

EXAMPLE # 2

```
A$ = "E34D" : PRINT DEC(A$)  <RETURN>
```

EXAMPLE # 3

```
10 INPUT"ENTER A HEX VALUE";A$
20 IF LEN(A$) > 4 THEN GOTO 70
30 FOR I = 1 TO LEN(A$)
40 B$ = MID$(A$,I,1) : B = ASC(B$)
50 IF ((B>47) AND (B<58)) THEN GOTO 80
60 IF ((B>64) AND (B<71)) THEN GOTO 80
70 PRINT "BAD NUMBER" : GOTO 10
80 NEXT : PRINT DEC(A$) : GOTO 10
```

KEYWORD: **HEX$**

ALTERNATE: **NONE**

FORMAT: HEX$(a)

(a$ is a decimal number less than 65536)

MODE: DIRECT and PROGRAM

This command will help you to easily convert decimal numbers to their hexadecimal equivalent. The decimal number can be either a real number or a variable and it must be enclosed in parentheses.

EXAMPLE # 1

```
        PRINT HEX$(2744)  <RETURN>
```

EXAMPLE # 2

```
     A = 19 : PRINT HEX$(A)  <RETURN>
```

EXAMPLE # 3

```
10 INPUT"ENTER A VALUE";A
20 IF A>65535 THEN PRINT"BAD NUMBER":GOTO 10
30 PRINT HEX$(A) : GOTO 10
```

KEYWORD: **KEY**

ALTERNATE: **NONE**

FORMAT: (SEE TEXT)

MODE: DIRECT and PROGRAM

The KEY command is rather unusual in both its function and format. It actually has several uses, but all of the variations have something to do with the operation of the function keys. This section will explain each of the uses of the KEY command as if each was a separate command.

When you power-up your computer with S'more installed, each of the eight function keys is assigned a special function. The various formats of the KEY command allow you to turn-on, turn-off, examine and modify these special functions.

FORMAT # 1: KEY <RETURN>

Typing KEY and pressing the return key will list to the screen all of the special functions that are currently assigned to the function keys. Initially S'more assigns the function keys the following special functions, after power-up the KEY <RETURN> command will list them as follows:

```
KEY1,"KEY:" + CHR$(13)
KEY2,"RUN:" + CHR$(13)
KEY3,"PRINT DS$:" + CHR$(13)
KEY4,"DISK" + CHR$(34)
KEY5,"LIST:" + CHR$(13)
KEY6,"OLD:" + CHR$(13)
KEY7,"CATALOG DØ:" + CHR$(13)
KEY8,"AUTO 1Ø"
```

Each function key stores a string of characters and acts as an automatic typist, when you press function key F5 the typist will type the string of characters stored there which is L-I-S-T and <RETURN>. (NOTE: CHR$(13) is the same as the <RETURN> key.) So pressing the F5 function key will accomplish the same as typing the LIST <RETURN> command. The following is a list of what each function key will do when pressed:

F1 -	List the string values assigned to each function key.
F2 -	Run the current program in memory.
F3 -	Prints to the screen the current disk drive error channel message.
F4 -	prints DISK" to the screen.
F5 -	List the current program in memory.
F6 -	Performs the OLD function.
F7 -	Prints to the screen the disk directory of the disk in drive Ø of the current device number.
F8 -	Prints AUTO 1Ø to the screen.

FORMAT # 2: KEY OFF <RETURN>

This command turns the function key special functions off and assigns the function keys the values of CHR$(133-140) normally assigned under Commodore Basic.

FORMAT # 3: KEY ON <RETURN>

This command will turn the function key special functions back on after they have been turned off. The special functions will be the same as those that were active at the time the keys were turned off.

FORMAT # 4: KEY CLR <RETURN>

This command clears all of the special functions that were assigned to the function keys. After the KEY CLR command the function keys will have the value of a "null" or "empty" string.

FORMAT # 5: KEY NORM <RETURN>

This command will reset the function key special functions to their starting values. (IE: see the above list) The KEY NORM command overrides any other KEY commands or changes made to key functions.

FORMAT # 6: KEY a,b$ <RETURN>

(a is the key 1 thru 8)
(b$ is any string up to 128 characters long. See text)

This is the command format that allows you to re-define the special functions assigned to the function keys. The total amount of space reserved for function key definitions is 128 bytes. You can use all 128 bytes for the definition of one key or spread the space available among all eight keys. Using the alternate forms of KEYWORDs will allow you to pack more into your definitions. If you have a set of definitions that you find you are using a lot, you can create a basic program (like the one shown below), save it on disk and run it whenever you want to use your special key definitions.

```
Ø5 ' KEY DEFINITIONS
1Ø A$(1)="LIST"+CHR$(13)
2Ø A$(2)="SAVE"+CHR$(34)+"`Ø:WORK"+CHR$(13)
3Ø A$(3)="NUMBER"+CHR$(13)
4Ø A$(4)="NORM"+CHR$(13)
5Ø A$(5)="CATALOG DØ"+CHR$(13)
6Ø A$(6)="CATALOG D1"+CHR$(13)
7Ø A$(7)="RUN"+CHR$(13)
8Ø A$(8)=CHR$(13)+"DATA "
9Ø FOR I = 1 to 8 : KEYI,A$(i) : NEXT
99 NEW
```

You can even store a program and run it as a function key definition.

```
KEY CLR  <RETURN>

A$="Ø INPUT"+CHR$(34)+"ENTER FILE NAME"+
    CHR$(34)+";X$:RUNX$:"+CHR$(13)

A$=A$+"RUN:"+CHR$(13)

KEY8,A$  <RETURN>
```

When F8 is pressed, this little program will enter and run itself and ask you for the name of the next program you want to load from disk. It will then load and run the requested program. If you saved this as one of the KEY definitions in the previous program you would have a neat auto boot routine. How's that for power from one key stroke?

In the following examples are some other handy key definitions. Any time you find yourself with an item that you type in numerous times, you have a candidate for a function key definition. Remember that you can include any valid character in the key definition string, including RETURN (CHR$(13)), CURSOR DOWN (CHR$(17), DELETE (CHR$(2Ø)), CLEAR/HOME CHR$(147) and all the rest of the control characters.

EXAMPLE # 1

```
         KEY8,"PRINT#8,"    <RETURN>
```

EXAMPLE # 2

```
         KEY8,":FORI=ØTO"    <RETURN>
```

EXAMPLE # 3

```
  KEY8,"GETA$:IFA$=CHR$(Ø)GOTO"    <RETURN>
```

EXAMPLE # 4

```
        KEY8,"CARDCO,Inc."    <RETURN>
```

EXAMPLE # 5

```
          KEY8,CHR$(34)    <RETURN>
```

EXAMPLE # 6

```
1Ø KEY OFF
2Ø 'BODY OF YOUR PROGRAM
99 KEY ON : KEY NORM
```

PART TWO

ONE LINERS

As the name of this section suggests, the commands covered here are short but sweet. These commands were all missing from Commodore 64 Basic. In order to accomplish these simple tasks, Commodore required you to PEEK, POKE, SYS or do other unusual things that aren't required on other less advanced computers. Now, PEEK, POKE and SYS aren't all that bad, if you have a good memory for numbers and accurate typing fingers. But if you miss by even one number, you don't get a SYNTAX ERROR, you get a DISASTER. (NOTE: Also see PART 9 - PEEKS AND POKES for more information about PEEK and POKE commands and locations.)

These easy-to-learn and easy-to-remember commands should speed up your programming and prevent some **terminal** mistakes. Commodore is including forms of most of these functions in the C-128's new Basic 7.Ø.

KEYWORD: **CLS**

ALTERNATE: **NONE**

FORMAT: CLS

MODE: DIRECT and PROGRAM

The CLS command performs the simplest of all functions, it clears the screen of your video display.

NOTE: The CLS command also resets all of the 1ØØØ SCREEN COLOR RAM locations to their default value of Dark Grey (11).

EXAMPLE # 1

```
CLS <RETURN>
```

EXAMPLE # 2

```
1Ø DIM A$ (8Ø) : CLS : PRINT "HELLO"
```

KEYWORD: **UPPER**

ALTERNATE: **U<SHIFT/P>**

FORMAT: UPPER

MODE: DIRECT and PROGRAM

This command will not speed up your heart rate. The UPPER command shifts the screen to the UPPER CASE/GRAPHICS display mode. It works just like the <SHIFT/COMMODORE> key combination.

EXAMPLE # 1

```
UPPER <RETURN>
```

EXAMPLE # 2

```
10 CLS : UPPER : PRINT "HELLO"
```

KEYWORD: **LOWER**

ALTERNATE: **LO<SHIFT/W>**

FORMAT: LOWER

MODE: DIRECT and PROGRAM

LOWER is the opposite of the UPPER command. (Makes sense... ed.) The LOWER command will shift your screen display to the UPPER/LOWER CASE display mode. It works just like the <SHIFT/COMMODORE> key combination.

EXAMPLE # 1

```
            LOWER  <RETURN>
```

EXAMPLE # 2

```
1Ø CLS : LOWER : PRINT "HELLO"
```

KEYWORD: **REPEAT**

ALTERNATE: **RE<SHIFT/P>**

FORMAT: REPEAT a$

(a$ is either ON or OFF)
(NOTE: quote marks not needed)

MODE: DIRECT and PROGRAM

S'more powers-up with the REPEAT function OFF. Turning the REPEAT function ON allows all keys to repeat when held down for more than .6 seconds. Turning the REPEAT function OFF allows only the cursor keys, insert/delete and the space bar to repeat when held down.

EXAMPLE # 1

```
REPEAT ON <RETURN>
```

EXAMPLE # 2

```
10 CLS : LOWER : REPEAT ON
20 INPUT "Enter The Header";A$
30 REPEAT OFF : UPPER
```

KEYWORD: **NORM**

ALTERNATE: **NO<SHIFT/R>**

FORMAT: NORM

MODE: DIRECT and PROGRAM

NORM clears the screen, and resets the screen, border and cursor colors to their original values. Border = Cyan (3): Screen = White (1): Cursor = Dark Grey (11)

EXAMPLE # 1

```
                NORM  <RETURN>
```

EXAMPLE # 2

```
31Ø IF A$ = "Q" THEN NORM : END
```

KEYWORD: **STOP**

ALTERNATE: **S<SHIFT/T>**

FORMAT: STOP [a$]

(a$ is either ON or OFF)
(NOTE: quote marks not needed)

MODE: DIRECT and PROGRAM

The normal Commodore 64 Basic function of the STOP command (to stop program execution) is still supported, but S'more adds an additional function to this command. The STOP OFF command will cause the RUN/STOP and RESTORE keys to be de-activated. Unlike previous methods of de-activating these keys, STOP OFF does not interfere with the TI/TI$ function accuracy. STOP ON re-activates the RUN/STOP and RESTORE keys.

EXAMPLE # 1

```
STOP ON  <RETURN>
```

EXAMPLE # 2

```
10 STOP OFF : CLS
20 'BODY OF PROGRAM
99 IF A$ = "Q" THEN STOP ON : NORM : STOP
```

KEYWORD: **RESET**

ALTERNATE: **NONE**

FORMAT: RESET

MODE: DIRECT and PROGRAM

The RESET command will cause a system reset. The reset is a warm start type system reset. Screen and border colors are reset, and the screen looks like you just turned the computer on.

EXAMPLE # 1

```
                RESET  <RETURN>
```

EXAMPLE # 2

```
10 STOP OFF : CLS
20 'BODY OF PROGRAM
99 IF A$ = "Q" THEN RESET
```

KEYWORD: **MONITOR**

ALTERNATE: **M<SHIFT/O>**

FORMAT: MONITOR

MODE: DIRECT and PROGRAM

This command should be used when you have a Machine Language Monitor program loaded and initialized. The MONITOR command will cause a BRK to be executed and will jump to the location indicated by the monitor.

NOTE: We have provided a public domain monitor on the disk that comes with this cartridge.

EXAMPLE # 1

```
MONITOR  <RETURN>
```

EXAMPLE # 2

```
45 IF A$ = "M" THEN MONITOR
```

```
KEYWORD:    BORDER

ALTERNATE:  B<SHIFT/O>

FORMAT:     BORDER = a
               or
            a = BORDER

            (a is any value Ø - 15)


KEYWORD:    PAPER

ALTERNATE:  P<SHIFT/A>

FORMAT:     PAPER = a
               or
            a = PAPER

            (a is any value Ø - 15)


KEYWORD:    INK

ALTERNATE:  NONE

FORMAT:     INK = a
               or
            a = INK

            (a is any value Ø - 15)

MODE:       DIRECT and PROGRAM
```

All three of these commands work exactly the same way. These commands are provided as quick and easy ways to change the border, screen and cursor colors. BORDER refers to the color of the border, PAPER is the screen color and INK is the character color (or cursor color). BORDER, PAPER and INK are treated as variables within S'more, so you can do anything with them that you would do with any other numeric type variable. They can be used in the direct mode for example:

```
BORDER=4  <RETURN>
```

The above direct statement will change the border color to color # 4 (which, by the way, is purple) and assign the value of 4 to the variable BORDER. If you should want to know what color the border is (maybe you're color blind) use the direct statement as follows:

```
PRINT BORDER  <RETURN>
```

If the border color was 4, as assigned in the previous example, your computer will print the number 4 on the screen, thereby telling you that the variable border is assigned the value of 4. Again, remember to think of BORDER, PAPER and INK as numeric variables, and the values of the variables will always be equal to the appropriate color.

BORDER, PAPER and INK can also be used as variables in the program mode. You can assign a new value to the variable to change the color of the border or use the value of the variable BORDER just as you would use the value of any other variable. Here are several examples of using BORDER as a variable within a program.

EXAMPLE # 1

```
10 CLS : PRINT "ENTER A NUMBER (0-255)"
20 INPUT A : IF A > 255 THEN GOTO 10
30 BORDER = A : GOTO 10
```

EXAMPLE # 2

```
10 IF BORDER=3 THEN BORDER=4 : ELSE BORDER=3
```

EXAMPLE # 3

```
10 PRINT STR$(BORDER)
```

EXAMPLE # 4

```
10 BORDER = BORDER + 1
20 PAPER = PAPER +1
30 IF PAPER = 15 THEN PAPER = 0 : GOTO 50
40 FOR I = 0 TO 50 : NEXT : GOTO 20
50 IF BORDER = 15 THEN BORDER = 0
60 GOTO 10
```

NOTE: BORDER, PAPER, and INK can't be used as the objects of a basic command either within or outside of a program. For example INPUT BORDER will not work! You should use the format INPUT A : BORDER = A

NOTE: The value of BORDER, PAPER or INK will always be a whole number from Ø to 15 inclusive. BORDER, PAPER and INK will accept values from Ø to 255. A number greater than 255 will give you an ILLEGAL QUANTITY ERROR. Using a statement like BORDER = 255 will not set the value of BORDER equal to 255. The VALUE of BORDER will always be less than 16. If you set BORDER to equal a value greater than 16, S'more will divide the value by 16 and set BORDER equal to the remainder. For example, BORDER = 33 will result in the value of border being set to the remainder of the equation 33/16 which is 1.

PART THREE

DiskQuick

The commands in this section will all deal with quicker, better and/or easier ways to get information into and out of your disk drive. Commodore 64 Basic's built-in limitations in this area are a real problem for most programmers. Doing simple things like reading a directory from a disk were tedious and time consuming.

S'more will not solve all of the problems with Commodore disk drive access, but it will eliminate most of the tedious programming required to perform most disk related operations.

KEYWORD: **CATALOG**

ALTERNATE: **C<SHIFT/A?**

FORMAT:
```
CATALOG [Da [,Ub]]
   or
CATALOG [Da [ON Ub]]
(a is the drive number Ø or 1)
(b is the device number 8 to 15)
Defaults: a=Ø, b=8
```

MODE: DIRECT and PROGRAM

The catalog command can be used to print the directory of the current disk in the drive/device specified to either the screen or to your printer (to print to the printer use OPENx,4: CMDx: CATALOG).

EXAMPLE # 1

```
OPEN4,4 : CMD4 : CATALOG  <RETURN>
```

EXAMPLE # 2

```
CATALOG D1,U9  <RETURN>
       or
CATALOG D1 ON U9  <RETURN>
```

EXAMPLE # 3

```
1Ø GETA$:A=VAL(CHR$(A$)):'GET DEVICE #
2Ø IFA<8THENGOTO1Ø:'BETWEEN 8 OR 9
3Ø CATALOG DØ,UA:GOTO1Ø
```

KEYWORD: **LOAD**

ALTERNATE: **L<SHIFT/O>**

FORMAT: LOAD a$ [,b [,c]]

KEYWORD: **SAVE**

ALTERNATE: **S<SHIFT/A>**

FORMAT: SAVE a$ [,b [,c]]

KEYWORD: **VERIFY**

ALTERNATE: **V<SHIFT/E>**

FORMAT: VERIFY a$ [,b [,c]]

FOR ALL OF THE ABOVE COMMANDS:

(a$ is the filename)
(b is the device number 1-15)
(c is the secondary address Ø-2)
Default: b=8 : c=Ø

MODE: DIRECT and PROGRAM

All of the normal disk commands listed have been given the default value of 8 for the device number so you don't need to type ,8 whenever you want to verify, save or load a program. The cassette can still be used, but you must specify the device number of ,1 to access the cassette.

The <SHIFT/RUN-STOP> key combination will now load and run the first program on your disk drive, instead of the cassette.

These commands will also ignore anything following the filename that is not preceded by a comma. This will allow you to list a directory, move your cursor to the line listing the program you want to load, type LOAD and press return.

EXAMPLE # 1

```
LOAD  "DEMO"          PRG
```

EXAMPLE # 2

```
20 LOAD"TEST":'FROM DISK
```

EXAMPLE # 3

```
9999 SAVE"`0:MYPROG":VERIFY"MYPROG"
```

KEYWORD: **RUN**

ALTERNATE: **R<SHIFT/U>**

FORMAT:
```
RUN [a]
  or
RUN [a$ [,b [,c]]]
(a is the starting line number)
(a$ is the filename to run)
(b is the device number 1-15)
(c is the secondary address Ø-1)
```

MODE: DIRECT and PROGRAM

The first format shown is the standard Commodore 64 Basic RUN command with the starting line number option. This format is supported by S'more.

The second format shown includes the program name, device number and secondary address options. This format allows you to specify a program and source (disk/tape device number) to be loaded and run. This command will also ignore anything following the filename that is not preceded by a comma. This will allow you to list a directory, move your cursor to the line listing the program you want to run, type RUN and press return.

The RUN command can also be used from within a program to load and run another program. Using one main menu program and including a RUN"MENU" statement at the end of each of the programs on the menu you can create a self contained system. (See the example below.)

EXAMPLE # 1

```
RUN "MENU"
```

EXAMPLE # 2

```
RUN "PROGRAM",9
```

EXAMPLE # 3

```
10 CLS : PRINT " SELECT A GAME TO PLAY"
20 PRINT : PRINT "1. NOMAD"
30 PRINT : PRINT "2. LOST IN HEAVEN"
40 PRINT : PRINT "3. MINE DISASTER"
50 PRINT : PRINT "4. TIC TAC TOE
60 PRINT : PRINT "5. MONOPOLY
70 PRINT : PRINT "6. BLACKJACK
80 PRINT : PRINT "7. ROULETTE
90 PRINT : PRINT "8. MONGOLIAN NERDS
100 GET A$ : RUN A$
```

NOTE: Programs saved under program names "1" (for NOMAD), "2" (for LOST...), etc.

KEYWORD: **MERGE**

ALTERNATE: **M<SHIFT/E>**

FORMAT: MERGE a$ [,b]

(a$ is the filename to be merged)
(b is the device number 1-15)
Default: b=8

MODE: DIRECT

The MERGE command allows you to combine (or merge) a program stored on disk or tape with the program in memory. MERGE does a complete inter-locking combining of the two programs. If one program contains lines 1, 3, 7 & 2Ø; and the other contains 2, 6 & 16; the resulting program will have lines 1, 2, 3, 6, 7, 16 & 2Ø.

NOTE CAUTION: If a line with the same line number exists in both programs, the line from the program merged (from disk or tape) will replace the existing line from the program in memory.

NOTE: This form of combining programs can be time consuming. With large programs it can require several minutes to complete the merge, even after the disk drive (or cassette) has stopped running. It is faster to load the larger of the programs into memory and then MERGE the shorter program into the longer one.

EXAMPLE # 1

```
        MERGE "SCREEN FORMAT"  <RETURN>
```

EXAMPLE # 2

```
          MERGE "UTILITY",9  <RETURN>
```

EXAMPLE # 3

```
         MERGE "FROM TAPE",1  <RETURN>
```

KEYWORD: **DISK**

ALTERNATE: **DI<SHIFT/S>**

FORMAT: DISK a$ [,b]

(a$ is any valid disk command)
(b is the device number)

MODE: DIRECT and PROGRAM

The DISK command eliminates the constant opening and closing of the command channel (OPEN15,x,15) to the disk drive that is required by Commodore 64 Basic. The DISK command replaces OPEN 15,b,15,a$:CLOSE 15

EXAMPLE # 1

To format a disk with Commodore Basic

```
   OPEN 15,8,15,"NØ:MYDISK,MD" : CLOSE 15
```

The same command with S'more

```
          DISK "NØ:MYDISK,MD"
```

In the program mode you never have to worry about leaving the command channel open accidentally and getting a FILE OPEN error. (See the following example.)

NOTE: DISK will not close or interfere with any open channels to the disk drive.

EXAMPLE # 2

```
 10 CLS : PRINT : PRINT DS$ : PRINT : PRINT
    "SELECT FUNCTION:"
 20 PRINT : PRINT "1. SEE DIRECTORY"
 30 PRINT : PRINT "2. INITIALIZE DRIVE"
 40 PRINT : PRINT "3. FORMAT A DISK
 50 PRINT : PRINT "4. DELETE A FILE
 60 PRINT : PRINT "5. RENAME A FILE
 70 GET A$ : ON VAL(A$)+1 GOTO 70, 80, 90,
    100, 130, 150 : GOTO 70
 80 CLS : CATALOG : GOTO300
 90 A$ = "I0" : GOTO 200
100 PRINT : INPUT " ENTER NAME OF DISK ";N$:
    IF LEN(N$) > 15 THEN GOTO100
110 PRINT : INPUT " ENTER DISK ID # ";I$:
    IF LEN(I$) <> 2 THEN GOTO110
120 A$="N0:" + N$ + "," + I$ : GOTO 200
130 PRINT : INPUT " ENTER NAME OF FILE ";N$:
    IF LEN(N$) > 15 THEN GOTO130
140 A$= "S0:" + N$ : GOTO 200
150 PRINT : INPUT " ENTER OLD FILENAME";O$:
    IF LEN(O$) > 15 THEN GOTO150
160 PRINT : INPUT " ENTER NEW FILENAME";N$:
    IF LEN(N$) > 15 THEN GOTO160
170 A$= "R0:" + N$ + "=0:" + O$
200 DISK A$ : CLS : PRINT : MS$ = DS$ :
    CATALOG : PRINT : PRINT MS$
300 GET A$ : IF A$ = "" THEN GOTO 300
400 GOTO 10
```

NOTE: This program will be also be found on the disk that is included with S'more. LOAD"DISKQUICK"

KEYWORD: **DS**

ALTERNATE: **NONE**

FORMAT:
```
PRINT DS
   or
A = DS
```

KEYWORD: **DS$**

ALTERNATE: **NONE**

FORMAT:
```
PRINT DS$
   or
A$ = DS$
```

MODE: DIRECT and PROGRAM

Using the DS and DS$ functions will give you quick access to the DOS error messages from the disk drive error channel. The DS and DS$ functions eliminate the need to OPEN the drive error channel and then INPUT the error message. DS itself is treated like a reserved variable in S'more Basic. DS$ is treated like a reserved string variable. DS will always be equal to the current drive error number and DS$ will always be equal to the current drive error message. A listing of the DOS error numbers and messages and their meanings can be found in the information contained in your disk drive operating manual.

Remember that each time you use the DS/DS$ function, S'more reads the DOS error message. Any other disk command will clear the error channel. One method to retain the value of DS/DS$ for later use is shown in the EXAMPLE # 2 program for the DISK command. (See line number 2ØØ; MS = DS)

NOTE: One **very** important benefit of the DS and DS$ functions is that they **CAN** be used in the DIRECT mode, unlike standard Commodore 64 Basic which has no provisions to read the disk drive error channel in the DIRECT mode.

EXAMPLE # 1

```
                PRINT DS$  <RETURN>
```

EXAMPLE # 2

```
1Ø A=DS: REM - READ DOS ERROR CHANNEL
2Ø IF A <> Ø THEN PRINT "DISK ERROR" DS$
```

KEYWORD:	**DOPEN#**
ALTERNATE:	**DO<SHIFT/P>**
FORMAT:	DOPEN#a,b,c,d$,e (a is the logical file number) (b is the device number) (c is the secondary address) (d$ is the filename) (e is the relative file length)
MODE:	DIRECT and PROGRAM

The DOPEN# command is used for creating relative files. The command format requires the following information:

1. A logical file number between 1 and 127. You may have only one file with this logical file number open at any one time, but you may have several files open with different logical file numbers at the same time.

2. A device number between 8 and 15 for the disk drive desired.

3. A secondary address between 2 and 14.

4. The file name of the file to be used.

5. The record length for the records in the file between 2 and 254.

For a complete explanation of relative files and their uses we recommend that you refer to the section on relative files in the manual that was supplied with your disk drive.

The DOPEN# command may be used in place of the normal Commodore OPEN command.

EXAMPLE # 1

To open a relative file with a record length of 200 characters in Commodore 64 Basic:

```
10 OPEN8,8,8,"RELATIVE,L"+CHR$(200)
```

To open a relative file with a record length of 200 characters with S'more:

```
10 DOPEN#8,8,8,"RELATIVE",200
```

NOTE: Use RECORD# (next page) to access information within relative files

NOTE: After the relative file exists, the normal OPEN8,8,8,"FILE" command format may be used to access the file.

KEYWORD: **RECORD#**

ALTERNATE: **RE<SHIFT/C>**

FORMAT: RECORD#a,b [,c]

(a is the logical file number)
(b is the record number)
(c is the pointer to the position within the record to start at.)
Default: c=Ø

MODE: DIRECT and PROGRAM

The RECORD# command is used to access individual records within relative files. The logical file number refers to the logical file number assigned to the file when it was opened using the DOPEN# command (See DOPEN#). The record number refers to the record number of the record within the logical file that you wish to access. A relative file on a 1541 Commodore drive may contain up to 72Ø individual records. (See the manual that came with your disk drive for limits on the number of records available.) A record may be (as you declared in the DOPEN#'s record length parameter) up to 254 characters long, the pointer value allows you set the pointer to any desired start position within the record for the next read or write function to that record.

Once you have set up the file and record using the DOPEN# and RECORD# commands you can now put data in the record or retrieve data from the record using the standard PRINT#, GET#, and INPUT# statements. See the appendix of this manual for a program (the program is also on the disk that comes with S'more, LOAD"MAIL") that demonstrates the use of relative files to do a mailing list.

EXAMPLE # 1

```
20 RECORD#8,37
```

* Finds the 37th record in the file.

EXAMPLE # 2

```
20 RECORD#8,25,10
```

* Finds the 25th record in the file and sets the pointer to the 10th character in the record.

EXAMPLE # 3

```
10 OPEN#3,9,3,"LIST"
20 FOR I = 0 TO 99
30 RECORD#3,I : A$(I) = ""
40 GET#3,A$ : A$(I) = A$(I) + A$
50 IF A$ <> CHR$(13) THEN GOTO 40
60 NEXT : CLOSE3
```

PART FOUR

OUT WITH STYLE

The commands in this section will let you print, print-out and output information with a lot more ease than Commodore 64 Basic. These commands specifically control screen and printer output formatting functions. As you will see creating attractive, easy-to-use screens and reports need not be a time-consuming chore.

Printing neatly organized and formatted output to your screen and printer will become a relative easy task with S'more's powerful printing commands. Spending a little extra time learning to use the commands in this section will be a valuable investment of your time.

KEYWORD: **AT**

ALTERNATE: **CHR$(1)**

FORMAT: AT a,b
or
CHR$(1)CHR$(a)CHR$(b)

(a is the row (Ø to 24))
(b is the column (Ø-39))
Defaults: a=Ø : b=Ø

MODE: DIRECT and PROGRAM

The AT command is used to place the cursor at the desired position on your screen. The AT command can only be used with statements that print to the screen. (IE. PRINT AT, INPUT AT and INFORM AT, INLINE AT, which will all be discussed in this section.)

There are two formats for the AT command. The first is simple enough, the KEYWORD AT is followed by two decimal numbers representing the row and column desired as the cursor position. For example:

```
PRINT AT 2Ø,2 "PRESS ANY KEY"  <RETURN>
```

This command will print the statement "PRESS ANY KEY" at the 3rd print position (Ø is the 1st print position) on the 21st line (Ø is the 1st line) of the screen.

Multiple AT statements can be used. For example:

```
50 A=3 : B=3 : C=18 : D=13
60 PRINT AT A,B "HERE" AT C,D "THERE"
```

The alternate format of the AT command is available only with the PRINT command, and has been provided to allow you to send the AT command as part of a string, for example:

```
30 A$=CHR$(1) : B$=CHR$(2) : C$=CHR$(0) :
   D$=CHR$(25)
40 PK$=A$+D$+B$"PRESS ANY KEY"+A$+C$+C$
50 PRINT PK$
```

Remember, the examples of the AT command shown here used with the PRINT command can also be used with the INPUT, INLINE and INFORM commands described later in this chapter.

EXAMPLE # 1

```
10 CLS : FOR I = 0 TO 20
20 PRINT AT I,0 I : NEXT
30 FOR I = 0 TO 20
40 PRINT AT I,4 LEFT$(A$(I),14): NEXT
50 FOR I = 0 TO 20
60 PRINT AT I,20 I+20 : NEXT
70 FOR I = 0 TO 20
80 PRINT AT I,24 LEFT$(A$(I+20),14): NEXT
```

EXAMPLE # 2

```
10 DIM P$(39),Q$(7,99) : FOR I = Ø TO 39 :
   P$(I) = CHR$(I) : NEXT
20 Q$=CHR$(Ø)+CHR$(1)+CHR$(Ø)+"ENTER * TO
   END DATA ENTRY"
30 Q$=CHR$(Ø)+CHR$(1)+CHR$(Ø)+"=========
   ==============="
40 Q$(Ø)=P$(Ø)+P$(3)+P$(Ø)+"FIRST NAME:  "+Q$
50 Q$(1)=P$(Ø)+P$(5)+P$(Ø)+"LAST NAME:   "+S$
60 Q$(2)=P$(Ø)+P$(7)+P$(Ø)+"ADDRESS:     "
70 Q$(3)=P$(Ø)+P$(9)+P$(Ø)+"CITY:        "
80 Q$(4)=P$(Ø)+P$(9)+P$(29)+"STATE:      "
90 Q$(5)=P$(Ø)+P$(11)+P$(Ø)+"ZIP CODE:   "
100 Q$(6)=P$(Ø)+P$(13)+P$(Ø)+"TELEPHONE:  "
110 Q$(7)=P$(Ø)+P$(15)+P$(3)+"NOTES:      "+
    P$(Ø)+P$(17)+P$(Ø)
120 L(Ø)=24:L(1)=24:L(2)=24:L(3)=15:L(4)=2
    L(5)=11:L(6)=14:L(7)=8Ø:
130 FOR II=ØTO99 : CLS
140 FOR I=ØTO7 : PRINT Q$(I) : NEXT
150 FORI=ØTO7: PRINTQ$(I);: INFORM;LEN(L(I):
    Q$(I,II)
160 IF Q$(Ø,II) = "*" THEN I=8 : N= II :
    II=1ØØ : NEXT : NEXT : GOTO 3ØØ
170 NEXT
180 PRINT AT 21,2"PRESS * IF ALL CORRECT"
190 PRINT AT 23,2"PRESS SPACE BAR TO MAKE
    CORRECTIONS";:GETKEY A$
200 IF A$ = "*" THEN NEXT : GOTO 3ØØ
210 GOTO 13Ø
```

NOTE: This is a complete formatted data input screen. With a little modification it could be used for any type of data entry.

KEYWORD:	**USING**
ALTERNATE:	**US<SHIFT/I>**
FORMAT:	USING a$; b[$] [,c[$] ,..etc] (a$ is the FORMAT STRING (See the text for details) (b[$] can be any numeric or string variable) (c[$]...etc can be a list of numeric or string variables)
MODE:	DIRECT and PROGRAM

THE USING command must follow a PRINT or PRINT#, command. For example all of the following variations are valid commands:

```
        PRINT USING A$;B
              or
     PRINT US<SHIFT/I/ A$;B
              or
          ? USING A$;B
              or
       ? US<SHIFT/I> A$;B
              or
      PRINT#1, USING A$;B
              or
   PRINT#1, US<SHIFT/I> A$;B
              or
    P<SHIFT/R>1, USING A$;B
              or
 P<SHIFT/R>1, US<SHIFT/I> A$;B
```

The USING command provides a simple means to print formatted data to the screen, printer, disk or other device. If you wanted to print out a list of numbers on the screen with all of the decimal points aligned, this would be an easy task for the USING command. Understanding and using the USING command will save you huge amounts of time and memory. It will be impossible to cover all of the many possible uses for the USING command in this manual, but the examples and recommended uses included here will give you some idea of the potential of this extremely powerful command.

First you must understand how the command works. The format of the USING command requires you to provide a string variable called the FORMAT STRING. The FORMAT STRING is a group of literal and special characters, each with a special meaning, that define a format (think of the it as a template or a mold) that the item(s) you want printed are put into. Within the template, the special characters can reserve places for one or several items. Each of the places reserved by the special characters is called a FIELD.

The special characters that are reserved for use within the USING statement are:

POUND SIGN or <SHIFT/3>	#
GREATER THAN SIGN or <SHIFT/.>	>
EQUAL SIGN	=
PLUS SIGN	+
MINUS SIGN	-
DECIMAL POINT	.
COMMA	,
DOLLAR SIGN or <SHIFT/4>	$
FOUR CARETS (UP ARROWS)	^^^^

All of the other characters are treated as literal or real characters. That is, if a literal character is included in a FORMAT STRING it is printed exactly where and as it is in the FORMAT STRING. To understand this concept fully you must first understand what a special character is and does.

*** USING WITH STRINGS**

SPECIAL CHARACTER (#)

The simplest to understand and most used special character is the NUMBER (#) or <SHIFT/3>). The NUMBER (#) sets aside space (called a FIELD) in your FORMAT STRING for one character. So if you define a FORMAT STRING as A$="#####" you would have a FIELD five (5) characters long. If you then printed something USING A$, whatever you print would be exactly five (5) characters long. When USING is asked to print an item that is shorter than the FIELD reserved for it in the FORMAT STRING, it adds filler characters (which by the way can be defined to be any character, but default to blank spaces unless redefined - See PUDEF) to make up the difference. If the item is too long, it is simply truncated (chopped off) to fit. Here are some examples (remember this is starting at the simplest level) of how USING would print various data:

EXAMPLE # 1

```
    PRINT USING "#####";"DATA"  <RETURN>
```

RESULT:

DATA (followed by one space)

EXAMPLE # 2

```
     PRINT USING "#####";"HI"  <RETURN>
```

RESULT:

HI (followed by three spaces)

EXAMPLE # 3

```
  PRINT USING "#####";"HOLLYWOOD"  <RETURN>
```

RESULT:

HOLLY

Well that's only a start, but be sure you understand the concept before going on.

Now remember that you can print as many items as you want with the USING command, for example:

EXAMPLE # 1

```
   PRINT USING "##";"A",BC",DEF"  <RETURN>
```

RESULT:

```
A BCDE
```

NOTE: The Blank space after A.

EXAMPLE # 2

```
10 US$ = "####"
20 A$="A": B$="BC": C$="DEF: D$="GHIJ":
   E$="KLMNO": F$="PQRSTU"
30 PRINT USING US$;A$,B$,C$,D$,E$,F$
```

RESULT:

```
A   BC  DEF GHIJKLMNPQRS
```

NOTE: Three blank spaces were added to A$ ("A"), two to B$ ("BC"), and one was added to C$ ("DEF"), D$ was left unchanged and E$ and F$ were truncated to be only four characters long "KLMN" and PQRS" respectively.

SPECIAL CHARACTER (>)

There are several other things that the USING command does with strings. You can request that the USING command print strings right justified. To right justify a string the USING command fills in the FIELD in the FORMAT STRING with spaces at the beginning of the string instead of the end. To tell the USING command to right justify a string use the special character GREATER THAN (>) or (<SHIFT/.>) as part of your FIELD definition within the FORMAT STRING. Remember that GREATER THAN (>) is counted by USING when determining the FIELD length. (I.E. "##>#" has a FIELD length of four (4) characters.) The following example uses the same information as the last example but will right justify the strings:

```
10 US$ = "#>##"
20 A$="A": B$="BC": C$="DEF: D$="GHIJ":
   E$="KLMNO": F$="PQRSTU"
30 PRINT USING US$;A$,B$,C$,D$,E$,F$
```

RESULT:

```
   A  BC DEFGHIJKLMNPQRS
```

NOTE: Three blank spaces were added in front of the A in A$, two in front of the "BC" in B$, one in front of the DEF in C$, D$ was again left unchanged and E$ and F$ were still truncated to be only four characters long "KLMN" and PQRS" respectively.

SPECIAL CHARACTER (=)

USING can also center an item within a FIELD by use of EQUAL (=). As with right justification, if the length of the string is less than the length of the defined FIELD, USING adds spaces to the string until it is the desired length. USING starts by adding a space to the right of (behind) the string, then checks to see if the string is long enough. If it is still too short, USING adds a space to the left (in front) of the string. This process goes on until the string reaches the desired length. For example:

EXAMPLE

```
10 US$="###=####":' FIELD LENGTH OF 8
20 READ A$ : PRINT USING US$;A$
30 IF LEN(A$) < 10 THEN GOTO 20
40 DATA A,AB,ABC,ABCD,ABCDE,ABCDEF
50 DATA ABCDEFGH,ABCDEFGHI,ABCDEFGHIJ
```

RESULT:

```
    A          (followed by 4 spaces)
    AB         (followed by 3 spaces)
   ABC         (followed by 3 spaces)
   ABCD        (followed by 2 spaces)
  ABCDE        (followed by 2 spaces)
  ABCDEF       (followed by 1 space)
 ABCDEFG       (followed by 1 space)
 ABCDEFGH      (leading space added)
ABCDEFGHI      (unchanged)
ABCDEFGHI      (truncated)
```

LITERAL CHARACTERS

Hopefully, by now, you understand the concept that a FIELD is an area within a FORMAT STRING that is reserved for information that you want printed in a certain format. Literal characters on the other hand are printed exactly as they appear in the format string. For example:

EXAMPLE # 1

```
10 US$="HELLO ########"
20 A$="BILL": B$="MICHELLE"
30 PRINT USING US$;A$
40 PRINT USING US$;B$
```

RESULT:

```
HELLO BILL
HELLO MICHELLE
```

EXAMPLE # 2

```
10 US$="ORIENT#######"
20 A$="S": B$="ED" : C$="ATION"
30 PRINT USING US$;A$
40 PRINT USING US$;B$
50 PRINT USING US$;C$
```

RESULT:

```
ORIENTS
ORIENTED
ORIENTATION
```

EXAMPLE # 3

```
10 US$="THE ##=## BALL"
20 A$="RED": B$="BLUE" : C$="BLACK"
30 PRINT USING US$;A$
40 PRINT USING US$;B$
50 PRINT USING US$;C$
```

RESULT:

```
THE  RED  BALL
THE  BLUE BALL
THE BLACK BALL
```

Literal characters also serve as FIELD separators within a FORMAT STRING. You can construct FORMAT STRINGs up to 254 characters long that use many variables in this manner. For example:

EXAMPLE # 1

```
10 US$="NAME: ############# "
20 US$=US$+"ADDRESS: ################ "
30 US$=US$+"#>########### ## #####"
40 READ N$,A$,C$,S$.Z$,X$
50 PRINT USING US$;N$,A$,C$,S$,Z$
60 IF Z$= "" THEN GOTO 40
70 DATA Thomas Bill,123 Here St,Heroville,
   ME,01223,
80 DATA Thornton Sue,9999 Some St,Oakton,IL,
   60607,
90 DATA Clemente David,1212 Cottonwood Dr,
   Wichita,KS,67207,-
```

RESULT:

```
NAME: Thomas Bill   ADDRESS: 123 Here St.          Heroville ME Ø1223
NAME: Thornton Sue  ADDRESS: 9999 Some St            Oakton IL 6Ø6Ø7
NAME: Clemente Davi ADDRESS: 1212 Cottonwood        Wichita KS 672Ø7
```

Remember that you can use any character as a literal character. If you are printing a form for example you could use characters in the FORMAT STRING to print items in reversed or double size type. You can even include the <RETURN> character (CHR$(13)) like this:

EXAMPLE # 1

```
1Ø US$="NAME:     ######################"+
   CHR$(13)
2Ø US$=US$+"ADDRESS: #####################"
   +CHR$(13)+CHR$(13)
3Ø US$=US$+"#>########### ## #####"+
   CHR$(13)+CHR$(13)
4Ø READ N$,A$,C$,S$.Z$,X$
5Ø PRINT USING US$;N$,A$,C$,S$,Z$
6Ø IF X$= "" THEN GOTO 4Ø
7Ø DATA Thomas Bill,123 Here St,Heroville,
   ME,Ø1223,
8Ø DATA Thornton Sue,9999 Some St,Oakton,IL,
   6Ø6Ø7,
9Ø DATA Clemente David,1212 Cottonwood Dr,
   Wichita,KS,672Ø7,-
```

RESULT:

```
NAME:    Thomas Bill
ADDRESS: 123 Here St

    Heroville ME Ø1223


NAME:    Thornton Sue
ADDRESS: 9999 Some St

       Oakton IL 6Ø6Ø7


NAME:    Clemente David
ADDRESS: 1212 Cottonwood Dr

      Wichita KS 672Ø7
```

That covers all of the USING command's special characters that are used with strings. The variations presented here are only a small part of the potential uses for string manipulation with the USING command. Let your imagination be your guide and experimentation will provide many, many useful variations for this very powerful command.

* USING WITH NUMBERS

SPECIAL CHARACTER (#)

As with string values, the NUMBER (#) sets aside space (a FIELD) in your FORMAT STRING for one digit. So if you define a FORMAT STRING as A$="#####" you would have a FIELD five (5) digits long. If you then printed something USING A$, whatever you print would be exactly five (5) digits long. When USING is asked to print an item that is shorter than the FIELD reserved for it in the FORMAT STRING, it adds filler characters (which can be defined to be any character, but default to blank spaces unless redefined - See PUDEF) to make up the difference. Unlike strings however, USING always right justifies numbers by adding the filler characters (normally spaces) in front of the number. If the item is too long, simply chopping off the excess numbers to fit as is done with string data would create a real mess, so USING fills the area with *'s (asterisks) to tell you that your number was too big to fit. Here are some examples of USING with numeric data:

EXAMPLE

```
10 A$="#####" : READ A
20 PRINT USING A$;A
30 IF A<>123456 THEN GOTO 20
40 DATA 1
50 DATA 12
60 DATA 123
70 DATA 1234
80 DATA 12345
90 DATA 123456
```

RESULT:

```
    1   (led by 4 spaces)
   12   (led by 3 spaces)
  123   (led by 2 spaces)
 1234   (led by 1 space)
12345
*****
```

There are a couple of unexpected things that happen with numeric values and the USING command. For example try this:
EXAMPLE

```
    PRINT USING "###";123,-123  <RETURN>
```

RESULT:

```
123
***
```

Why did the negative value of the number overflow and cause the asterisks to be printed? Because the minus sign is part of the number and there wasn't any place to put it. There are two special characters provided by USING to take care of this situation.

SPECIAL CHARACTERS (+) & (-)

The PLUS (+) and MINUS (-) characters are special characters that can be used to both provide room for and define the location of the sign (+/-) of a numeric value. If you want a value to be printed in the traditional format with a MINUS (-) preceding all negative value numbers, but no PLUS (+) preceding positive numbers, all you need to do is put a MINUS (-) special character at the beginning of the FIELD. For example:

EXAMPLE

```
    PRINT USING "-###";123,-123  <RETURN>
```

RESULT:

```
 123
-123
```

If you want to show the sign of all values printed, you must use the PLUS (+) special character as the first character of the FIELD. For example:

EXAMPLE

```
    PRINT USING "+###";123,-123  <RETURN>
```

RESULT:

```
+123
-123
```

If you want a value to be printed in the traditional accounting type format with a MINUS (-) trailing all negative value numbers, but no PLUS (+) after the positive numbers, all you need to do is put a MINUS (-) special character at the end of the FIELD. For example:

EXAMPLE

```
    PRINT USING "###-";123,-123  <RETURN>
```

RESULT:

```
123  (with 1 trailing space)
123-
```

If you want the sign of all values to be printed following the value, you must use the PLUS (+) special character as the last character of the FIELD. For example:

EXAMPLE

```
    PRINT USING "###+";123,-123  <RETURN>
```

RESULT:

```
123+
123-
```

The only thing that will cause you any trouble with the +/- special characters is the fact that you can only choose one option for each FIELD. In other words you can't have a PLUS (+) or MINUS (-) at both the beginning and the end of the FIELD.

SPECIAL CHARACTER (.) (DECIMAL)

Here's another oddity to try out:

```
    PRINT USING "######";123.45  <RETURN>
```

RESULT:

```
123
```

What happened to the .45? Well, USING expects whole numbers unless you direct it to do otherwise. If USING is expecting a whole number and you supply a fractional value, USING rounds off (this is a true rounding function) to the nearest whole number before it prints the value. Here are some examples of how USING treats fractional values:

EXAMPLE

```
10 US$="#####"
20 READ A: PRINT USING US$;A
30 IF A<>99999.5 THEN GOTO 20
40 DATA 5.4999,5.5
50 DATA 100.000001,100.99999
60 DATA 55555.111111,55555.9999999
70 DATA 99999.49,99999.5
```

RESULT:

```
    5
    6
  100
  101
55555
55556
99999
*****
```

In fact, the only way to get USING to print a fractional (decimal) value is to place the DECIMAL (.) special character at the point in the FIELD that you want the decimal point to appear. If you place the DECIMAL (.) special character in a FIELD, USING will format numeric values according to the following rules:

RULES FOR DECIMAL FIELDS

If there are too many digits, including the minus sign if required, on the left of the DECIMAL (.) to fit in the space allotted in the FIELD, the FIELD will be filled with asterisks (*).

If extra digits exist to the right of the DECIMAL (.), they will be **rounded** to fit the space allocated in the FIELD.

If there are no digits to the left of the DECIMAL (.) a single ZERO (Ø) will be added to the left of the DECIMAL (.)

If there are not enough digits to the left of the DECIMAL (.) to fill the FIELD, spaces will be added to fill out the FIELD.

If there are not enough digits to the right of the DECIMAL (.) to fill the FIELD, ZEROES (Ø) will be added to fill out the FIELD.

Some examples of the above rules should help you to understand them better:

VALUE	FIELD	RESULT
1ØØØ	###.###	*******
-1ØØ	###.###	*******
999.9995	###.###	*******
999.99949	###.###	999.999
1ØØ	###.###	1ØØ.ØØØ
-1Ø	###.###	-1Ø.ØØØ
1Ø	###.###	1Ø.ØØØ
1	###.###	1.ØØØ
-.1	###.###	-Ø.1ØØ
.1	###.###	Ø.1ØØ
-.Ø1	###.###	-Ø.Ø1Ø
.Ø1	###.###	Ø.Ø1Ø
.ØØ1	###.###	Ø.ØØ1
-.ØØØ1	###.###	Ø.ØØØ
.ØØØ5	###.###	Ø.ØØ1
.ØØØ49	###.###	Ø.ØØØ

SPECIAL CHARACTER (,) (COMMA)

The COMMA (,) special character can also be used within a numeric field as shown:

VALUE	FIELD	RESULT
1234.56	#,###.##	1,234.56
234.56	#,###.##	234.56
1234567	#,###,###	1,234,567
34	#,###,###	34

SPECIAL CHARACTER ($) (DOLLAR)

The DOLLAR ($) special character can be used in either of two ways. If the DOLLAR ($) is placed as the first character of a FIELD, a FIXED DOLLAR SIGN ($) will be printed in that position with every value. For example:

VALUE	FIELD	RESULT
.Ø1	$###.##	$ Ø.Ø1
.1	$###.##	$ Ø.1Ø
1Ø	$###.##	$ 1Ø.ØØ
1ØØ	$###.##	$1ØØ.ØØ

If the DOLLAR ($) is placed as the second character of the FIELD, a FLOATING DOLLAR SIGN ($) will be printed in front of every number. For example:

VALUE	FIELD	RESULT
.Ø1	#$##.##	$Ø.Ø1
.1	#$##.##	$Ø.1Ø
1	#$##.##	$1.ØØ
1ØØ	#$##.##	$1ØØ.ØØ

SPECIAL CHARACTER ^^^^ (FOUR CARETS)

The FOUR CARETS (^^^^) or (UP ARROWS ON YOUR KEYBOARD) are used to signify that the FIELD is to be printed using scientific notation. All of the rules for numeric FIELDS and decimal placement apply and the FOUR CARETS (^^^^) must be the last characters of the FIELD definition in the FORMAT STRING.

For example:

VALUE	FIELD	RESULT
1ØØØØØØ	##^^^^	1ØE+Ø5
1ØØØØØØ	#^^^^	1E+Ø6
1ØØØØØØ	#.#^^^^	1.ØE+Ø6
-.ØØØ1Ø5	+#.#^^^^	-1.1E-Ø4

PRINT# AND THE USING COMMAND

The USING command can also be used with the PRINT# command. The USING command will therefore allow you to send formatted information to any device on the serial bus. The value of sending formatted information to the printer is easy to realize, but most people overlook the power USING adds to relative files. By creating a FORMAT STRING and using the USING command to print information to a relative file you can be sure that your data will never exceed the record size space allocated by the relative file LEN parameter. The USING command will also allow you to precisely locate data within each record, allowing easy access to individual data items within each record by using the position pointer parameter of the RECORD# command.

KEYWORD: **PUDEF**

ALTERNATE: **PU<SHIFT/D>**

FORMAT: PUDEF "abcd"

```
(a is the filler character)
(b is the comma character)
(c is the decimal point)
(d is the leading dollar sign)
Default: a=" ":b=",":c=".":d="$"
```

MODE: DIRECT and PROGRAM

PUDEF is the command that allows you to change some of the special characters used within the USING command. PUDEF only affects the USING command if the variable to be printed is a numeric variable. If you lived in England, you would probably want to use the British Pound sign in place of the U.S. Dollar sign. The four special characters that can be redefined. They are the SPACE ( ), COMMA (,), DECIMAL (.) and DOLLAR ($) characters. As stated in the description of the USING command, the SPACE ( ) character is used as a filler character to fill strings or numbers to the proper FIELD size. PUDEF allows you to change the SPACE ( ) to be any other keyboard character. Here is an example of the PUDEF command being used to change the SPACE ( ) character to the ASTERISK (*) character:

EXAMPLE # 1

```
1Ø US$="##########" : A = 12345
2Ø PRINT USING US$;A
3Ø PUDEF"*,.$"
4Ø PRINT USING US$;A
```

RESULT:

```
     12345
*****12345
```

EXAMPLE # 2

```
1Ø US$="######.##" : A=123.4
2Ø PRINT USING US$;A
3Ø PUDEF" ,:$"
4Ø PRINT USING US$;A
```

RESULT:

```
   123.4Ø
   123:4Ø
```

Any of the four special characters can be redefined using the PUDEF command, by simply placing the desired replacement character in the position occupied by the character to be replaced.

PART FIVE

IN STYLE

Commodore 64 Basic provides only two commands to get data from input devices like your keyboard, disk drive and cassette. The limitations of this lack of diversity make some operations very difficult if not impossible to accomplish.

The additional input commands provided by S'more will allow you more flexibility in constructing programs. The commands are designed to allow you more control over both the format and the contents of the data received, while cutting substantially the number of program statements required.

KEYWORD: **INPUT**

ALTERNATE: **NONE**

FORMAT: INPUT [AT a[,b]]; [c$;] d[$]

(AT a and b: See AT)
(c$ any text string in quotes)
(d[$] is the numeric or string variable to be input)

MODE: PROGRAM

This command works just like the Commodore 64 Basic INPUT command except that it is able to use the AT command for formatted location of the input. (See AT.)

NOTE: The bug in the Commodore 64 Basic version of the command has been repaired in the S'more version of the INPUT command. Unlike the Commodore 64 Basic INPUT command, S'more's version will not input the prompt as part of the data if the data fills up the line and is continued on the next line.

EXAMPLE # 1

```
10 INPUT A
20 INPUT "NAME: "; B$
30 INPUT "AGE:  "; B
40 INPUT AT 7,2;"NAME: "; C$
```

KEYWORD: **INLINE**

ALTERNATE: **NONE**

FORMAT: INLINE [AT a[,b]]; [c$;] d$

(a and b See AT)
(c$ any text string in quotes)
(d$ is the variable to be input)

MODE: PROGRAM

Unlike the INPUT command, INLINE will not accept a numeric variable in its format as the variable to be input, it works with string variables only.

EXAMPLE

```
10 INPUT A :' THIS IS OK
20 INPUT A$:' THIS IS OK
30 INLINE A : ' THIS CAUSES A SYNTAX ERROR
30 INLINE A$ : ' THIS IS OK
```

INLINE will **NOT** provide the ? prompt that INPUT does, it simply puts a flashing cursor where it expects data to begin. INLINE also accepts all characters including colons, commas and quotation marks as part of the inputted string variable. (No "EXTRA IGNORED" error message is generated when INLINE encounters a comma in the string data received.) With INLINE it would be possible to input **Los "O" San, Ca.** as one string

INLINE reads all characters from the position the cursor first appears on the screen to the end of the logical screen line. Experimentation will provide the best explanation as to how this works. Try the following examples and see what happens:

Example

```
10 CLS : LOWER : REPEAT ON
20 PRINT AT A,B;80-B-33:INLINEATA,B;
   '****CHARACTERS MAY BE ENTERED : ';A$
30 CLS : PRINT'HERE IS WHAT INLINE SAW
   AS YOUR RESPONSE':
40 PRINT AT 6,0;A$
50 INFORM AT 10,0;'TRY AGAIN (Y/N): ';
   LEN(1);A$ : IF A$='N'THEN NORM : END
60 A=A+3 : IF A>24 THEN A=A-23
70 B=B+5 : IF B>39 THEN B=B-38
80 goto10
```

* NOTE: The *'s should be CURSOR LEFTS.

This program will prompt you for data from many different screen locations, enter a lot of characters, or enter only a few, and see what happens. The program will prompt you for a YES or NO to continue.

NOTE: There is a copy (LOAD"INLINE") of this program on the disk that comes with your S'more cartridge.

KEYWORD: **INLINE#**

ALTERNATE: **IN<SHIFT/L>**

FORMAT: INLINE# a,b$

(a is the device number)
to input)
(b$ is the variable to get)

MODE: PROGRAM

The INLINE# command works just like the INPUT# command from Commodore 64 Basic except that INLINE# accepts all characters up to the next carriage return (CHR$(13)). INLINE# does not recognize commas and colons as data separators like the Commodore 64 BASIC input command, it includes all characters as part of the string inputted. INLINE# will input string data to a maximum string length of 88 characters.

EXAMPLE

```
10 OPEN8,8,8,"FILE"
20 RECORD#8,32,22
30 INLINE#8,A$
```

NOTE: Using INLINE# with PRINT# USING and the RECORD# pointer can provide a very quick method of retrieving parts of records from relative files.

KEYWORD: **INFORM**

ALTERNATE: **IN<SHIFT/F>**

FORMAT: INFORM [ATa[,b]][;c$];LEN(d);e$

(a and b See AT)
(c$ any text string in quotes)
(d is the maximum length of the string variable to be input, up to a maximum of 8Ø characters)
(e$ is the variable to be input)

MODE: PROGRAM

INFORM is a special form of the input command. As with INLINE, INFORM accepts string variables only. But, the INFORM command actually limits the input data string to the number of characters specified in the LEN(d) parameter of the INFORM command. When the maximum length defined for the input string is reached, INFORM will recognize only the RETURN and DELETE keys, all other keys will be ignored.

The INFORM command also locks out the cursor direction keys, the comma, the quote mark, the clear/home key, the insert key, the shifted return key and all control key combinations. If any of these keys are pressed the INFORM statement simply ignores them.

Inform will accept up to 8Ø characters of input as determined by the LEN(d) parameter. The 8Ø characters might cover several lines on your screen. There might be information on some of those lines from previous program activity, but UNLIKE the INPUT command, INFORM will ignore any characters on the screen and only accept characters from the keyboard.

INFORM will allow you to construct very controlled screen input routines, to prevent many of the problems that cause programs to fail when users of the program enter unexpected data. See the MAIL program that is on the disk that comes with S'more for examples of some of the many uses and variations of this command.

EXAMPLE

```
1Ø INFORM LEN(1)
2Ø INFORM "NAME: ";LEN(12(;N$
3Ø INFORM AT9,2;LEN(15);A$
4Ø INFORM AT15,2;"ADDRESS: ";LEN(8Ø);AD$
```

KEYWORD: **GETKEY**

ALTERNATE: **GETK<SHIFT/E>**

FORMAT: GETKEY a$ [,b$ [,c$ [,more...]]]

(a$ is a one character)
(b$ is a one character)
(c$ is a one character)

MODE: PROGRAM

The GETKEY command waits for a key to be pressed and inputs the character. GETKEY also allows you to GET several characters with one command. So, the GETKEY command can be used to replace several lines of code normally used to GET single characters.

Commodore 64 Basic:

```
  10 GET A$ : IF A$ = "" THEN GOTO 10
```

S'more Basic

```
  10 GETKEY A$
```

Commodore 64 Basic:

```
  10 POKE 198,0 : WAIT 198,1 : GET A$
  20 POKE 198,0 : WAIT 198,1 : GET B$
```

S'more Basic

```
  10 GETKEY A$,B$
```

PART SIX

TO ERR IS HUMAN

We all make mistakes, sometimes it is even to our advantage to do things wrong. (I hope you realize that statement refers to programming errors not the morality of living.) As we are all well aware, when your Commodore 64 finds an error condition it halts program operation. This may or may not be what the programmer intended or indeed would prefer to happen.

For instance, what if you had a program that figured averages for sales prices of various items that you sold. But yesterday you didn't sell any of one of the items. Your program would have to divide nothing by nothing and would halt operation by saying **DIVIDE BY ZERO ERROR.** Wouldn't it be nice to say "Ooops, I'm sorry machine, I know you can't do that. But, would you please continue with my program anyway so I can find out what the rest of the prices were?"

By using the error handling routines provided by S'more Basic, you can prevent your programs from coming to an untimely and undesired crashing stop.

KEYWORD: **TRAP**

ALTERNATE: **T<SHIFT/R>**

FORMAT: TRAP a

(a is any program line number)

MODE: PROGRAM

TRAP is the command used to tell your computer not to stop the program and display the error message if an error occurs while running the program. Instead of stopping program execution TRAP causes the computer to go to the program line number listed in the TRAP command and continue program execution.

The TRAP function is turned on when your computer executes the TRAP command during program execution. TRAP with no line number following it will turn off the TRAP function. You may have several different error handling routines in your programs, but only one TRAP may be active at a time. If, for example you had error handling routines at lines 1000 and 2000. TRAP 1000 would send all errors to line 1000. If later in the program you had TRAP 2000, that command would supersede the previous TRAP command and send all errors to line 2000. And, again, TRAP with no line number listed would disable the TRAP function.

Most commonly the TRAP command is used to recover from errors caused by peripheral I/O devices (ie: DEVICE NOT PRESENT ERROR) and to prevent program crashes caused by bad data inputs (ie: OVERFLOW, ILLEGAL QUANTITY, STRING TOO LONG, etc.) The TRAP command can also be very useful during programming and de-bugging. You can even provide your own customized error messages.

While TRAP will allow you to produce CRASH-PROOF programs, don't let it become a crutch for bad programming. The purpose of TRAP is to divert the program, when an error occurs, to a routine that takes, or prompts the user to take, corrective action to fix the problem. Most errors must be corrected before program execution can resume. Some errors can be ignored, but most of the time data loss or program mis-function will happen if the problem is not fixed. - That's why they are called ERROR HANDLING ROUTINES.

WARNING: If a TRAP command refers to a non-existent line number the program will not notice it until an error occurs. When the error occurs, program execution will halt and the error message "UNEDF'D STATEMENT" will be shown, no matter what the actual error was. This can be a *#%?* to catch, so be careful when assigning TRAP line numbers.

NOTE: An error in an error handling routine can't be TRAPped, so double check your error handling routines.

KEYWORD: **ER**

ALTERNATE: **NONE**

FORMAT: ER

KEYWORD: **EL**

ALTERNATE: **NONE**

FORMAT: EL

MODE: DIRECT and PROGRAM

ER and EL are reserved variables just like TI, TI$, DS, DS$ and ST. When your Commodore 64 encounters an error condition it assigns ER a numeric value that corresponds to the type of error found. (See ERR$(er) on the next page for meanings of the values returned by ER.) If there has been no error encountered ER will contain a value of -1. After an error, ER is reset to -1 after execution of the next successful program statement.

If an error has occurred, EL will be set equal to the line number that the error occurred in. If no error has happened, EL will equal 65535.

KEYWORD: **ERR$**

ALTERNATE: **NONE**

FORMAT: ERR$(er)

(er is the error number - See ER)

MODE: DIRECT and PROGRAM

The ERR$(er) command is used to find out what an error number as returned by the ER command means. (There is a chart in the appendix that lists the error codes as well.) ERR$(er) will be set equal to the error message that corresponds to the error condition. You can then print the error message. Or ERR$(x) can be used to print any desired error message any place in your program that you might like to include it by putting the desired error number inside of the parentheses.

EXAMPLE

```
PRINT ERR$(16)  <RETURN>
```

RESULT:

```
OUT OF MEMORY
```

WARNING: Attempting to print an ERR$(-1) (ER value when no error exists) will cause an ILLEGAL QUANTITY ERROR.

KEYWORD: **RESUME**

ALTERNATE: **RES<SHIFT/U>**

FORMAT: RESUME [NEXT]

MODE: PROGRAM

RESUME is the command that returns your program execution to where it was when the error occurred. After TRAP sends your program to your error handling routine (and hopefully you fix the problem), RESUME does exactly what it says, it resumes program execution right where it left off. RESUME returns program to the same program statement that caused the error condition. If your error handling routine didn't take the proper corrective action and the problem still exists, TRAP will send it again to the error handling routine, and RESUME will send it back, creating a infinite loop, and therefore a program crash. This is not the best situation, so S'more gives you the RESUME NEXT option.

RESUME NEXT directs program execution to the first program statement after the statement in which the error occurred. So using RESUME NEXT, even if an error condition can't be corrected, the program statement causing it can be bypassed. The following are some examples of possible error handling routines:

```
 1Ø TRAP 5ØØ
 2Ø OPEN 4,4 : PRINT#4,"PRINTER OK" :
    PRINT"PRINTER OK"
 3Ø TRAP 6ØØ
 4Ø OPEN 15,8,15,"IØ" : CLOSE15 : PRINT
    "DISK DRIVE OK"
 5Ø TRAP 7ØØ
 6Ø PRONT"ERROR"
 7Ø PRINT AD$(999)
 8Ø BORDER=5ØØ
 9Ø OPEN4,4 : PRINT#4,"ALL DONE"
399 END
4ØØ TN=INK : INK=PAPER : CLS : PRINT"LIST"
    EL":GOTO 44Ø:" : INK=TN
41Ø PRINT AT 7,Ø"* "ERR$(ER)" ERROR IN LINE"
    EL : PRINT : PRINT X$ : PRINT
42Ø PRINT"TRY AGAIN OR SKIP IT (T/S)"
43Ø POKE 198,2 : POKE 631,19 : POKE 632,13
    END
44Ø GETKEY A$ : IF A$="S" THEN RESUME NEXT
45Ø RESUME
5ØØ X$="WHERE'S YOUR PRINTER DUMMY?":
    GOTO 4ØØ
6ØØ X$="YOU FORGOT TO TURN YOUR DRIVE ON.":
    GOTO 4ØØ
7ØØ X$="O.K. WHAT DID YOU DO NOW, STUPID?":
    GOTO 4ØØ
```

NOTE: This program is on the disk that comes with S'more, LOAD"ERROR".

PART SEVEN

DO...LOOP DIDDY DIDDY DUM DIDDY DUM

Programming purists have scorned basic in general and Commodore 64 Basic in particular for its lack of structured programming ability. Well, here it is... The Do...Loop.

Loops are the heart of higher level structured programming environments, and offer several advantages over normal basic FOR...NEXT loops and GOTO statements. S'more incorporates fully implemented DO...LOOP structure into basic, allowing you to choose to use and learn about structured programming.

KEYWORD: **DO**

ALTERNATE: **NONE**

FORMAT: DO

KEYWORD: **LOOP**

ALTERNATE: **LO<SHIFT/O>**

FORMAT: LOOP

MODE: DIRECT and PROGRAM

DO and LOOP mark the starting point and ending point of a DO...LOOP. A DO...LOOP simply loops from its ending point back to its starting point until it is instructed to do differently.

EXAMPLE # 1

```
10 CLS
20 DO
30  PRINT "HI"
40 LOOP
```

EXAMPLE # 2

```
10 DO
20  BORDER=BORDER+1:IFBORDER=16THENBORDER=0
30 LOOP
```

KEYWORD: **WHILE**

ALTERNATE: **W<SHIFT/H>**

FORMAT: WHILE (logical argument)

KEYWORD: **UNTIL**

ALTERNATE: **U<SHIFT/N>**

FORMAT: UNTIL (logical argument)

MODE: DIRECT and PROGRAM

These commands can be added to either the DO or LOOP commands as conditions to abort the looping process. For example if we said DO UNTIL A=3, our do loop would continue looping until a was equal to three at UNTIL statement.

EXAMPLE

```
10 DO UNTIL A=3
20  A=A+1 : PRINT A
30 LOOP
40 PRINT "DONE"
```

When A became equal to 3 in line 10, the program execution would jump to the next program statement after the LOOP command (ie: LINE 40).

EXAMPLE

```
10 DO
20  A=A+1 : PRINT A
30 LOOP UNTIL A=3
40 PRINT "DONE"
```

When A became equal to 3 in line 30, the program execution would again jump to the next program statement after the LOOP command (ie: LINE 40)

The WHILE command is the exact opposite of the UNTIL command, program execution is transferred to the line following the loop command only when the condition is **NOT** met.

EXAMPLE

```
10 DO WHILE A<>3
20  A=A+1 : PRINT A
30 LOOP
40 PRINT "DONE"
```

When A became equal to 3 in line 10, the program execution would jump to the next program statement after the LOOP command (ie: LINE 40). As with UNTIL, WHILE can also follow the LOOP command.

EXAMPLE

```
10 DO
20  A=A+1 : PRINT A
30 LOOP WHILE A<>3
40 PRINT "DONE"
```

In addition to these functions DO...Loops can be nested:

EXAMPLE

```
10 DO : CLS
20  DO UNITL BORDER=16
30   BORDER=BORDER+1
40    DO UNTIL PAPER=16
50     PAPER=PAPER+1
60    LOOP
70   PAPER=0
80  LOOP
90  BORDER=0 : PRINT "PRESS C TO CONTINUE":
    GETKEY A$
99 LOOP WHILE A$ = " "
```

That's how DO...LOOPs work.

```
10 PRINT"IS THIS FUN? (Y/N)"
20 DO:GETA$:LOOP UNTIL A$="Y"
30 PRINT"I THOUGHT SO."
```

KEYWORD: **EXIT**

ALTERNATE: **EX<SHIFT/I>**

FORMAT: EXIT

MODE: DIRECT and PROGRAM

THe EXIT command allows you to exit a DO...LOOP at any time the EXIT command is encountered. When the EXIT command is encountered, program execution drops to the program statement following the LOOP command.

EXAMPLE

```
10 CLS : BORDER=Ø : PAPER=Ø : DO
20  DO : BORDER=BORDER+1
30   IF BORDER=15 THEN EXIT
40  LOOP : BORDER=Ø
50 PAPER=PAPER+1 : IF PAPER=16 THEN EXIT
60 LOOP : NORM : END
```

PART EIGHT

STRING THINGS AND OTHER THINGS

String manipulation can always be a difficult task. No matter how many commands and tools your computer offers for string manipulation, there always seems to be one more that you could use to accomplish a task more easily.

S'more Basic gives you two additional string manipulation tools. It adds one new string handling command, and provides a new way to use an existing command. These two new functions should make some string manipulations easier.

S'more also provides two other existing Commodore 64 Basic commands with additional (and badly needed) functions. At last you will have IF...THEN...ELSE capability on your Commodore 64.

KEYWORD: **INSTR**

ALTERNATE: **IN<SHIFT/S>**

FORMAT: INSTR a$,b$ [,c]

(a$ is the string to be searched)
(b$ is the sub-string to find)
(c is the starting position)

MODE: DIRECT and PROGRAM

INSTR will return a numeric value equal to the starting position of sub-string (b$) within a given string (a$). If the sub-string is not found INSTR will return a value of ZERO (Ø). INSTR will allow you to quickly test for the presence of a character or group of characters within a string. The additional parameter, starting position (c), allows you to declare the character position within the main string to start searching from. This extra feature will let you check for multiple occurrences of a character or group of characters within a string. The value supplied by the INSTR command allows you to further process the string using the MID$, LEFT$ and RIGHT$ commands.

EXAMPLE:

```
    PRINT INSTR ("ABCDEF","C")  <RETURN>
```

RESULT: 3

```
    PRINT INSTR ("ABCDEF","G")  <RETURN>

RESULT: Ø

   PRINT INSTR ("ABCDEF","C",4)  <RETURN>

RESULT: Ø

    PRINT INSTR ("ABCDEF","DE")  <RETURN>

RESULT: 4

    PRINT INSTR ("ABCDEF","DF")  <RETURN>

RESULT: 5
```

```
 1Ø A$="12,345,678.9Ø" : B$=A$
 2Ø REM * * * * * CHECK DECIMAL POINTS *
 3Ø DP=INSTR(A$,".")
 4Ø IF DP>Ø THEN IF INSTR (A$,DP+1) <> Ø
   THEN PRINT "TOO MANY DECIMAL POINTS"
 5Ø REM * * *  PERFORM VAL(X) FUNCTION *
 6Ø CP=INSTR(B$,","):IF CP=Ø THEN GOTO 1 ØØ
 7Ø B$=LEFT$(B$,CP-1)+RIGHT$(B$,LEN(B$)-CP)
 8Ø GOTO 6Ø
1ØØ PRINT A$
11Ø PRINT B$
12Ø PRINT VAL(B$)
```

RESULT:

```
12,345,678.9Ø
12345678.9Ø
 12345678.9Ø
```

KEYWORD: **MID$**

ALTERNATE: **M<SHIFT/I>**

FORMAT: MID$(a$,b [,c]) = d$

(a$ is any existing string)
(b is the starting position)
(c is the number of characters)
(d$ is the string to substitute)
DEFAULT: c=LEN(d$)

MODE: DIRECT and PROGRAM

IMPORTANT NOTE: The normal function of the MID$ command as in Commodore 64 Basic is still operative. The format shown above is a second function for this command.

The new function added to the MID$ command by S'more Basic will allow you to substitute characters within a string. You are limited to replacement of equal quantities of characters only. You can't replace seven characters with five or five with seven.

The starting position parameter allows you to set the character position (from the right) of the first character to be replaced. The number of characters to be replaced will be equal to the number of characters in the replacement string unless it is set with the optional parameter.

EXAMPLES:

```
10 A$="1234567890"
20 MID$(A$,3)="###" : PRINT A$

RESULT:  12###67890


10 A$="1234567890" : B$="#####"
20 MID$(A$,5,2)=B$ : PRINT A$

RESULT:  1234##7890


10 A$="1234567890"
20 MID$(A$,LEN(A$)-3)=MID$(A$,LEN(A$)-4,3)
30 PRINT A$

RESULT:  1234566780


10 A$="APRIL 15, 1985" : B$="1986"
20 MID$(A$,INSTR(A$,",")+2)=B$ : PRINT A$

RESULT:  APRIL 15, 1986
```

KEYWORD: **ELSE**

ALTERNATE: **E<SHIFT/L>**

FORMAT: IF...THEN...[:ELSE]

MODE: PROGRAM

With Commodore 64 Basic you learned how the IF...THEN conditional statement works. The ELSE command lets you specify an action to be taken if the conditional statement in the IF...THEN command is not true.

EXAMPLE # 1

```
100 IF A=B THEN GOTO 400 : ELSE GOTO 500
```

EXAMPLE # 2

```
100 IF A=B THEN X=4 : GOSUB 200 : ELSE X=5 :
    GOSUB 300
```

EXAMPLE # 3

```
100 IF A=B THEN X=8 : ELSE IF A=C THEN X=9 :
    ELSE IF A=D THEN X=10 : ELSE X=11
```

KEYWORD: **RESTORE**

ALTERNATE: **RE<SHIFT/S>**

FORMAT: RESTORE [a]

(a is any program line number)
Default a=Ø

MODE: DIRECT and PROGRAM

S'more Basic's version of the RESTORE command works just like the Commodore 64 Basic version except S'more allows you to specify the line number of your program that that will be read as the next data statement.

EXAMPLE

```
 1Ø READ A$ : PRINT A$
 2Ø RESTORE 12Ø
 3Ø READ A$ : PRINT A$
 4Ø RESTORE 11Ø
 5Ø READ A$ : PRINT A$
1ØØ DATA BILL
11Ø DATA MARY
12Ø DATA MIKE
```

RESULT:

```
BILL
MIKE
MARY
```

PART NINE

PEEKS AND POKES

When using S'more Basic you should never have to use the PEEK or POKE commands again. All of the common programming functions that used to require peeking and poking and remembering special numbers for particular memory locations have been replaced with a simple-to-remember, easy-to-use set of commands.

When the S'more cartridge reconfigures your computer's memory to allow full use of the RAM provided, it changes some of the locations that you may have used to accomplish some tasks. Other areas of memory that used to be open and unused are now used by S'more and peeking and poking these locations will cause programs to fail to operate properly.

This section of the S'more manual will be devoted to discussing how to do the things that you used to have to use the PEEK and POKE commands to accomplish. You will also want to consult the MEMORY MAP and MACHINE LANGUAGE WITH S'MORE BASIC sections in the appendix for more information.

S'MORE BASIC RESERVED VARIABLES

Within S'more Basic, memory locations that are needed to accomplish certain functions are treated as RESERVED VARIABLES. Commodore 64 Basic uses this same system for the TI and TI$ RESERVED VARIABLES used to refer to the clock/time function. The following is a list of S'more's RESERVED VARIABLES and the corresponding Commodore 64 Basic memory (PEEK/POKE) locations.

RESERVED VARIABLE	MIN/MAX SUBSCRIPT	MIN/MAX VALUE	MEMORY LOCATION
CIA(x)	x=Øto15	Ø-255	5632Ø-56336
CIA(x)	x=16to32	Ø-255	56576-56591
COL(x)	x=Øto999	Ø-255	55296-56295
SID(x)	x=Øto28	Ø-255	54272-543ØØ
VIC(x)	x=Øto46	Ø-255	53248-53294
VID(x)	x=Øto999	Ø-255	1Ø24-2Ø23
BORDER	-----	Ø-15	5328Ø
PAPER	-----	Ø-15	53281
STOP	-----	ON-OFF	788/789 8Ø8/8Ø9
REPEAT	-----	ON-OFF	65Ø

Using each of these RESERVED VARIABLES you can do the same thing as if you peeked or poked the location, for example:
COMMODORE 64 BASIC:

```
          PRINT PEEK(53267)  <RETURN>
```

S'more Basic:

```
            PRINT VIC(19)  <RETURN>
```

Both of these statements will print the value of the VIC chip location (register) that refers to the light pen x position.

COMMODORE 64 BASIC:

```
           POKE (54296),15  <RETURN>
```

S'more Basic:

```
             SID(24)=15  <RETURN>
```

Both of these statements will set the value of the SID chip location (register) that refers to the volume level to 15.

For the purpose of converting Commodore 64 Basic programs to run under S'more Basic, all you need to do is refer to the chart of location conversions at the end of this section and make the suggested changes. If however you are interested in programing the special functions that are available in your Commodore 64 computer, you should consult the Commodore Programmers Reference Guide for a more complete explanation of what these memory locations are used for.

Here is a brief explanation of the functions controlled by the S'more RESERVED VARIABLES:

CIA(x) refers to the Complex Interface Adapter chips. You should think of the CIA(x) RESERVED VARIABLE as an array variable that has been dimensioned (DIM CIA(31)) to hold 32 subscripted variables. There are two of CIA chips, each has 16 ports or registers. The first CIA chip (CIA#1) is addressed using the first 16 (Ø to 15) available subscripts of the CIA(x) variable, and the second CIA chip (CIA#2) is addressed using the last 16 (16 to 31) subscripts.

SID(x) refers to the Sound Interface Device chip. You should think of the SID(x) RESERVED VARIABLE as an array variable that has been dimensioned (DIM SID(28)) to hold 29 subscripted variables. Each of these 29 variables will refer to one of the 29 registers on the SID chip.

VIC(x) refers to the Video Interface Controller chip. You should think of the VIC(x) RESERVED VARIABLE as an array variable that has been dimensioned (DIM VIC(46)) to hold 47 subscripted variables. Each of these 47 variables will refer to one of the 47 registers on the SID chip.

COL(x) refers to the Color RAM area. This is the area in memory where the computer stores the color information for the screen. There are 1ØØØ Color RAM locations that match the 1ØØØ possible character locations on your video screen. You should think of the COL(x) RESERVED VARIABLE as an array variable that has been dimensioned (DIM COL(999)) to hold 1ØØØ subscripted variables. Each of these 1ØØØ variables will refer to one of the 1ØØØ Color RAM locations.

VID(x) refers to the Video RAM area. This is the area in memory where the computer stores the character information for the screen. There are 1ØØØ Video RAM locations that match the 1ØØØ possible character locations on your video screen. You should think of the VID(x) RESERVED VARIABLE as an array variable that has been dimensioned (DIM VID(999)) to hold 1ØØØ subscripted variables. Each of these 1ØØØ variables will refer to one of the 1ØØØ Video RAM locations.

BORDER, PAPER, STOP and REPEAT are discussed individually, refer to each item for an individual explanation of its functions.

Again, due to the changing around of the memory configuration done by S'more, peeks and pokes to specific memory locations simply will not work. For example, all of the RAM area from location ØCØØ HEX (that is 3Ø72 decimal) up to FEFF Hex (that is 65279 decimal) is free for use as basic programming space. So a POKE to location 5328Ø decimal (which was used under Commodore 64 Basic to change the border color) would have no effect. The reason for this is because S'more has moved the Video Interface Controller chip out of the way of basic programming, so poking 5328Ø just pokes a value into location 5328Ø of normal RAM. This movement of things is almost total, so if you want to PEEK or POKE around, consult the memory map in the appendix to see what is where.

NOTE: Also see MACHINE LANGUAGE WITH S'MORE BASIC in the appendix if you are having problems. M/L subroutines will run with S'more, but must be converted correctly.

LOCATION CONVERSION CHART

LOCATION	CONVERSION PROCEDURE
Ø-767	Basic Work Area/Pointers etc. PROBABLY O.K. *
768-819	These are the KERNAL VECTORS and are left untouched. DO NOT CHANGE THESE - O.K.
82Ø-827	Add 2244 to the number
828-1Ø19	Add 516 to the number
1Ø2Ø-1Ø23	Add 2Ø4Ø to the number
1Ø24-2Ø23	Change to VID(Ø-999) or Add 2Ø48 to the number
2Ø24-2Ø47	Add 2Ø48 to the number
2Ø48-4Ø959	Add 2Ø48 to the number
4Ø96Ø-53247	NOT NORMALLY USED **
53248-53294	Change to VIC(Ø-48)
53295-54271	NOT NORMALLY USED **
54272-5423Ø	Change to SID(Ø-28)
54231-55295	NOT NORMALLY USED **
55296-56295	Change to COL(Ø-999)
56295-56319	NOT NORMALLY USED **
5632Ø-56335	Change to SID(Ø-15)
56336-56575	NOT NORMALLY USED **
56576-56591	Change to SID(16-31)
56592-65279	NOT NORMALLY USED **
6528Ø-65535	DO NOT USE - RESERVED AREA

* Most of the items in these areas are unchanged, but refer to the memory map if problems arise.

** These locations are not defined by Commodore 64 Basic.

NOTES

APPENDIX
****

MACHINE LANGUAGE PROGRAMMING

Generally machine language programming will fall into one of two categories. This section will deal with each category separately. The first category will consist of programs that are written completely in machine language. The second category will consist of hybrid programs that are written in a combination of Basic and machine language. Machine language programs and hybrid Basic/machine language programs can both be used with S'more Basic as long as they are written correctly. Most existing machine language programs and routines will need to be modified to some extent. In cases where the programs or routines access specific ROM locations (bypassing the standardized KERNAL vectors), these routines will have to be completely rewritten. Due to the revision of the memory allocation, most machine language programs and routines will need to be relocated to a safe area within S'more Basic's new memory confines.

1ØØ% MACHINE LANGUAGE PROGRAMS

Converting existing machine language programs for use with S'more Basic may well be a monumental task, depending on the complexity of the program. More than likely, this job should be attempted by the author of the original code. The guidelines for converting machine language programs will be the same as those for writing original code and machine language sub-routines.

If you want to write new machine language programs to run in S'more's expanded 6ØK available memory space, there are a couple of possibilities. First, consider writing the program in S'more Basic format and then using the S'more Basic Compiler which will be available in the fourth quarter of 1985 to compile your basic program into machine language code. Using the S'more Basic Compiler does not require any machine language programming knowledge or experience on the part of the programmer. The operation of the S'more Basic Compiler is automatic; you just provide a running, error-free version of a basic program, and the S'more Basic Compiler will provide you with a ready-t- run compiled machine language program. A compiled S'more Basic program will give you full advantage of S'more's 6ØK work space and the increased speed that a machine language program provides. Short programs will require more memory after they have been compiled, due to the overhead required by the compiler. Longer compiled programs, however, will probably require less memory than their S'more Basic versions.

The other possibility, of course, is to generate machine language code directly, or by using a machine language monitor or assembler. The programmer will need to be familiar with 65Ø2 machine language programming, as well as the operations of the Commodore 64 I/O, Video, Sound and Screen operations. Any standard 65Ø2 machine language code assembler will be useful, but assemblers that utilize specific locations in the Commodore 64 ROMs will not work. Most machine language monitor programs will run in the S'more Basic environment. Included on the disk that comes with your S'more cartridge is the public domain machine language monitor program (RUN"SMON") that is used by CARDCO's R&D staff to prepare machine language programs and routines. All machine language programs should be written to conform with the machine language programming guidelines at the end of this section.

MACHINE LANGUAGE SUB-ROUTINES

You may want to add machine language sub-routines to your S'more Basic programs to increase the speed of an operation, or to make an operation perform tasks that are not easily done within normal basic. The screen dump program (RUN"SDUMP") on the disk that is provided with your S'more cartridge is an example of such a sub-routine.

Creating machine language sub-routines for use with S'more Basic programs will require knowledge of 65Ø2 machine language programming, and must conform with the guidelines for machine language programming provided at the end of this section. S'more Basic's expanded memory provides ample room for the addition of machine language routines and allows the use of the same commands (PEEK, POKE, SYS and USR) that standard Commodore 64 Basic provides. The major differences between writing routines for S'more Basic and Commodore 64 Basic are the location of flags, pointers and data caused by the variation in memory maps between the two basics, the location of the routine (where in memory the routine is located) and the amount of dependence on existing ROM routines allowed.

As with all machine language programming a good machine language monitor like the public domain machine language monitor program (RUN"SMON") that is included on the disk that comes with your S'more cartridge and an assembler program can be valuable tools and make machine language programming easier.

GUIDELINES FOR MACHINE LANGUAGE PROGRAMS

1. PROGRAM LOCATION

The memory configuration used by S'more Basic is different than the one used by Commodore 64 Basic. The locations available for safe storage of machine language programs and routines will therefore be different.

Any stand alone machine language program (a program that is not part of, or to be used with a basic program) can use and be located anywhere within all of the 61,183 bytes of the available RAM from $1ØØØ to $FEFF. If the program does not allow cassette I/O, the 2Ø1 bytes reserved for the cassette buffer (located from $Ø54Ø to $Ø5FF) may be utilized. Additionally, if the program does not allow RS-232 I/O, 512 more bytes (located from $Ø6ØØ to Ø7FF) are available from the RS-232 input and output buffers.

If your machine language routines are to be used with basic programs you must be careful to protect them from being destroyed by basic's use of memory for variable and program storage. You could hide your routines from basic in the cassette buffer (located from $Ø54Ø to $Ø5FF) or RS-232 buffers (located from $Ø6ØØ to Ø7FF) if you are sure that cassette and/or RS-232 functions will never be used. But, a better method is to tell basic where you are putting your routine, and tell basic not to use that area of RAM. This can be done by locating your program at the very top of the available RAM area (up to $FEFF) and resetting the pointer that tells basic the highest location that it can use. Resetting the end of basic pointer, located at $ØØ37-ØØ38, will reserve the space between the new pointer value and the actual top of memory which is $FEFF for your routines and protect them from any basic operations that would normally use that area of RAM.

An even better method would include a program that reads the existing end of basic pointers and relocates the program and resets the pointers to accommodate the new program location. This would allow you to load several machine language routines, each would locate itself just below the previous routine, preventing conflicts and allowing simultaneous operation of several routines.

2. S'MORE BASIC MEMORY MAP

The S'more Basic MEMORY MAP is listed right after this section. The MEMORY MAP will provide you with all of the descriptions and memory locations you will need to use to interface machine language routines with basic programs. Review the

MEMORY MAP carefully before you address any areas of RAM below $1ØØØ. Most of the pointer, flag, register and vector locations are the same as those in Commodore 64 Basic, however, some are not. Using the MEMORY MAP and checking critical locations used by your routines against the standard Commodore 64 Basic memory map will highlight any differences. All of the necessary flags, pointers, etc. are available, but you may have to look a bit to find them.

NOTE: In some cases the wording describing a S'more Basic memory location may differ from the wording describing the corresponding Commodore 64 Basic memory location. This does not necessarily mean that the two locations are different in function.

3. ROM CALLS, KERNAL CALLS AND VECTORS

When the S'more Basic cartridge is installed the Commodore 64 Basic ROMs are not available and any calls or jumps to these locations in ROM will actually wind up in the active basic RAM area under S'more Basic. The existing KERNAL calls used by Commodore Basic have been retained with only minor changes, however it it mandatory that you use the KERNAL JUMP VECTORS provided in the S'more Basic MEMORY MAP to access these KERNALs. Commodore has provided a standardized set of KERNAL vector locations which S'more Basic observes as strictly as possible. If these vectors are used, your programs will run on this as well as future versions of S'more Basic.

4. RAM/ROM BANKING

As long as the KERNAL vectors referred to in the previous paragraphs are observed, the RAM/ROM banking will be transparent to the programmer as well as the end user, and the programmer need not make any allowances for the banking procedures. If however you need to address the banking function, the S'more Basic MEMORY MAP will provide the necessary locations for the banking stacks and registers. The operation of the banking system is similar to that of the Commodore +4 computer and is detailed in the documentation available from Commodore for that machine.

SPECIAL NOTE: Be sure that you are aware that two of the favorite places to put machine language routines have been changed. The cassette buffer has been moved from $Ø33C to $Ø54Ø. The area from $CØØØ to SCFFF can no longer be used safely because it is now part of the active basic RAM area.

S'more Memory Map
****

HEX ADDRESS	DESCRIPTION
0000	6510 ON-CHIP DATA DIRECTION REGISTER
0001	6510 ON-CHIP 5-BIT I/O-REGISTER
0002	SEARCH TOKEN FOR BASIC-STACK
0003-0004	JUMP VECTOR: FLOATING/INTEGER CONVERSION
0005-0006	JUMP VECTOR: INTEGER/FLOATING CONVERSION
0007	SEARCH CHARACTER (':', $00 FOR DATA)
0008	FLAG : SCAN FOR QUOTE AT END OF STRING
0009	SCREEN COLUMN FROM LAST TAB
000A	FLAG : 0=LOAD, 1=VERIFY
000B	INPUT BUFFER POINTER / NUMBER OF SUBSCRIPTS
000C	FLAG : ROUTINE TO DIMENSION DEFAULT ARRAY
000D	DATA TYPE: $FF = STRING, $00 = NUMERIC
000E	DATA TYPE: $80 = INTEGER, $00 = real
000F	FLAG FOR GARBAGE COLLECTION/QUOTE IN LIST/SCAN FOR DATA
0010	UTILITY FLAG FOR USER FUNCTION CALL
0011	FLAG : $00=INPUT/INFORM, $40=GET, $98=READ
0012	FLAG : TANGENT SIGN/COMPARISON RESULTS
0013	FLAG : TEMPORARY OUTPUT CHANNEL
0014-0015	TEMPORARY 16-BIT INTEGER PARAMETER (FOR LINE NUMBER, ETC.)
0016	POINTER TO ACTUAL ELEMENT IN STRING STACK
0017-0018	POINTER TO LAST TEMPORARY STRING ADDRESS
0019-0021	STACK FOR TEMPORARY STRINGS
0022-0025	UTILITY POINTER AREA
0026-002A	MULTIPLY: FLOATING POINT PRODUCT
002B-002C	POINTER: START BASIC PROGRAM
002D-002E	POINTER: END BASIC PROGRAM/START OF VARIABLES
002F-0030	POINTER: START OF ARRAYS
0031-0032	POINTER: END OF ARRAYS
0033-0034	POINTER: BOTTOM OF STRING STORAGE
0035-0036	UTILITY POINTER FOR STRINGS
0037-0038	POINTER: END BASIC RAM

HEX ADDRESS	DESCRIPTION
0039-003A	CURRENT BASIC COLUMN LINE NUMBER
003B-003C	CURRENT PROGRAM POINTER (FOR CHARGET ROUTINE)
003D-003E	UTILITY POINTER FOR SEARCH IN STACK
003F-0040	CURRENT DATA LINE NUMBER
0041-0042	POINTER: ADDRESS FOR CURRENT DATA ITEM
0043-0044	VECTOR: FOR INPUT ROUTINE
0045-0046	POINTER: TO NAME OF CURRENT BASIC VARIABLE
0047-0048	POINTER TO CURRENT BASIC VARIABLE DATA
0049-004A	POINTER: INDEX VARIABLE FOR/NEXT LOOP
004B-004C	POINTER FOR COMPARISON OPERATIONS
004D	MASK FOR COMPARISON OPERATIONS
004E-004F	POINTER: FOR FN FUNCTION
0050-0052	UTILITY: STRING DESCRIPTIONS FOR VARIABLE MANAGEMENT
0053	FLAG FOR HELP (LINE NUMBER)
0054	6510 JUMP-COMMAND FOR FUNCTIONS
0055-0056	JUMP VECTOR TO DECLARE FUNCTIONS
0057-005B	REGISTER FOR ARITHMETIC, ACCUMULATOR #3
005C-0060	REGISTER FOR ARITHMETIC, ACCUMULATOR #4
0061-0065	FLOATING POINT ACCUMULATOR #1
0066	SIGN FOR FLOATING POINT ACCUMULATOR #1
0067	POINTER: EVALUATION OF CONSTANT VALUE
0068	OVERFLOW FOR FLOATING POINT ACCUMULATOR #1
006A-006E	FLOATING POINT ACCUMULATOR #2
006F	RESULT : SIGN COMPARISON OF ACCUM #1/ACCUM #2
0070	FLOATING POINT ACCUMULATOR #1 - LOW ORDER
0071-0072	UTILITY POINTER
0073	ACCUMULATOR FOR SYS-COMMAND (NORMAL: 030C)
0074	X-REGISTER FOR SYS-COMMAND (NORMAL: 030D)
0075	Y-REGISTER FOR SYS-COMMAND (NORMAL: 030E)
0076	PROCESSOR STATUS FOR SYS-COMMAND (NORMAL: 030F)
0077	LENGTH OF DS$
0078-0079	POINTER TO DS$
007A-007B	UTILITY REGISTER FOR BELOW ROM-TECHNOLOGY

HEX ADDRESS	DESCRIPTION
007C-007D	STACK POINTER FOR BASIC STACK
007E-007F	FREE
0080	FLAG FOR GETKEY
0081	FLAG FOR DIRECT/PROGRAM MODE
0082-008A	FREE
008B-008F	LAST RND SEED VALUE
0090	STATUS WORD ST
0091	FLAG FOR STOP KEY/SCAN
0092	TIME CONSTANT FOR TAPE
0093	0=LOAD, 1=VERIFY
0094	FLAG FOR BYTE IN SERIAL BUS OUTPUT BUFFER
0095	SERIAL BUS OUTPUT BUFFER (FOR EOI)
0097	TEMPORARY STORAGE FOR REGISTER
0098	NUMBER OF OPEN FILES
0099	ACTIVE (DEFAULT) INPUT DEVICE NUMBER
009A	ACTIVE (DEFAULT) OUTPUT DEVICE NUMBER
009D	FLAG : $80=DIRECT MODE, $00=PROGRAM MODE, $C0=MONITOR
009E	TAPE ERROR LOG - PASS 1
009F	TAPE ERROR LOG - PASS 2
00A0-00A2	CLOCK (FOR TI, TI$)
00A3-00A4	BIT COUNTER FOR SERIAL OUTPUT
00A5	UTILITY FLAG FOR SERIAL OPERATIONS
00A6-00AD	UTILITY WORK AREA
00AE-00AF	POINTER: END OF PROGRAM (TAPE)
00B0-00B6	UTILITY WORK AREA
00B7	LENGTH OF FILENAME
00B8	LOGICAL FILE NUMBER
00B9	SECONDARY ADDRESS
00BA	DEVICE NUMBER
00BB-00BC	POINTER: CURRENT FILENAME
00C0	FLAG FOR TAPE MOTOR
00C1-00C2	START ADDRESS FOR INPUT/OUTPUT
00C3-00C4	END ADDRESS FOR INPUT/OUTPUT

HEX ADDRESS	DESCRIPTION
00C5	CURRENT KEY PRESSED
00C6	NUMBER OF CHARACTERS IN KEYBOARD BUFFER
00C7	FLAG FOR REVERSE ON
00C8	END OF LOGICAL LINE FOR INPUT
00C9-00CA	CURSOR X,Y POSITION AT START OF INPUT
00CB	FLAG: SHIFTED CHARACTERS = 64
00CC	FLAG: CURSOR BLINK ON = 0
00CD	TIMER: CURSOR ON/OFF
00CE	CHARACTER UNDER CURSOR
00CF	FLAG: LAST CURSOR BLINK ON/OFF
00D0	FLAG: INPUT/GET FROM KEYBOARD
00D1-00D2	POINTER: CURRENT SCREEN LINE ADDRESS
00D3	CURSOR POSITION ON CURRENT LINE
00D4	FLAG: IN QUOTE MODE? NO = 0
00D5	CURRENT LINE LENGTH
00D6	CURRENT CURSOR LINE (REAL)
00D7	TEMPORARY STORAGE FOR DATA
00D8	NUMBER OF INSERTS
00D9-00F2	SCREEN LINK TABLE
00F3-00F4	POINTER: COLOR RAM LOCATION
00F5-00F6	VECTOR: KEYBOARD DECODER TABLE
00F7-00F8	POINTER: TO RS-232 INPUT BUFFER
00F9-00FA	POINTER: TO RS-232 OUTPUT BUFFER
00FB-00FE	NOT USED
00FF	REAL VS. ASCII MARKER
0100-010F	BUFFER FOR REAL VS. ASCII CONVERSION
0100-0127	BUFFER FOR FILENAME
0128-01FC	6510 SYSTEM STACK
01FD-01FF	RESERVED TEMPORARY DATA STORAGE
0200-0258	BASIC - INPUT BUFFER
0259-0262	KERNAL TABLE: LOGICAL FILE NUMBERS ACTIVE
0263-026C	KERNAL TABLE: DEVICE NUMBER FOR EACH FILE

HEX ADDRESS	DESCRIPTION
026D-0276	KERNAL TABLE: SECONDARY ADDRESS FOR EACH FILE
0277-0280	KEYBOARD BUFFER
0281-0282	POINTER: START BASIC RAM - BOTTOM OF MEMORY
0283-0284	POINTER: END BASIC RAM - TOP OF MEMORY
0285	FLAG: TIMEOUT-FLAG FOR PARALLEL IEEE488 BUS
0286	CURRENT CHARACTER COLOR CODE
0287	BACKGROUND COLOR UNDER CURSOR
0288	HIGH BYTE - TOP OF SCREEN RAM (PAGE)
0289	LENGTH OF KEYBOARD BUFFER
028A	FLAG: FOR REPEAT KEY(s) - 128 = REPEAT
028B	COUNTER FOR REPEAT SPEED
028C	COUNTER FOR REPEAT DELAY
028D	FLAG: FOR SHIFT/COMMODORE/CONTROL KEYS
028E	PREVIOUS SHIFT/COMMODORE/CONTROL FLAG
028F-0290	VECTOR: KEYBOARD DECODER TABLE
0291	FLAG: SHIFT/CBM DISABLE
0292	FLAG: AUTO SCROLL DOWN (0=ON)
0293-029E	RS-232 EMULATOR
029F-02A0	STORAGE FOR IRQ VECTOR DURING TAPE I/O
02A1	CIA 2 NMI FLAG
02A2	CIA 1 TIMER
02A3	CIA 1 INTERRUPT FLAG
02A4	CIA 1 FLAG FOR TIMER OVER/UNDER TIME
02A5	STORAGE FOR SCREEN LINE INDEX
02A6	FLAG: PAL = 1 / NTSC = 0
02A7	CONSTANT FOR SYSTEM VARIABLE NULL VALUE
02A8-02A9	RESERVED
02AA	FLAG: FOR BANK SELECT IRQ
02AB	FLAG FOR KEY ON/OFF
02AC	POINTER: LAST CHARACTER OUTPUTTED
02AD	RELATIVE POINTER: FUNCTION KEY TEXT
02AE	POINTER: LENGTH OF FUNCTION KEY TEXT
02AF	ALLOWABLE INPUT LENGTH FOR INFORM

HEX ADDRESS	DESCRIPTION
02B0	FLAG: MERGE ACTIVE
02B1	FLAG: HELP ON/OFF
02B2	FLAG: PRINT AT ACTIVE
02B3	TEMPORARY STORAGE FOR PRINT AT
02B4	START STORAGE FOR PRINT AT / TEMP FOR CATALOG
02B5	LINE STORAGE FOR PRINT AT / TEMP FOR CATALOG
02B6-02B7	LINE NUMBER FOR CONTINUE
02B8-02B9	ADDRESS FOR CONTINUE
02BA	PRINT USING FILLER CHARACTER
02BB	PRINT USING GROUP MARKER CHARACTER
02BC	PRINT USING DECIMAL POINT CHARACTER
02BD	PRINT USING DOLLAR SIGN CHARACTER
02BE-02C1	TEMPORARY POINTER TO FORMAT STRING
02C2	LAST ERROR NUMBER
02C3-02C4	LINE NUMBER OF LAST ERROR
02C5-02C6	LINE NUMBER OF ERROR HANDLING ROUTINE (TRAP)
02C7	TEMPORARY STORAGE FOR TRAPPED LINE
02C8-02C9	ADDRESS OF LAST ERROR
02CA	STACK POINTER: RESUME FROM ERROR
02CB-02CC	TEMPORARY STORAGE OF ACTUAL ADDRESS FOR DO
02CD-02CE	TEMPORARY STORAGE OF ACTUAL LINE NUMBER OF DO
02CF-02D0	STEP INCREMENT FOR AUTO
02D1	FLAG: TRACE ON/OFF
02D2-02D5	TEMPORARY STORAGE FOR VARIOUS ROUTINES
02D6-02E8	WORK AREA FOR PRINT USING, FIND AND CHANGE
02E9	UTILITY STORAGE FOR INDIRECT LOAD
02EA-02F1	TABLE: LENGTH OF FUNCTION KEY STRINGS
02F2	FLAG: SCROLL UP ACTIVE
02F3-02F4	BELOW RAM STACK
02F5-02F6	RAM ACTIVE STACK
02FB-02FC	REGISTER: BANKING - NMI
02FD-02FF	REGISTER: BANKING - UTILITY

HEX ADDRESS	DESCRIPTION
0300-0301	VECTOR : PRINT ERROR MESSAGE
0302-0303	VECTOR : BASIC DIRECT MODE - INPUT LOOP
0304-0305	VECTOR : TOKENIZE BASIC TEXT
0306-0307	VECTOR : BASIC LISTING TOKEN TO ASCII
0308-0309	VECTOR : INTERPRETER LOOP
030A-030B	VECTOR : CALCULATE ARITHMETIC FORMULA
030C-030D	VECTOR : USER TOKEN
030E-030F	VECTOR : USER TOKEN TO ASCII CHANGE
0310-0311	VECTOR : USER TOKEN EXECUTE
0312-0313	VECTOR : USR FUNCTION
0314-0315	VECTOR : IRQ HARDWARE INTERRUPT
0316-0317	VECTOR : BRK INSTRUCTION INTERRUPT
0318-0319	VECTOR : NMI - NON-MASKABLE INTERRUPT
031A-031B	VECTOR : KERNAL OPEN ROUTINE
031C-031D	VECTOR : KERNAL CLOSE ROUTINE
031E-031F	VECTOR : KERNAL CHKIN ROUTINE
0320-0321	VECTOR : KERNAL CHKOUT ROUTINE
0322-0323	VECTOR : KERNAL CLRCHN ROUTINE
0324-0325	VECTOR : KERNAL CHRIN ROUTINE
0326-0327	VECTOR : KERNAL CHROUT ROUTINE
0328-0329	VECTOR : KERNAL STOP KEY ROUTINE
032A-032B	VECTOR : KERNAL GETIN ROUTINE
032C-032D	VECTOR : KERNAL CLALL ROUTINE
032E-032F	VECTOR : I/O CONFIGURATION + BASIC WARM START
0330-0331	VECTOR : KERNAL LOAD ROUTINE
0332-0333	VECTOR : KERNAL SAVE ROUTINE
0334-0353	GET CHARACTER ADDRESS
003A	GET NEXT CHARACTER
0344	TEST IF CHARACTER OR NUMBER
0354-035E	LDA (txtptr),y
035F-0369	LDA (index1),y
036A-0374	LDA (index2),y
0375-037F	LDA (strng1),y

HEX ADDRESS	DESCRIPTION
0380-038A	LDA (lowtr),y
038B-0395	LDA (facmo),y
0396-0399	LDA (impptr),y
039A-039D	LDA (defptr),y
039E-03A1	LDA (linnum),y
03A2-03A5	LDA (varptn),y
03A6-03A9	LDA (lofac),y
03AA-03AD	LDA (lofac+3),y
03AE-03B1	LDA (hifac),y
03B2-03B5	LDA (hifac+2),y
03B6-03B9	LDA (facho),y
03BA-03BD	LDA (dscpnt),y
03BE-03C1	LDA (argmo),y
03C2-03C5	LDA (hifac+3),y
03C6-03C9	LDA (fndpnt),y
03CA-03D5	INDIRECT LOAD ROUTINE
03D6-03E1	LONG JUMP ROUTINE FOR NMI WITH RAM ACTIVE
03E2-03F2	LOAD ROUTINE FOR FUNCTION KEY VALUE
03F3-053F	BASIC STACK
0540-05FF	CASSETTE BUFFER
0600-06FF	RS-232 INPUT BUFFER
0700-07FF	RS-232 OUTPUT BUFFER
0800-0BFF	EXECUTE SPACE FOR VARIOUS ROUTINES
0C00-0FE7	SCREEN RAM
0FF8-0FFF	SPRITE ADDRESS POINTERS
1000-FEFF	BASIC WORK SPACE
FF00-FF27	BUFFER FOR DS, DS$
FF28-FFA7	STORAGE FOR FUNCTION KEY DEFINITIONS
FFE4-FFFB	UTILITY ROUTINE FOR RAM NMI

S'MORE BASIC COMPILER

If you want the speed of a machine language program, but don't have the time to become a machine language programmer, the S'more Basic Compiler is for you. The S'more Basic Compiler wil transform your slow running basic programs into quick machine code. The S'more Basic Compiler Will speed up basic program execution by as much as 2000% while compacting large basic programs into compact machine code requiring less memory. (NOTE: short basic programs may become longer due to the operating system overhead required by the compiler.) But, most importantly, the S'more Basic Compiler has been written specifically to take advantage of the extended memory and command set provided by your S'more Basic cartridge.

If you put a value on your time, think about how long it would take to save $40.00 worth of your time if your programs ran 20 times faster. The S'more Basic Compiler will be released during the late 3rd or early 4th quarter of 1985. The suggested retail of the S'more Basic Compiler will be **$39.95**.

Contact your dealer or CARDCO's customer service dept. for details and ordering information.

THE S'MORE DEMO DISKETTE

There are several worthwile programs (along with several worthless, but tutorial ones) on the Demo Diskette that has been provided with your S'more cartridge. The Demo Diskette is not copy protected, so please make a back up copy of the disk as soon as possible. If your Demo Diskette fails, return it with $3.ØØ to cover shipping and handling to CARDCO's Customer Service at the address shown at the beginning of this book.

NOTICE: The Demo Diskette is provided as a bonus, the disk is not advertised as part of the S'more Basic Cartridge unit sale, and until you opened the package you didn't even know it was there. CARDCO Inc. in no way guarantees the performance, serviceability or reliability of the disk or the programs on it. CARDCO will replace the disk, or sell you a new one for $3.ØØ including shipping and handling as described above.

The instructions for use of the Demo Diskette and the programs on it are on the disk itself and may be printed on your printer or seen on your video srceen. To use the instructions type:

```
RUN"I"  <RETURN>
```

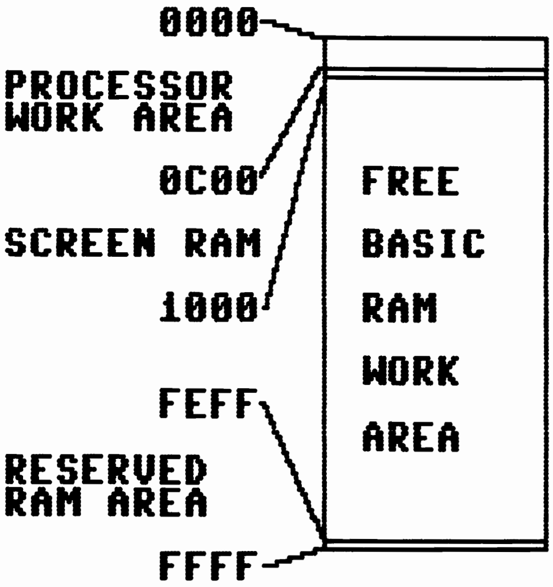
and follow the prompts on the screen.

There are no instructions provided for the SMON program. A list of the commands available has been provided, but it is beyond the scope of this program to teach you how to program using machine language.

Most of the other programs on the Demo Diskette are menu driven and should require little if any instruction for their use. Those instructions that are needed will be provided by the "I" program.

Remember, none of these programs is copy protected in any way. The purpose of providing these programs to you is for your educational benefit. We hope you take the time to look at the programs and see how the expanded capabilities of S'more Basic are used. Use the programs, list the programs, modify them or do whatever you want to with them as long as you learn from them.

If you write a program using S'more Basic and you would like to make it available to future purchasers of the S'more Basic cartridge, mail us a copy. If we find it to be of merit and include it on future Demo Disks, we will pay you $1ØØ.ØØ for all rights to your program.



S'more Basic Memory Allocation

This entire manual was composed using the **WRITE NOW!/64** word processor. This high performance word processor is available from your local CARDCO dealer. **WRITE NOW!** is available on quick loading and reliable cartridges for both the VIC-2Ø ($39.95) and the Commodore 64 ($49.95).

Because the **WRITE NOW!** word processor was designed by the same people that designed your **G-Wiz** printer interface, you can be assured that all of the advanced features of your printer and interface will be fully available for your use.

Additional features of **WRITE NOW!/64** are:

8Ø column output to the screen
Full header and footer capability
Cut & Paste buffer
Full block functions.
Full disk drive commands included
    (FORMAT, LOAD, SAVE, SCRATCH & RENAME
     disk files from within the program)
Full search and search/replace functions
Prints text directly from disk files
Full formatting commands
Single key non-distructive disk directory
Four on-line HELP screens available
Prints up to 99 copies of each document
Prints complete or partial documents
User defined tab stops
Page numbers can be located anywhere
Page number in standard or Roman numerals
Wait for keyboard input anywhere in text
Optional conditional page command available
Fully links with other NOW! series programs
Keyboard overlay included