BASIC™

A TUTORIAL

Commodore 64

ORBYTE™ SOFTWARE

*Commodore is a registered trademark of Commodore Business Machines.

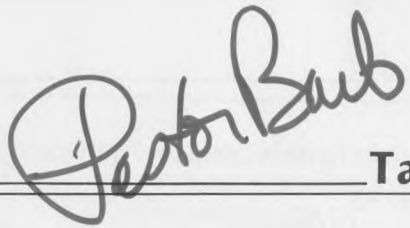


Table of Contents

TABLE OF CONTENTS

TABLE OF CONTENTS

Introduction To Basic, A Tutorial

A Message From Professor Orbyte

Welcome to the exciting world of computer programming. Whether you are a computer novice or whether you've already had experience with programming, you will find that your new program BASIC, A TUTORIAL by Orbyte Software will be an invaluable asset for using your Commodore 64 to its fullest potential.

BASIC, A TUTORIAL is a program that instructs you — in a tutorial manner — the computer language BASIC, Beginners All-purpose Symbolic Instruction Code. This program covers in detail all the basic aspects of this language including keywords, programming punctuation, the format of a BASIC program, subroutines, data handling techniques, setting up a program menu, improving your programs, and even an index of BASIC terminology. This workbook-style manual will also help you to follow along at your own pace and try actual on-screen programming examples corresponding to each lesson.

It is important to remember, however, that although BASIC, A TUTORIAL is designed to help you learn to program by teaching you the fundamentals of the BASIC language, your learning and retention of this material will ultimately depend on the effort you put into programming. If you seriously want to learn to program, you will have to spend much time studying the lessons of BASIC, A TUTORIAL again and again, and take the time to perform each of the lesson assignments. Computer programming can not be learned in one or two sittings.

In order to use this program effectively and reach your goal of programming in BASIC, the following learning guide has been designed. You are strongly advised to use it. . .

1. Each time you use BASIC, A TUTORIAL you will need to allow yourself at least two or three hours of non-interruption.

2. The BASIC lessons in this workbook-style manual are

extremely similar to the lessons displayed on your monitor screen. DO NOT avail this similarity as reason to merely skim or entirely ignore the manual, however. It has been specifically designed in this format for two reasons. First, the repetition both on-screen and in-text will enhance your comprehension. Secondly, the similarity enables you to study and review the lessons even when your computer is not available (ie. when traveling, when relaxing, etc.). Therefore, using your manual in conjunction with the program is extremely important.

3. To properly study the BASIC lessons, read the first screen of lesson text displayed on your monitor. When finished reading that text, read the manual text corresponding to the screen. Go on to the next screen, read it, then follow with the appropriate text in the manual. Continue in this way, while at the same time executing each of the examples as they appear in the lessons. We suggest that you work out these examples prior to pressing f1 and having them displayed on screen. It is also recommended to keep a notebook and take notes on important or challenging information in the lessons.

4. It is best to study only four lesson chapters at a time, (as you advance to the more difficult lessons you may wish to tackle them one at a time) then review them before proceeding to the next chapters. Repetition of the lessons will help to reinforce the topics covered and keep them fresh in your mind as you continue. Because the lessons progressively become more challenging, be sure that you thoroughly understand the lesson you are presently studying before proceeding to the next.

5. Be sure to do each of the assignments placed intermittently throughout the BASIC lessons in the manual. These assignments are more challenging than the short examples in the text and require you to perform actual programming steps. Through these assignments you will see how your knowledge of the BASIC language has progressed.

   To do these assignments, you will need to remove the BASIC, A TUTORIAL program disk from the disk drive, turn your computer off then on again, and perform the assignments directly on screen with your

computer. When you have successfully completed the assignment, shut the computer off then turn it on again. Reload BASIC, A TUTORIAL and continue with the next lesson.

If you choose, instead of turning off the BASIC program to do the assignment, you can simply write the assignment solution on paper. Then when you are finished using BASIC, A TUTORIAL, you can remove the disk, turn the computer off then on, and then enter the solution to see your programs perform on screen.

Sample solutions to the assignments are given in the APPENDIX for further assistance. Remember, however, that these are only sample solutions. In programming there are many means to the same result and therefore there exists great opportunity for creativity. Be innovative! Your solution may be different from the sample but still produce the same desired result.

6. When you feel you've learned enough for one sitting, put BASIC, A TUTORIAL away. (However, you may want to review the manual later in the day.) Take BASIC out on the following day, review the lessons you've already learned, and proceed to the next lessons.

7. When all the lessons and assignments of BASIC, A TUTORIAL have been completed, you will feel a great sense of achievement. And you should! You will have — with thought-provoking challenge and a lot of fun — successfully learned the basic concepts of the BASIC language and actually used them in writing short programs.

   If so desired, you may repeat the program again so that the concepts are more thoroughly understood. Keep in mind that the more times you review the lessons and assignments, the more familiar and fluent BASIC programming will become to you.

Programming your Commodore 64 is not only fun and challenging—it gives you an uplifting feeling of self-esteem and achievement. We at Orbyte Software hope that BASIC, A TUTORIAL will assist you in reaching this goal.

Getting Started

EQUIPMENT SET UP

Before you can begin using your BASIC, A TUTORIAL program, you will first have to set up all equipment in your computer system. This equipment includes:

1. Commodore 64 Keyboard
2. Commodore VIC 1541 Single Disk Drive
3. TV or Monitor
4. Commodore VIC 1525 Printer (or other printer that has been properly interfaced with the system*)

   *IMPORTANT: BASIC, A TUTORIAL is designed to operate with the Commodore 1541 disk drive and the VIC 1525 printer. Other printers are also compatible with BASIC, A TUTORIAL but these printers must be set up with the proper Commodore interface. Due to the mass variety of printers and interfaces now on the market, however, Orbyte Software cannot guarantee this compatibility.

To set up your computer system's equipment, follow the instructions in the Commodore 64 User's Manual.

Lesson #19 of BASIC, A TUTORIAL concerns using a printer when writing programs. Although it is not required to use a printer when using BASIC, it is helpful in fully understanding printer techniques. If you will be using a printer, hook it up to your computer system at this time and insert computer paper (following the instructions supplied with your printer).

Next, turn on all the equipment, making sure that the Commodore 64 keyboard is the last to be activated. Turning on the keyboard before the other equipment may damage the system. Upon turning on all the equipment, the following screen will appear . . .

```
**** COMMODORE 64 BASIC V2 ****

64K RAM SYSTEM  38911 BASIC BYTES FREE

READY.
```

(If this screen doe not appear, check to see if the equipment has been properly connected. Turn the computer off, then on again. Consult the Commodore 64 User's Manual for any difficulties.)

LOADING THE PROGRAM

When the above screen appears, insert BASIC, A TUTORIAL disk A into the disk drive. The cursor on the screen will be flashing, indicating that the computer is waiting for a command from you. To load the BASIC program from disk A, type in LOAD"PART ONE",8 then hit the RETURN key. The computer will now search for and load the program indicated by the message SEARCHING FOR PART ONE LOADING.

When the word READY appears, type in RUN then hit RETURN. The title screen will then be displayed. At this time, press the F7 function key located to the right of the keyboard. The first screen of lesson text "INTRODUCTION" will then appear. If you wish to continue reading the text, press f7 or if you prefer to go to the MAIN MENU,press M. (The MAIN MENU allows you to skip to a particular lesson, a particular BASIC keyword, or again start from the first screen of lesson text. This menu will be explained in greater detail in the following section.)

If you decide to press f7 during the first screen of text (INTRODUCTION) to continue with the lessons, go to the section of this manual LESSON 1 and continue from there. If you decide to press M to go to the MAIN MENU, continue with the following section.

NOTES:

1. You will notice that at the bottom of each screen displaying lesson text there is a message:

   Press f7 to continue - f1 to go back

   This notifies you that by pressing the f7 key you will be brought to the next screen of lesson text. If you press the f1 key you will be brought back to the preceeding text screen. Pressing the M key when this

message is displayed will automatically access the MAIN MENU.

2. Disk A contains your BASIC, A TUTORIAL lessons 1 - 12. Disk B contains lessons 13 - 20. If you wish to continue with lesson 13 immediately after finishing lesson 12, follow the directions at the end of lesson 12: press f7 to load; remove disk A from the disk drive and insert disk B; press f1 to load disk B. Lesson 13 will immediately begin after the program is loaded.

   If you wish to turn off your computer after lesson 12 and continue with lesson 13 at another time follow these instructions: When you are ready to load disk B (lessons 13 - 20), insert disk B into the disk drive and type LOAD"*",8 RETURN. When the word READY appears, type RUN RETURN. Lesson 13 will automatically begin.

The Main Menu

THE MAIN MENU . . . AN OVERVIEW

Upon pressing the M key in any of the screens displaying lesson text, the following screen will read:

Press f7 to load menu.

Press f1 to return to lesson.

If during this screen you press f1, you will automatically be brought back to the first screen of the lesson you were currently studying. If you press f7 while in the above screen, the screen will display the message PLEASE STAND BY . . . indicating that the MAIN MENU is being loaded. After it is loaded, it will be displayed as follows:

In using this BASIC TUTORIAL you may:

f1 START TUTORIAL FROM BEGINNING

f3 GO TO A PARTICULAR LESSON

f5 GO TO ALPHABETICAL INDEX

Press function key (right side of keyboard) indicating choice.

The three options available in this menu are as follows:

f1 - The f1 function key allows you to start BASIC, A TUTORIAL from the first screen of lesson text.

f3 - The f3 function key will call up the table of contents of all the lessons in BASIC, A TUTORIAL. From this listing of lessons, you can choose the one you want to study.

f5 - The f5 function key calls up an alphabetical listing of keywords and techniques used in BASIC programming.

From this listing you can select a particular item and automatically proceed to the section of BASIC, A TUTORIAL that teaches you about it.

As indicated in the MAIN MENU screen, select the option you want and press the corresponding function key (f1 to START TUTORIAL FROM BEGINNING, etc.) Follow by reading the explanations of each option below.

f1 START TUTORIAL FROM BEGINNING

Upon pressing f1 in the MAIN MENU, the screen will read PLEASE STAND BY. . . The program will then be loaded and the title screen will be displayed. Press f7 and the first screen of lesson text will begin.

f3 GO TO A PARTICULAR LESSON

Upon pressing f3 in the MAIN MENU, the table of contents will immediately appear listing all the various lessons of BASIC, A TUTORIAL.

TABLE OF CONTENTS

f7 down - f1 up - f3 to select lesson

f5 to return to menu

As you will notice, the word INTRODUCTION appears in red letters. As notified by the message at the bottom of the screen, press the f7 key to move the red letters down or the f1 key to move the red letters up until they are at the lesson you want to study. When the lesson number and name appear in red, press the f3 key to go to that lesson. Or, if you decide not to choose a lesson and instead want to go back to the MAIN MENU simply press f5.

Upon selecting a lesson and pressing f3, the screen will read PLEASE STAND BY. . . while it is loading that particular lesson. Upon loading, the first screen of the lesson you chose will be displayed.

f5 GO TO ALPHABETICAL INDEX

Upon pressing f5 in the MAIN MENU, the screen will appear blank for a moment, then the ALPHABETICAL INDEX will appear as follows:

```
                ALPHABETICAL INDEX
        Abbreviating keywords
        Addition
        Arithmetic operations
        ABS
        AND
        ASC
        ATN
        Background color
        Basic keywords
        Border color
        Bubble sort example
        Clearing the screen
        Color printing
        Combining disc files
        Commas
        Copying files
        Cursor controls
        CHR$
        CLOSE
        CLR
        CMD
  f7 down - f1 up - f4 to select
  f5 new page - f6 last page - M = menu
```

As you will notice, the first item in the index "Abbreviating key words" appears in red. Use the f7 key to move the red letters down or the f1 key to move the red letters up until the item you want to

select is in red. When it is in red, press f4 and the lesson containing an explanation of the item you chose will be loaded and displayed on screen. If the item you want to look up is not listed in this screen, press f5 and the next screen of the ALPHABETICAL INDEX will appear. Press f5 and the index listing will continue again. Pressing f6 will bring you back to the previous index screen. Pressing the M key will bring you back to the MAIN MENU.

The entire ALPHABETICAL INDEX consists of seven screens. The first of these screens has been described above. The others are as follows:

ALPHABETICAL INDEX

CMD
CONT
COPY
COS
Data Handling
Deleting program lines
Deriving trig functions
Digital clock example
Direct mode
Disc command channel
Disc files
Disc utilities
Division
Dominoe program example
DATA
DEF FN
DIM
Entering program lines
Erasing files
Exponentiation
END

f7 down - f1 up - f4 to select
f5 new page - f6 last page - M = menu

ALPHABETICAL INDEX

END
EXP
Floating-point variables
Format of Basic programs
Formatting new discs

f7 down - f1 up - f4 to select
f5 new page - f6 last page - M = menu

ALPHABETICAL INDEX

f7 down - f1 up - f4 to select
f5 new page - f6 last page - M = menu

ALPHABETICAL INDEX

f7 down - f1 up - f4 to select
f5 new page - f6 last page - M = menu

ALPHABETICAL INDEX

ALPHABETICAL INDEX

LESSON 1

The Keyboard

1 - 1. THE COMMODORE 64 KEYBOARD

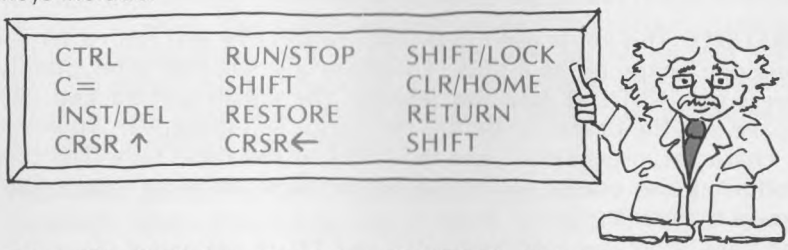
The Commodore 64 keyboard contains two character sets that you can use from the keyboard or in your programs. The set that is active when you first turn on your computer is the UPPER CASE/GRAPHICS set. In this set the upper case alphabet and the numbers 0-9 are available. Additionally, a large number of graphics characters are provided on the front of many of the keys.

The second character set is called the UPPER/LOWER CASE character set. It provides both capital and small letters and a few graphics characters.

To switch between sets press the SHIFT and the Commodore logo key C= at the same time. The rest of this lesson will describe each of the two keyboards and the special control characters such as SHIFT and C= that you just learned.

1 - 2. KEYBOARD CONTROL KEYS

There are 66 keys available on the Commodore keyboard. Of these, 12 keys do not actually produce characters. They are used to control the computer or modify the other keys. These control keys include:



CTRL The CTRL key stands for CONTROL. This key is used mainly to set the color of the screen text and reverse video characters. Note that the numbered keys 0-8 have color abbreviations on the front of them. By pressing the CTRL key and any one of these keys at the same time, you can change the color of the text.

For example, if you press CTRL and the 3 key at the same time and then type in some text, it will appear red.

Key 9 RVS ON causes text to appear in reverse video. If you press CTRL and the 9 key at the same time and type in some text, it will appear in reverse video.

Key 0 RVS OFF turns off the reverse video when used with the CTRL key. As you can see, the CTRL key has a great effect on the function of the numeric keys.

RUN/STOP This key is used to start and stop program operation. During the operation of a program if the RUN/STOP key is pressed, the program will stop and an advisory BREAK IN × × × × READY will appear on screen. If the SHIFT key is pressed at the same time as the RUN/STOP key, the computer will attempt to load and run a program from the cassette tape unit.

SHIFT LOCK This key simply latches the keyboard as if the SHIFT key was being held down constantly. To release the shift mechanism, simply press SHIFT LOCK again.

C= AND SHIFT KEYS These two keys are mainly used to modify the functions of the other keys. Their operation will be more fully described in the discussion of the two keyboards coming up later in this lesson.

CLR/HOME This key has two related functions. Pressing the CRL/HOME key by itself causes the text cursor to move to the first position of the top screen line. This is the HOME function. By pressing the SHIFT key at the same time as the CLR/HOME key, you will still cause the cursor to home but you will also cause the screen to clear. If you press the SHIFT key and the CLR/HOME key simultaneously, the screen will go totally blank and any text you enter afterward will appear on the top line of the screen.

INST/DEL This key is used to INSERT or DELETE text from a screen line one character at a time. To delete a single character, simply press the INST/DEL key and release. The cursor will backup one space and the character before the cursor will disappear. To insert a character, press SHIFT and INST/DEL at the same time. The text following the cursor will move to the right allowing you a free space to type in.

RESTORE This key is used to exit a program and restore the computer to the state it was in when it was first turned on. At times this is the only way to get out of a program. To restore, simply press and hold the RUN/STOP key and press the RESTORE key at the same time.

RETURN This key is used just as the return key on a typewriter is used. The only difference is that technically the RETURN key on the Commodore 64 causes data or a program line to be entered into memory rather than causing a mechanical carriage to return as on a typewriter.

CRSR← The farthest right key on the bottom row is the cursor left or right key. By pressing this key alone you cause the blinking cursor to move to the right. If you press the SHIFT key at the same time as the CRSR key, you will cause the cursor to move to the left.

CRSR ↑ The next key to the left of the cursor left or right key is the cursor up or down key. Pressing this key causes the cursor to move down the screen. When shifted, this key causes the cursor to move up the screen.

1 - 3. MAIN KEYBOARD

When the computer is first powered up the UPPER CASE/GRAPICS character set is active. For ease of discussion we will call this line the main keyboard. The main keyboard contains the previously discussed control keys, the lettersA through Z, the numbers Ø through 9, four function keys, the usual punctuation characters found on any typewriter, and some special math function keys.

ALPHABETICAL KEYS The alphabetical keys will produce the normal alphabet as on a normal typewriter except they will be all capital letters. The UPPER CASE/GRAPHICS mode is called that because if you shift one of these alpha keys, or if you press the C= key and an alpha key, you get one of a set of strange symbols used as graphic characters. These symbols are combined to create graphic pictures on screen.

Notice that on the front edge of each alpha key there are two graphic symbols. The symbol on the left front of each key is obtained by holding down the C= key while pressing the desired alpha key. To obtain the graphic character on the right side of the front edge of any alpha key, simply press the SHIFT key at the same time as the desired alpha key.

NUMERIC KEYS The keys numbered 0 through 9 produce those numbers when pressed in the UPPER CASE/GRAPHICS mode. However, when the SHIFT key is held at the same time, they produce the punctuation marks shown above the numbers(!"#$%&'()). Along the front edge of each numbered key is an abbreviation for a color. Recall from an earlier discussion that the color on screen could be changed to any one of these eight colors by holding the CTRL key down while pressing the desired color key. As it so

happens there are yet eight more colors available for use. By holding the C= key and pressing the desired color key you may obtain a slightly different shade of color.

FUNCTION KEYS On the far right hand side of the keyboard are four keys labeled f1, f3, f5, and f7. These are special function keys which you may define in your programs. Unless defined, these keys have no effect. They will be described further in future lessons.

1-4 ALTERNATE KEYBOARD

The second character set available on the Commodore 64 computer is the UPPER/LOWER CASE set. This character set behaves much more like a normal typewriter in that all the alphabetical keys cause lower case letters when pressed and capital letters when shifted. There are still some graphics characters available by holding the C= key down but the graphics that were available from the main keyboard when the SHIFT key was pressed are missing. The numbered keys and function keys operate in the alternate keyboard exactly the same way that they did on the main keyboard. Do not be concerned if you do not completely understand the use of every key on the keyboard at this point in the tutorial. Each of the keyboard functions introduced so far will be further described in lessons to come.

LESSON 2

Format of a Basic Program

2 - 1. DIRECT AND PROGRAM MODES

There are two primary program modes on the Commodore computer - DIRECT and PROGRAM.

In the DIRECT mode, programming commands are entered from the keyboard with no line numbers and the RETURN key is pressed. This command is then executed (performed) immediately. For example:

```
PRINT "JOE"
```

This is a direct command to print the word JOE. To execute this command after typing it in (entering) simply press the RETURN key. Joe will then be immediately printed on screen.

In the PROGRAM mode, programming statements are entered with line numbers in front of them. When the computer notes a line number, it places the statement in memory for later execution. In this way a number of program commands can be stored and so build a program line by line. For example:

```
10 A=2
20 B=9
30 PRINT A+B
```

Here are three program statements numbered 10, 20, and 30. When this program is run, the program statements will be executed in order and the result (11) will be printed on screen. In this way programs of hundreds or even thousands of lines of Basic statements can be written.

2 - 2. KEYWORDS - THE BASIC BUILDING BLOCKS

The Commodore 64 interpreter contains a list of 71 Basic keywords that have special meanings for the computer. These keywords can each be translated into a form of machine code that the computer can execute. These keywords are the blocks with which you can build a meaningful program.

There are two types of keywords; those with arguments and those without. The arguments represent data that the Basic keyword can operate on. This data always appears immediately after the keyword. An example of a Basic keyword with an argument would be as follows:

```
SQR(25)
```

The SQR is the Basic keyword which stands for square root. The 25 is the argument. This line will take the square root of 25. The result of this statement would of course be 5.

Many keywords do not have arguments. For example:

```
NEW
```

This keyword has no data following it. The keyword NEW causes the program that is currently stored in the computer to be erased so a new program may be entered.

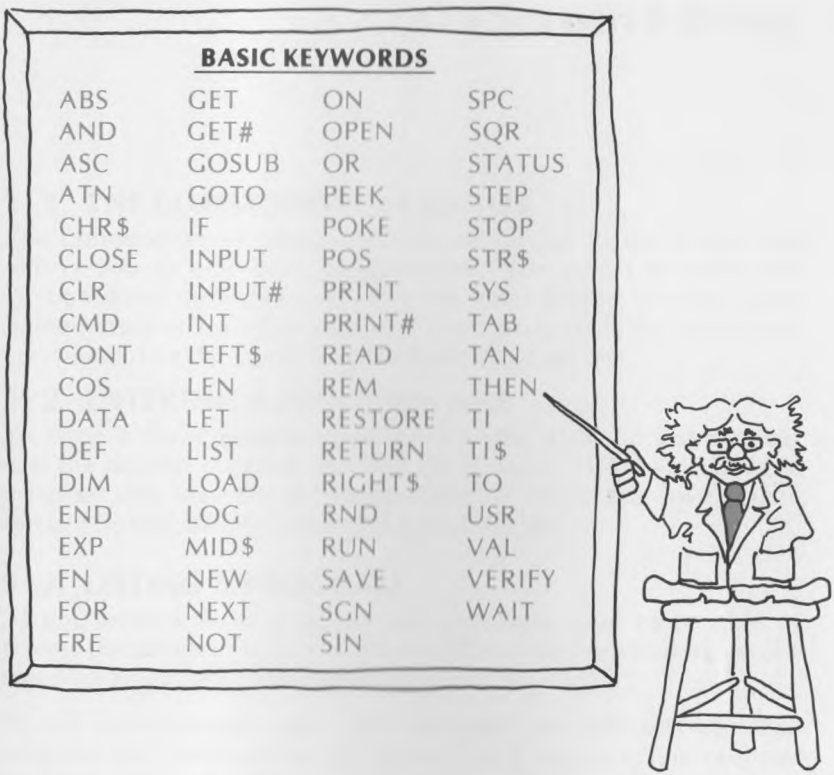
Keywords are abbreviations. These abbreviations must be entered exactly. You cannot enter SQUAR in place of SQR to get a square root.

The table (right) lists all 71 keywords used in the Commodore Basic language. Do not be concerned if you do not understand the purpose of each one of them. They will be discussed in future lessons.

2 - 3. PUNCTUATION - SPECIAL MEANINGS

In addition to the keywords, the punctuation symbols have special meanings for the computer. Often these symbols are used with the keywords to perform a certain function.

For example, the asterisk (*) is commonly used in English to indicate a footnote. But in the Basic language it is used as the mathematical multiplication symbol instead of ×. Other punctuation symbols also have different meanings. These symbols will be introduced as needed in future lessons.


BASIC KEYWORDS
ABS GET ON SPC
AND GET# OPEN SQR
ASC GOSUB OR STATUS
ATN GOTO PEEK STEP
CHR$ IF POKE STOP
CLOSE INPUT POS STR$
CLR INPUT# PRINT SYS
CMD INT PRINT# TAB
CONT LEFT$ READ TAN
COS LEN REM THEN
DATA LET RESTORE TI
DEF LIST RETURN TI$
DIM LOAD RIGHT$ TO
END LOG RND USR
EXP MID$ RUN VAL
FN NEW SAVE VERIFY
FOR NEXT SGN WAIT
FRE NOT SIN

LESSON 3

On Screen Editing

3 - 1. THE COMMODORE 64 EDITOR

The Commodore 64 editor controls the output to the screen and allows you to edit Basic program text. The editor monitors the keyboard and determines whether the input should be acted upon immediately or stored as part of a Basic program. This lesson tells you how to use the editor to write Basic program text.

3 - 2. ENTERING A PROGRAM LINE

To enter a Basic program line, type in the assigned line number and the desired program text and press the RETURN key. A Basic program line may not be longer than 80 characters (two screen lines). Any text entered after character 80 is lost.

3 - 3. LISTING A PROGRAM

As you write a Basic program you obviously want to be able to review the program lines you have written. The Commodore screen can display 25 lines of text vertically.

To list your program type LIST and press the RETURN key. The program will printout on the screen. Each program line requires at least one and sometimes two screen lines to display. Many times your program is too long to display all of it on the screen at the same time. The program will scroll up the screen until the last line of program has been listed.

To list a select group of program lines, enter the following command in DIRECT mode:

```
LIST 22000-22050
```

This causes all program lines between 22000 and 22050 to printout on the screen. You may also list from a particular program line to the end of program memory as follows:

```
LIST 22030-
```

This will cause all program lines numbered 22030 or greater to print out. You may also list one single program line like this:

```
LIST 22030
```

This would list only line 22030.

3 - 4. DELETING PROGRAM LINES

To delete a program line entirely, simply type the program line number and press the RETURN key. This causes that line to be completely erased from the program.

3 - 5. ALTERING PROGRAM LINES

To edit an individual program line you must first list the line on screen. Once it is on screen you may position the blinking cursor over the text on that line using the cursor control keys (CRSR ↑ and CRSR←).You may then edit that text by typing over existing text and using the INST/DEL key to insert or delete characters. When the line is corrected to your satisfaction, press the RETURN key to enter the new program line in place of the old one.

3 - 6. COPYING PROGRAM LINES

At times you will find that the same program line is used at a number of places in your program. Rather than retype the line each time it is needed, it is much easier to recopy the line. Simply list the desired line, type a new line number over the old one, and press the RETURN key to enter the line. The original line will remain unchanged and the new line will be an exact duplicate except it will have a different line number.

3 - 7. CLEAR/HOME

The CLR/HOME key is located in the upper right hand corner of the keyboard. This key is used to reposition the cursor up to the first character of the top screen line. This is the HOME position. When used with the SHIFT key, the home function is still performed but the screen is also cleared of all text and graphics.

3 - 8. THE NEW COMMAND

One Basic keyword that is almost never used in a program is the NEW command. By entering this command in the direct mode, all Basic program lines are completely erased from the computer memory. This is a quick way to clear the computer before you start a new program - thus it's name.

LESSON 4

Simple Screen Printing

4 - 1. THE VIDEO SCREEN AS AN OUTPUT DEVICE

Your computer has the ability to send information to a number of output devices such as printers, modems, disc drives, etc. The video screen or television is by far the most used output device in your system.

One of the reasons that home computers are so useful is that they are interactive. Basically that is to say you can talk to them. Most commonly the computer speaks through the video screen. In reply, you may enter data using the keyboard. Most popular programs are based on this concept of communication. Because of this, one of the most useful things you can learn in any computer language is the technique for printing things on the screen. This lesson will introduce some of those techniques.

4 - 2. THE PRINT STATEMENT

The PRINT statement is used to send data to the screen. The data may be a variable or a literal. Literals are always enclosed in quotes as in the following Basic Print statement:

```
PRINT "JOHN DOE"
```

Normally each time a PRINT statement is executed the data is printed on the next available screen line. A group of PRINT statements like those below would then print vertically down the screen.

```
100 PRINT"JOHN DOE"
110 PRINT"JANE DOE"
120 PRINT"DAVE DOE"
130 PRINT"PHIL DOE"
```

Each name printed by the statement above would print starting at

the first character position of each line as shown below:

JOHN DOE
JANE DOE
DAVE DOE
PHIL DOE

There are no specific limits to the length of the data that is enclosed in quotes. However, keep in mind that the total line length of the Basic statement can never exceed 80 characters (two screen lines).

4 - 3. SPACING WITH COMMAS

The comma is one of several punctuation symbols that can be used to format PRINT statements. The comma causes the print cursor to move to the next available preset tab position. These preset tab positions occur every ten characters across the line (1, 11, 21, and 31). Using commas we could reformat the printing of our four names as shown below:

```
100 PRINT"JOHN","JANE","DAVE","PHIL"
```

Note that in our example statement each name is enclosed in quotes and separated by commas. When executed, this statement will print the names on screen much as shown below:

JOHN JANE DAVE PHIL

The four names are printed with each name beginning at one of the preset tab positions on the same line. If we had printed five names instead of four, the fifth name would have still been printed at the next available tab preset, in this case character 1 of the next line.

4 - 4. SPACING WITH SEMICOLONS

The semicolon is used very much like the comma was used in the last example. In fact, the way it is used in a Basic print statement is the same. The results are a bit different however.

Semicolons are used to splice things together on a screen line. Any data listed after a semicolon will be printed at the next screen position. Let's take our previous example and insert semicolons in place of the commas we used before.

```
100 PRINT"JOHN";"JANE";"DAVE";"PHIL"
```

This statement will cause all the names to be run together on one screen line as shown below:

```
JOHNJANEDAVEPHIL
```

This can be quite useful particularly when dealing with more than one program line. The example below illustrates this:

```
100 PRINT"JOHN ";
110 PRINT"DOE"
```

These print statements will cause both names to appear on the same line as shown below. Note that a space is included after the name JOHN in line 100.

```
JOHN DOE
```

4 - 5. SPACING WITH TAB

As you can see, commas and semicolons are quite useful for formatting print statements. Unfortunately, they are not very flexible. The TAB keyword does provide some flexibility. This keyword allows you to advance the print cursor position by any number of spaces up to 255. The argument for this keyword is the number of spaces. Consider the following example:

```
100 PRINT TAB(20)"JOHN DOE"
```

This statement will cause JOHN DOE to be printed 20 spaces to the right rather than at the first character position of the line.

Assignment:

Write a short program to print the names and ages of each person in your family in the center of the screen. Refer to Appendix A for a sample solution.

LESSON 5

The Variable

5-1. VARIABLES

One of the reasons that computers can be so useful is that they have the ability to remember a large number of bits of information without error and update that information as needed. The concept of the variable is key to using this ability.

Variables can be thought of as storage spaces where your programs can store data. You create these spaces by giving them a name. You may then assign a value to these variables by setting them equal to constants or as the result of calculations performed in the program.

In the Basic language data must be one of three general types:

- INTEGER NUMBER
- FLOATING-POINT NUMBER
- STRING

Eash of these three data types and the variables used to store them will be described in the remaining paragraphs of this lesson.

5-2. INTEGER VARIABLES

Integer variables can contain only whole numbers (numbers without decimal points) between −32768 and +32767. It should be noted that when dealing with numerical data of any type NEVER USE COMMAS inside a number. As a matter of interest, integer numbers are stored in memory as two-byte binary numbers.

The name of an integer variable can be any combination of the alphabet and the numbers 0-9. The first character of the name must be a letter of the alphabet and the name cannot contain any of the Basic keywords.

Actually, the Commodore 64 only recognizes the first two characters of the name. Naturally you would not want two different variables to have the same first two letters. An integer variable

should have a percent sign (%) as the last character. This is called a type designator and it shows that the variable is an integer type. For example:

```
200 AB% = 127
210 PRINT AB%
```

In line 200 we are assigning the variable AB% A VALUE (127). Line 210 prints whatever value is stored in variable AB% on the screen (127 in this case).

5 - 3. FLOATING-POINT VARIABLES

Floating-point variables are more powerful than integer variables in that they can store any decimal number between −999999999 and +999999999. These number may include decimal fractions and mixed numbers such as 127.786. However, floating-point variables use more memory than integer variables (5 bytes versus 2 bytes). For this reason they should be used in place of integer variables only where this precision is needed or where memory capacity is not a concern.

Floating-point variables are named in the same manner as integer variables except that the type designator (%) is left off. The following example shows this:

```
200 AB = 127.58
210 PRINT AB
```

Line 200 assigns the value 127.58 to the floating-point variable AB. Line 210 prints the value of AB to the screen.

Note that when the AB variable prints, it will appear in the second space from the left side of the screen rather than the first space. When printed, all numeric variables carry one leading space and one trailing space.

5 - 4. STRING VARIABLES

String variables are slightly different from our previous variable types in that they are used to store alphanumeric data. All the letters of the alphabet, numbers and symbols can all be stored in string variables. The only character you cannot store in a string variable is the quotation mark. That is because the quotation symbol is used to define the string.

String variables can hold up to 255 characters but you must be

careful not to exceed the normal 80 character limit for the Basic program line that assigns the string variable a value.

Like integer variables, string variables must have a type designator as the last character of the name. For string variables this type designator is the dollar sign ($). The following example shows the use of the dollar sign and quote marks when assigning string variables:

```
300 A$="CATERPILLAR TRACTOR CO."
310 PRINT A$
```

Line 300 assigns A$ the value "CATERPILLAR TRACTOR CO.". Line 310 will print the contents of A$ on screen.

5 - 5. MIXING DATA TYPES IN PRINT STATEMENTS

At this point we know how to print literal data and how to print variable data using the PRINT statement. In most programs it is desirable to be able to print both types of data on one screen line using one print statement. This is actually quite easy to do.

```
200 I=2217.75
210 PRINT"JANUARY INCOME=$";I
```

Note that in the above example the literal data JANUARY INCOME=$ is separated from the variable I by a semicolon. As you may recall from a previous lesson the semicolon is used to splice two items together on a screen line. These two print statements would cause a display similar to that below:

```
JANUARY INCOME=$ 2217.75
```

Let's take a look at a bit longer example that illustrates the same thing.

```
200 A$="JANUARY "
210 I=2217.75
220 PRINT A$;"INCOME=$";I
230 A$="FEBRUARY "
240 J=2257.98
250 PRINT A$;"INCOME=$";J
260 A$= TOTAL "
270 PRINT A$;"INCOME=$"I+J
```

Note that in line 270 we have also performed a simple math function within a print statement by adding variables I and J. The

program lines above will produce a screen printout like that shown below:

```
JANUARY INCOME = $ 2217.75
FEBRUARY INCOME = $ 2257.98
TOTAL INCOME = $ 4475.73
```

Note that a space appears between the dollar signs and the numbers in this last example. This is because numeric variables are printed with a leading space and a trailing space. The leading space appears between the dollar sign and the number.

As you can see, mixing data types in print statements is quite easy. The important thing to remember is to make sure there is a semicolon or a comma separating the data elements in the print statements.

LESSON 6

Simple Structures

6-1. THE GOTO STATEMENT

As you have seen, the lines of a Basic program are normally executed in the order of the line numbers. The GOTO statement allows us to modify this action so that the program jumps to a different line rather than the next line in sequence.

The argument for the GOTO statement is the number of the program line you want to jump to. Consider the following example:

```
200 GOTO 250
210 PRINT"THREE"
220 GOTO 270
230 PRINT"TWO"
240 GOTO 210
250 PRINT"ONE"
260 GOTO 230
270 PRINT"FINISHED"
```

This nonsensical bit of program simply illustrates how the GOTO statement can be used to alter the order of execution in a Basic program. This example will cause a screen printout as shown below:

```
ONE
TWO
THREE
FINISHED
```

6-2. SUBROUTINES

Sometimes different parts of your program will have similar jobs to do, and you will find yourself typing in the same program lines several times. This is not necessary. You may type the lines once as a subroutine and then call that subroutine from anywhere in your program. This is accomplished using the GOSUB and RETURN statements.

The GOSUB statement (GO to SUBroutine) operates like the GOTO

GOSUB .... RETURN

statement in that it causes the program to jump to the line number specified in the argument. This would be the line number of the first line of the subroutine. The last line of the subroutine must be the RETURN statement. This causes a jump back to the line after the GOSUB statement. You might say that the GOSUB statement is just like a GOTO statement that remembers where it came from. Let's take a look at a short example.

```
200 A=2000
210 GOSUB 300
220 A=3000
230 GOSUB 300
240 A=5000
250 GOSUB 300
260 STOP

300 PRINT A
310 A=A+25
320 PRINT A
330 RETURN
```

This program sets the variable A to a series of values, calling the subroutine at line 300 after each value. The subroutine at 300 prints A, adds 25 to it, and prints the new value.

As you can see, this technique can save a great deal of typing and quite a bit of memory space.

6-3. THE LOOP

One of the computer's handiest traits is that it just never gets bored. With great patience it will perform the same task over and over.

The loop is a programming technique that allows us to repeat a section of our program a specified number of times. The Basic language really offers only one type of loop. That is the FOR/NEXT loop. In this form, we set a variable, run through the section of program we want to use, add 1 to the number in the variable, and check to see if that number equals the number of times we want that program section to run. If it does, we exit that section of program.

The statement that allows this is the FOR/NEXT statement. Consider the example below:

```
200 FOR I=1 TO 5
210 PRINT I
220 NEXT I
230 PRINT"DONE"
```


FOR /NEXT 1 or Step _

Line 200 shows that we will use the variable I as our counter and that we want to perform this loop 5 times. Line 210 is the portion of the program that we want to repeat. Here it is only one line, but we could just as easily have a hundred lines of code in its place.

Line 220 is where the real work is done. First it checks to see if I is equal to 5. If so it allows the program to go on to the next program line. If I was not yet equal to 5, this line adds 1 to the number held in I and jumps back to line 210 for another repetition of this program segment. The process of adding 1 to a number is called an increment. Subtracting 1 from a number is called a decrement.

Actually, we are not restricted to just incrementing our variable by 1 during the NEXT statement. By adding a STEP statement to line 200 we can increment our variable by whatever value we like. Consider the following example:

```
200 FOR I=0 TO 10 STEP 2
210 PRINT I
220 NEXT I
230 PRINT"DONE"
```

The number following the STEP statement can be any number up to 255 except 0. If we did use 0 we would never increment our variable and so we would be stuck in the loop forever. This is called an endless loop. You can, however, break out of these sort of situations using the RUN/STOP key.

We can also count down through the loop by making the step increment negative.

```
200 FOR I=10 TO 2 STEP -2
210 PRINT I
220 NEXT I
230 PRINT"DONE"
```

If this example we set the variable I to a value of 10 and counted down to 2 in increments of 2.

6 - 4. NESTING LOOPS

It is possible to run two FOR/NEXT loops together, one inside the other. The important thing to remember here is that one loop is entirely inside the other. What must be avoided is two FOR/NEXT loops that overlap without either being entirely inside the other.

Consider the following example:

```
200 FOR A=0 TO 6
210 FOR B=0 TO A
220 PRINT A;":";B;" ";
230 NEXT B
240 PRINT
250 NEXT A
```

This example prints out a set of six-spot dominoes on screen. Note that the B loop (lines 210-230) is held entirely within the A loop (lines 200-250).

In the dominoe example we also introduce another concept of note. On line 210 we use the current value of the A loop as the limit for the B loop count. For example: during the third iteration of the A loop the value of A would of course be 3. This would cause the B loop to run 3 times. The next iteration of the A loop would cause the B loop to run 4 times and so on.

Note that in the printout caused by this example, the left side of each dominoe is the value of the A variable and the right side of each dominoe is the value of the B variable.

6 - 5. TERMINATING PROGRAMS

When a Basic program runs it will stop automatically when it runs out of lines to execute and will return to the direct mode. Many times you will have more than one program in memory and it is convenient to have each one stop on its own. For these reasons it's considered good form to terminate any program with a STOP or END command.

These commands have no argument and they both do the same thing - stop the program. The only difference between the two commands is the message displayed on screen after the program stops.

When a program is ended by a STOP command, the message BREAK IN LINE XXXX READY will appear on screen. If the END command is used, the message will simply read READY.

Interestingly, you can place STOP commands throughout a program in order to check variables etc. When the program stops you can print variables, examine program lines, or list the program. As long as you do not edit any of the program text, you can restart the program where it stopped by entering a CONT command in the direct program mode. This can be a very useful technique when debugging (finding the errors in) troublesome programs.

LESSON 7

Arithmetic Operations

7-1. ADDITION (+)

Addition is performed on the Commodore computer just as it is in normal arithmetic with the plus sign (+) indicating that the two quantities are to be added together. The following example illustrates this.

```
100 A=3+2
110 B=7+11
120 C=A+B
130 PRINT A,B,C
```

Note that we have used an arithmetic operation in the statement that assigns a variable. This is allowed.

7-2. SUBTRACTION (−)

The subtraction operation also functions like normal arithmetic and the minus sign (−) is used to indicate a subtraction. Consider the following example:

```
100 A=26
110 PRINT A
120 A=A-11
130 PRINT A
```

Note that in line 120 we used the variable A in the arithmetic operation that assigned a value to A. Although this may seem a bit odd, it is acceptable and a common programming technique.

7-3. MULTIPLICATION

In almost all computers today the asterisk (*) is used to indicate a multiplication operation. This is to avoid confusing the letter X with the multiplication symbol ×. The following example shows the use of the asterisk as a multiplication symbol:

```
100 A=12
110 B=7*A
120 PRINT A,B
```

7 - 4. DIVISION (/)

The slash symbol (/) is used to indicate the division operation. The number to the left of the / is divided by the number to the right.

```
100 AF=25
110 DD=AF/4
120 PRINT AF,DD,DD/2
```

Note that we can include arithmetic operations in print statements as was done in line 120.

7 - 5. EXPONENTIATION

Multiplying any number by itself a given number of times is known as exponentiation. For example: 2*2 is the same as 2 squared or 2 raised to the second power. The Commodore 64 uses an up arrow symbol (↑) to indicate exponentiation. The number to the left of the arrow is raised to the power indicated by the number to the right of the arrow. Consider this example:

```
100 A=2↑2
110 B=3↑3+A↑2
120 PRINT A,B
```

Note that line 100 sets A equal to 2 squared (4). Line 110 sets B equal to 3 cubed (27) plus A squared (16). This equals 43.

7 - 6. HIERARCHY OF OPERATIONS

You may have noticed that in some of our examples we have combined more than one arithmetic operation in an expression. A=20*4/13−6↑4 would be a suitable example of this. In order to predict what the computer will calculate this expression to be it is necessary to understand the hierarchy of operations. This is simply the order in which each operation will be performed. This order is shown below:

Negative Numbers

EXPONENTIATION (↑)
MULTIPLICATION AND DIVISION (* /)
ADDITION AND SUBTRACTION (+ −)

The system first goes through the entire expression and checks for up arrows (↑) indicating numbers with exponents to be resolved. In our example A=20*4/13−6↑4, the 6↑4 would be calculated to be 1296. The expression would then read:

A=20*4/13−1296

The next thing checked for would be items that are to be multiplied or divided. It does not matter whether the multiplication or division is done first because the answer will be the same either way. Our expression contains 20*4/13. This is calculated to be 6.15384615. This causes our expression to be restated as:

A=6.15384615−1296

Lastly, any instances of addition or subtraction are performed. This would cause our expression to be resolved into:

A= −1289.84615

7-7. BEATING THE HIERARCHY

There are times when the hierarchy just discussed does not suit our purposes. For example the expression 3*2+2 would normally give 6+2 or 8 when calculated using the standard hierarchy. But what if we acutally wanted to add the twos together before multiplying by three. We would then get the answer 12. We can beat the hierarchy through the use of parenthesis.

Any portion of an expression found within parenthesis () will be reduced to a single value before doing any work outside the parenthesis.

This means that we can take the expression 3*2+2 and enclose the 2+2 in a pair of parenthesis: 3*(2+2). This will cause the 2+2 to be replaced by a 4 in the expression. The resulting answer will be 12.

When using parenthesis always make sure there are as many left parenthesis as there are right parenthesis. If you don't-when the program reaches the expression that contains the odd parenthesis, it will stop all execution and print the error message ?SYNTAX ERROR on the screen.

Parenthesis can also be used inside of parenthesis. This is called nesting. When the expression is evaluated, the subexpressions inside parenthesis will be worked from the inside outward. That

( ) up to 10 sets of nested ( )

is, the innermost set of parenthesis will be reduced to a single value first. Consider the following example:

((3*(2+2)+7)*4)

The 2+2 will be reduced first giving:

((3*4)+7)*4

The 3*4 will be reduced next giving:

(12+7)*4

The 12+7 will next be added giving:

19*4 or 76

The same expression with no parenthesis would equal 36.

ONE FURTHER NOTE ON EXPRESSIONS

You can use an expression any place in your program that the computer would normally expect to see a number except as a program line number. Consider the following example:

```
100 A=2
110 FOR C=1 TO 3*(4-A)
120 PRINT C
130 NEXT C
140 STOP
```

7-8. A PRACTICAL PROBLEM

Let's take a standard formula used in everyday business and convert it for use on the Commodore computer.

In order to calculate monthly mortgage payments on a home loan, three figures must be known. They are: amount of loan, interest rate, and length of loan.

The formula is as follows:

$$\text{Payment} = \frac{AI}{1-(1+I)^{-N}}$$

Where: A = loan amount
I = monthly interest rate expressed as a decimal (less than 1)
N = length of loan in months

Let's program this formula for a $44500 loan at 11.5% interest for 30 years.

```
100 A=44500
110 I=.115/12
120 N=30*12
```

Note that interest is expressed in decimal form and is divided by 12 to give monthly rather than annual interest rate. The length of loan is expressed in months (30 years * 12 months per year).

In this formula, the part above the line is AI. This indicates that A should be multiplied by I. For the computer this is A*I. The horizontal bar of course indicates a division operation. On the computer we use the slash / to perform division. So the top part of this formula is equivalent to A*I/.

The bottom part of the formula is what A*I is to be divided by so we will put the whole thing inside parenthesis. Further, the (1+I) portion of the expression is raised to a negative power of N. Recall that the up arrow (↑) is used to raise numbers exponentially. The bottom portion of the formula is then equivalent to:

```
(1-(1+I)↑-N)
```

The entire expression would then be:

```
A*I/(1-(1-I)↑-N)
```

We would then add this to our program storing the result of the calculation in the variable PMT (line 130) and printing PMT on the screen along with a dollar sign (line 140).

```
100 A=44500
110 I=.115/12
120 N=30*12
130 PMT=A*I/(1-(1+I)↑-N)
140 PRINT"$";PMT
```

This program then should calculate the monthly mortgage payments on a $44500 loan and print the result on screen.

Assignment

After paying on a home loan for several years, it might be nice to know how much is still owed on the loan. The formula for calculating loan balance is shown below.

LOAN BALANCE =

$$\left[\frac{1}{(1+I)^{-N}}\right]\left[P\left[\frac{(1+I)^{-N}-1}{I}\right]+A\right]$$

Where: A = original loan amount in dollars.
I = monthly interest rate expressed as a decimal percentage.
N = total number of payments already made.
P = monthly mortgage payment in dollars.

Write a program to calculate the loan balance on a $50000 loan at 12% annual interest. Monthly payments of $514.30 have been made for nine years (108 months). Use the formula above to write this program. A sample solution may be found in APPENDIX A of this manual.

LESSON 8

Relational Operators

We have learned in previous lessons that computers can remember fairly trivial bits of information quite accurately, and it can perform boring repetitive tasks quite reliably. However, to do really useful work it must be able to make basic decisions based on the data contained in memory.

Relational operators allow the computer to make such basic decisions as: which of two numbers is larger or whether or not two numbers are equal. The computer also has the ability to jump to a particular section of the program if a certain condition is true. This lesson describes these decision making abilities and how they can be used in Basic language programs.

8-1. THE IF/THEN STATEMENT AND EQUALITY

The IF/THEN statement is probably the most powerful decision maker in the Basic language arsenal. Generally this statement is of the form: IF condition X is true THEN do operation Y. It can be used in combination with almost any other Basic keyword.

The IF portion of the statement is followed by an expression which can include variables, strings, numbers or logical or relational operators. The word THEN appears on the same line and is followed by a line number or one or more Basic keywords. When the expression following the IF word is false, everything after the word THEN is ignored and execution continues with the next line number in the program.

If the result of the expression following IF is true, the program jumps to the line number that follows the THEN or performs whatever Basic statements follow it. Consider the following example:

```
100 A=1
110 IF A=2 THEN 150
120 PRINT "A DOES NOT EQUAL TWO"
130 A=2
140 GOTO 110
150 PRINT "NOW IT DOES"
```

In this example, line 100 sets A equal to 1. Line 110 checks to see if A equals 2. Since it does not, we continue at line 120 by printing A DOES NOT EQUAL TWO. Line 130 sets A equal to 2 and line 140 causes a jump back to line 110. This time A does equal 2. The line number after THEN is 150 so the program to line 150 and prints NOW IT DOES.

8 - 2. THE LESS THAN OPERATOR

The shifted comma key prints the symbol (<). This symbol represents the LESS THAN operation. This symbol is normally used in IF/THEN statements as follows:

```
100 IF A<15 THEN 150
```

The program will jump to ine 150 only if the value held in A is less than 15.

8 - 3. THE GREATER THAN OPERATOR

The shifted period key produces the (>) symbol. This symbol is used in IF/THEN statements to determine if one number is larger than another.

```
100 IF A>15 THEN 150
```

In the statement above, the program will jump to line 150 only if A is 16 or greater.

8 - 4. INEQUALITY

It is also possible to determine that two numbers are simply not equal by using the less than and greater than operators in combination.

```
100 IF A <>34 THEN 150
```

This statement will go to line 150 if A is either greater than or less than 34. In effect, this determines that A is not equal to 34.

We can carry this concept of combining relational operators such as , and = a step further. We can combine any two relational operators in a statement to obtain the desired result. The entire range of relationships we can test for is listed below:

=	EQUAL TO
<	LESS THAN
>	GREATER THAN
>< or <>	GREATER THAN OR LESS THAN
<=or =<	LESS THAN OR EQUAL TO
>=or =>	GREATER THAN OR EQUAL TO

Note that when combining two operators, it makes no difference which comes first on the line. The relation will hold true if either of the operators used holds true.

8 - 5.
USING RELATIONAL OPERATORS WITH STRING DATA

Relational operators can also be used to compare two strings of text. This may sound odd since we are not used to thinking of one text word as being greater than, less than, or equal to another. The computer has no trouble with this concept at all. When making a decision as to which string is greater, the computer compares each character of the two strings one character at a time from left to right. Letters at the beginning of the alphabet are considered to be LESS THAN the letters that occur later in the alphabet. Under this system the letter A is considered to be less than the letter B. In this way text strings are compared alphabetically much as numbers were compared by numerical order in our previous examples. Consider the following example:

```
100 A$="JACK"
110 B$="JASON"
120 IF B$<A$ THEN PRINT B$
130 IF A$<B$THEN PRINT A$
```

In line 120 of this example the computer compares the first character of each string to determine which string is least. Since the first character of each string is the letter J the computer must try another comparison with the second character. Both strings have the letter A as the second character and so again no conclusion can be reached. When the comparison is performed on the third character, it is clear that the letter C in A$ is less than the letter S in B$. Therefore, A$ is less than B$ and the IF/THEN statement

in line 120 is untrue. The PRINT statement in line 120 would be ignored and the program would continue at line 130. Line 130 performs exactly the same comparison. Since the IF/THEN statement in line 130 is true in this case, the PRINT A$ statement is performed.

It should be noted that for two strings to be considered equal (A$ = B$), all characters must match exactly between the two strings.

8-6. THE ON GOTO STATEMENT

The IF/THEN statement is not the only decision making statement in the Basic language. The ON GOTO statement can make some simple decisions based on the numerical value of a variable. For some operations it is even more efficient than the IF/THEN statement.

The ON portion of the statement is followed by a variable or expression containing a variable. The GOTO portion of the statement is followed by a list of line numbers. The result of the variable expression will indicate which program line number from the list that the program will jump to. Consider the statement: ON A GOTO 100, 200, 300. If A equals 1, the program will jump to the first number on the list (100). If A equals 2, the program would jump to the second number on the list (200) etc.

If A were greater than the quantity of line numbers contained in the list or if A equals 0, the entire ON GOTO statement will be ignored and the next line number in the program will be executed. Note that all the line numbers in the ON GOTO statement are separated by commas.

The ON GOTO statement can replace a number of IF/THEN statement lines with a single program line. Consider the following example:

```
100 A = 2
110 IF A = 1 THEN 300
120 IF A = 2 THEN 400
130 IF A = 3 THEN 500
300 PRINT "ONE"
310 END
400 PRINT "TWO"
410 END
500 PRINT "THREE'
510 END
```

This example executes a print statement determined by the value held in A. We can replace lines 110, 120 and 130 with one ON GOTO statement as follows:

```
100 A=2
110 ON A GOTO 300, 400, 500
300 PRINT "ONE"
310 END
400 PRINT "TWO"
410 END
500 PRINT "THREE"
510 END
```

The two program examples give exactly the same result.

One final word on the ON GOTO statement. By replacing the GOTO keyword with the GOSUB keyword you can create the ON GOSUB statement. It works the same way except that instead of just jumping to program lines, the ON GOSUB statement calls subroutines.

LESSON 9

Logical Operators

Logical operators (sometimes known as Boolean operators) are most commonly used to modify the meaning of statements containing relational operators.

There are three logical operators used in the Basic language:

AND
OR
NOT

This lesson will describe the use of these three operators.

9-1. THE AND OPERATOR

The AND operator is used to combine two relational statements and determine if they are both true. Consider the following example:

```
100 A=4
110 B=17
120 IF A=4 AND B=17 THEN 150
130 PRINT "NOT EQUAL"
140 END
150 PRINT "BOTH A AND B ARE CORRECT"
160 END
```

In the above example, since both relational statements in line 120 are true, the IF/THEN statement will cause a jump to line 150 and BOTH A AND B ARE CORRECT will be printed on screen. If either of the statements in line 120 had been false, line 130 would have been executed instead.

Let's try the same example with a different value in variable A:

```
100 A=2
110 B=17
120 IF A=4 AND B=17 THEN 150
130 PRINT "NOT EQUAL"
140 END
150 PRINT "BOTH A AND B ARE CORRECT"
160 END
```

Since A does not equal 4 the entire IF/THEN statement in line 120 considered false. The AND operator requires that both relational expressions in the statement be true.

9-2. THE OR OPERATOR

The OR operator is used much like the AND operator was in the last topic. However, the OR operator will recognize the statement as true if either of the two expressions is true.

```
100 A=20
110 FOR B=1 TO 7
120 IF A<=15 OR B<=3 THEN 200
130 PRINT "FALSE"
140 A=A-1
150 NEXT B
160 END
200 PRINT "TRUE"
210 GOTO 140
```

The first three times through this loop (lines 110—150) the B variable will be 3 or less causing a jump to 200 and TRUE to be printed. During the next two times through the loop, neither expression in line 120 will be true and FALSE will be printed. During the sixth and seventh iterations of the loop, the A variable will be 15 or less causing TRUE to be printed again. This example demonstrates that statements containing OR will be true if either expression is true.

NOTE: The loop ends after 7 times because line 110 sets the variable B from the 1 to 7.

9-3. THE NOT OPERATOR

The NOT operator is used to reverse the normal true/false result of relational expressions. Unlike the AND and OR operators which

affect two expressions, the NOT operator only affects the expression to the right of it. Consider this example:

```
IF NOT A<B OR C=>7
```

In this example the expression A<B is modified by NOT so that its result is true only if A is equal to or greater than B. The expression C=>7 is not affected at all. The overall statement will be true if A is equal to or greater than B or if C is equal to or greater than 7.

LESSON 10

Keyboard Input

Most programs require the user to enter information through the keyboard in response to queries printed on the screen. This ability to accept information on a more or less informal basis is a great advantage over computers of earlier times that required all data to be fed in batches on punched cards or paper tape. This lesson describes how to build programs that accept keyboard input.

10 - 1. THE INPUT STATEMENT

The easiest method of handling keyboard input is by using INPUT statements to define variables. When the program reaches a line containing an INPUT statement, it stops any further program execution and prints a question mark (?) on the screen. This indicates that the program is waiting for data to be entered. Data is entered by typing in the number or text and pressing the RETURN key. Program execution then continues with the next program line. Consider the following example:

```
100 PRINT "ENTER YOUR NAME"
110 INPUT N$
120 PRINT "HELLO ";N$
130 END
```

In this example N$ was defined by keyboard input. In this case we defined a string variable, but integer and floating-point variables are treated the same way. However, if string text is entered in response to an INPUT statement for a numeric variable, a data type mismatch occurs. This is because we cannot store string data in a numeric variable. The advisory ?REDO FROM START will appear on screen and the data must be reentered.

10 - 2.
USE OF PROMPT MESSAGES IN INPUT STATEMENTS

In the first line of our last example we printed the message "ENTER

YOUR NAME" on the screen so that the user would know what he was expected to enter in response to the question mark on the screen. Actually, it is not necessary to devote a program line to this task. We can combine this message with the input statement as a prompt. This statement would be of the form:

```
100 INPUT "PROMPT";N$
```

Note that the prompt is enclosed in quotation marks and separated from the variable N$ by a semicolon. Let's take a look at another example:

```
100 INPUT "ENTER FIRST NUMBER";A
110 INPUT "ENTER SECOND NUMBER";B
120 C=A*B
130 PRINT A;"TIMES";B;"=";C
```

This example will input two numbers, calculate their product and print the result.

10 - 3. THE GET STATEMENT

The INPUT statement works quite well for entering data but there are times when we just want to monitor the keyboard to see if any of the keys have been pressed and if so, which one. The GET statement does this quite well.

The computer contains a specialized area of memory called the keyboard buffer. This buffer will hold up to 10 keystrokes. The GET statement will go get the first key stored in this buffer and place it in whatever variable you specify in the GET statement. It is best to always use string variables for this purpose as they can store any character that is available from the keyboard. Consider the following example:

```
100 GET A$
110 IF A$="" THEN 100
120 PRINT A$
130 GOTO 100
```

In this example line 100 contains the GET statement. This line will retrieve the first character in the keyboard buffer and store it in the variable A$. Line 110 checks to see if a key was in fact pressed. If not it jumps back to line 100. In effect, this creates a small loop that runs until a key is pressed. Once a key is pressed, we break

out to line 120 which prints whatever key is stored in A$. Line 130 causes a jump back to line 100 to get the next keystroke. In this way, each key that is pressed is printed on screen.

10 - 4. THE FUNCTION KEYS

Along the right side of the keyboard are four keys which are marked f1 through f8. These keys are called function keys which is somewhat comical since they have no function.

They are provided for you to use in your programs in any way that you like. To use these keys you must first define them. In order to define them you need to be able to tell when one of them was pressed. To do that you need to know what character they produce. Surprise - they don't produce any characters. Even stranger - none of the other keys produce characters either.

All the keys on the Commodore keyboard produce numbers. These numbers represent characters in some cases and various computer functions in others. A table that lists these numbers and their meanings is provided in Appendix F of the Commodore 64 Users Guide that came with your computer. The function keys do have their own numbers assigned to them but the numbers do not represent any printable characters.

Notice that the top function key has two f numbers marked on it. The top of the key is marked f1 and the front edge of the key is marked f2. The key acts as f1 when pressed by itself and as f2 when the SHIFT key is held at the same time. The following table lists the ASCII numbers that correspond to each of the function keys.

f1	133	f2	137
f3	134	f4	138
f5	135	f6	139
f7	136	f8	140

The keyword CHR$ will convert any Commodore ASCII code number to its character equivalent. We can use this in conjunction with the GET statement to define our function keys.

```
100 GET A$
100 IF A$=""  THEN 100
120 IF A$=CHR$(135)  THEN GOTO 200
130 IF A$=CHR$(136)  THEN GOTO 300
140 GOTO 100
200 PRINT  "F5 KEY PRESSED"
210 END
300 PRINT "F7 KEY PRESSED"
310 END
```

In this example, we monitored the status of function keys f5 and f7 by checking to see if A$ was equivalent to CHR$(135) or CHR$(136). While in this example we simply printed the name of the key that was pressed, we could have just as easily jumped to a section of program that performed a much more exotic function. In this way we can use the function keys (or any other keys) to allow the user to select the portion of the program he would like to run.

10 - 5. MENUS

When writing a computer program, the first thing you should determine is what you would like the program to do. Take for example a program to maintain your checking account. You would probably want the program to allow you to enter checks as you write them. You might also want it to search for all checks written to a particular person or business and list them out on the screen. Perhaps you want to be able to categorize these checks so you can print out all expenditures made in a particular category for tax purposes. You may even want this program to list out all checks written in a certain time period. Basically, you should list all the general functions you would like to program to perform. Such a list might appear like this.

1. WRITE CHECKS
2. SEARCH CHECKS BY NAME
3. SEARCH CHECKS BY CATEGORY
4. SEARCH CHECKS BY DATE

What you have created in making this list is a program menu. You can write Basic program lines that will print this list on the screen as the first action performed by your program.

Certainly you could be more creative by adding a title to the screen, some color, directions to your house or whatever you like, but basically a menu is a list of functions that your program is capable of performing (or will be when the program is completed).

The next section of the program would be a keyboard monitoring routine such as that described in topic 10-3 or 10-4 of this lesson. You could use the numbered keys 1 through 4, function keys, or whatever keys you like to represent the items listed on your menu. However, rather than just print the key that was pressed as we did in our topic examples, use the keyboard monitor to branch to widely separated program line numbers. Consider the following example:

```
100 GET A$
110 IF A$="" THEN 100
120 IF A$="1" THEN 1000
130 IF A$="2" THEN 2000
140 IF A$="3" THEN 3000
150 IF A$="4" THEN 4000
160 GOTO 100
```

Once the menu is printed on the screen, the keyboard monitor above will cause the computer to patiently wait until one of the keys numbered 1-4 is pressed. The program will then branch to line 1000, 2000, 3000 or 4000, depending on which key is pressed.

You now have the bare framework of a BASIC program. Line 1000 could be the beginning of a routine to enter new checks. Line 2000 could be the beginning of a routine to search the check file by name and so on. In this way you can use menus to give your programs some structure and organization even before the bulk of it has been written. While not all programs lend themselves to a menu format, (games for example) most do and you will find that many business and home programs currently on the market use this menu driven approach. Most of all, by doing the menus first, you will create a clear picture in your mind of what you want your program to do.

A PRACTICAL PROBLEM

Recall from an earlier lesson that we had developed a program to calculate monthly mortgage payments for a $44000 dollar 30 year loan at an 11.5% interest rate. This is fine as far as it goes, but what if the loan we want to calculate is for a different amount.

We need to replace all those statements that assigned values to the variables with statements that input the variable values from the keyboard. In that way the program can be used to calculate any

mortgage value at any interest rate for any loan period. This is shown in the example below:

```
100 INPUT"ENTER LOAN AMOUNT";A
110 INPUT "ENTER ANNUAL INTEREST"; I
120 I = I/1200
130 INPUT"ENTER NO. OF YEARS";N
140 N = N*12
150 PMT = A*I/(1 - (1 + I)↑ - N)
160 PRINT"$";PMT
```

In this example line 100 prints the query "ENTER LOAN AMOUNT" on screen and waits for a keyboard input which is stored in variable A. After the loan amount has been entered, line 110 prints "ENTER ANNUAL INTEREST" on screen and waits for a keyboard entry defining the variable I. Since I is used as monthly interest in the program and must be expressed as a decimal percentage, line 120 divides I by 12 (months) times 100 (to make a decimal percentage) or 1200.

Line 130 is used to input the number of years the loan will run (variable N) and line 140 converts years to months. Line 150 does the calculation and line 160 prints the resulting monthly payment with a dollar sign. In this way we can use INPUT statements to convert a rather rigid one case program to an interactive program.

ASSIGNMENT

Recall from lesson 7 the program you wrote to calculate the loan balance remaining on an existing mortgage. Convert this program to be interactive much as we did in the last example.

Once the mortgage payment routine and loan balance routine are working to your satisfaction, create a menu and keyboard monitor that will let you select either of the two routines from the keyboard. Example solutions are provided in Appendix A of this manual.

LESSON 11

String Handling

Up to this point you have learned that the STRING is the data type used to store text information. Text storage and manipulation is one of the most common uses for the home microcomputer. Word processing programs, data base programs and really almost any non-game application requires that text be stored, displayed, copied, moved, sorted or rearranged to a greater or lesser degree. This lesson will describe some of the Basic language techniques commonly used to handle strings.

11 - 1.
CONVERTING BETWEEN STRING AND NUMERIC DATA

Numeric data items can only be compared (or assigned) to other numeric items. Likewise string data items can only relate to string type data items. At times this can prove to be quite inconvenient. We may, at times, want to mix data from a numeric variable with a message contained in a string variable. In that case, we must first convert one of the data types to the other. If we do not, the program will stop with the Basic error message:

?TYPE MISMATCH

We can convert numeric data to string data using the STR$ statement. Likewise, the VAL statement converts string data to numeric data.

The STR$ statement simply takes the numeric data from a numeric variable and assigns it to a string variable. Consider the example below:

```
100 F = 27
110 PRINT F
120 B$ = STR$(F)
130 PRINT B$
```

Line 100 assigns the numeric floating-point variable F a value of 27 and line 110 prints the value of F on the screen. Line 120 converts the numeric variable F to string format and stores it in B$. Note that in line 120 the F variable is contained within parenthesis immediately following the STR$ keyword. Both variables appear the same when printed on screen.

The value 27, which is now held in B$, can be combined with other string data. However, we cannot perform any arithmetic functions on it until we convert it back to a numeric data type.

The VAL statement is used to convert numbers contained in strings to a numeric data type as shown below.

```
100 F=VAL(B$)
110 PRINT F+3
120 PRINT B$
```

Note that if the first non-blank character of the string is not a plus sign, a minus sign, or a numeric digit, the VAL statement will return a value of zero. However, you can evaluate strings that contain text as long as they start with a number. The VAL statement will evaluate each character, from left to right until it reaches either the end of the string or a non-digit character. Consider this example:

```
100 B$="125 POUNDS"
110 F=VAL(B$)
120 PRINT F
130 PRINT B$
```

Note that the POUNDS text contained in B$ does not keep the VAL statement from correctly deriving the number 125 from the string because it follows the number 125 rather than preceding it.

11 - 2.

CONVERTING BETWEEN STRINGS AND ASCII CODES

As we mentioned at the beginning of this tutorial, the microprocessor contained in the computer can only deal with numbers. All characters contained in strings or entered from the keyboard are actually numbers that represent characters. Normally, the Basic language interpreter automatically converts these numbers to character equivalents before displaying them on the screen. At times we need to determine what the numbers themselves are. At other times we may have a number but we need to know the character equivalent.

As a matter of interest, the number system that establishes the relationship between numbers and characters is called the ASCII code. ASCII stands for American Standard Code for Information Interchange. Since most computers use this system, it is possible to send data between two computers of different manufacture.

On the Commodore computer we can obtain the ASCII code for a given character using the ASC statement. Likewise, we can translate an ASCII code number to a character equivalent using the CHR$ statement.

The ASC statement is used to convert a string character to its ASCII code equivalent. Consider this example:

```
100 F=ASC("A")
110 PRINT F
```

This example converts the literal character A to its ASCII equivalent and stores the result in variable F. Note that the A is in quote marks indicating that it is a literal character. The A and its quote marks are enclosed in parenthesis immediately following the ASC keyword. Line 110 prints the ASCII code equivalent of the letter A (in this case 65) on the screen.

We can also obtain the ASCII code for characters held in string variables. Consider the following example:

```
100 X$="MILDRED"
110 F=ASC (X$)
120 PRINT F
```

In this example, the ASC statement returns the code for the first character of X$. The code 77 represents the character M. You will find a complete table of ASCII codes in the back of the Commodore 64 User's Guide.

We can reverse the process of ASCII conversions by using the CHR$ statement. The CHR$ statement gives us the character equivalent for any given ASCII code. Let's try another example:

```
100 X$=CHR$(77)
110 PRINT X$
```

In this example, line 100 converts whatever number or numeric variable is held inside the parenthesis to a character and stores it in X$.

11 - 3. THE LEFT$ STATEMENT

At times it can be quite useful to be able to cut up a string variable and use part of it for something else. The LEFT$ statement allows us to remove a portion of a string starting from the left most character (beginning) of the string. We must also specify what length this substring we are removing is to be.

The LEFT$ keyword is always followed by a set of parenthesis which contain the name of the string we are cutting up and the length of the substring. Consider the following example:

```
100 A$ = "PRODUCTION"
110 B$ = LEFT$(A$,7)
120 PRINT B$
```

In this example we copied the first 7 characters of A$ (PRODUCT) into B$. If we had specified a length greater than the length of A$, the entire contents of A$ would have been stored in B$.

Notice that the length (7) and the source string name (A$) are held in the parenthesis and are separated by a comma.

11 - 4. THE RIGHT$ STATEMENT

The RIGHT$ statement functions almost exactly like the LEFT$ statement except that the substring is counted from the right most (end of string) character. Again the number of characters to be used from the original source string must be specified as shown in the example below:

```
100 A$ = "MICROPROCESSOR"
110 G$ = RIGHT $(A$,9)
120 PRINT G$
```

In this example, the last nine characters of A$ (PROCESSOR) will be copied from A$ into G$ and subsequently printed on screen.

11 - 5. THE MID$ STATEMENT

The MID$ function returns a substring which is taken from a larger string just as the LEFT$ and RIGHT$ functions did. However, while the LEFT$ function counted out the substring length from the left, and the RIGHT$ function counted from the right most character, the MID$ function counts from any character position in the string. In this way the MID$ function allows us to take a string of any length from any section of a larger string. But we must specify

both the starting position and substring length in the MID$ statement as shown in the following example:

```
100 A$="ACCOUNTING SOFTWARE"
110 G$=MID$(A$,3,5)
120 PRINT G$
```

Note that in line 110, the first number to follow A$ in parenthesis (3) specifies the starting position of the substring. The second number (5) gives the length of the substring. As a result, all characters from the third to the seventh character of A$ is copied into G$ and subsequently printed on screen.

If the position numeric is greater than the total length of the source string, or if the length numeric is zero, the resulting substring will be empty. It is interesting to note that the length numeric may be left out. In that case, the substring will consist of all characters from the specified starting position to the end of the source string.

11 - 6. DETERMINING STRING LENGTH

At times it can be very handy to know the number of characters contained in a string. The LEN statement is used to determine the length of a string. Using this function is quite easy as shown below:

```
100 A$="SAINT LOUIS MISSOURI"
110 PRINT LEN(A$)
```

This example will count all characters in A$, including punctuation and spaces, and print the result (20) on screen.

11 - 7. STRING CONCATENATION

We stated earlier that no arithmetic functions could be performed on string data. That is not entirely true. As a matter of fact, strings may be added together. It is not handled exactly like a normal arithmetic function however.

If we add one string to another using the plus sign (+), the second string is simply appended to the first. This is called CONCATENATION. The process is more aking to linking two items than it is to adding them together. Consider the following example:

```
100 A$="CONCAT"
110 B$="ENATION"
130 C$=A$+B$
140 PRINT C$
```

This example splices the ENATION STORED IN B$ onto the end of CONCAT in A$ and stores the result in C$. The CONCATENATION string held in C$ is then printed on screen.

It is important to remember that when we use various string handling statements to remove sections of a string and store them in another string variable, the source string is left unchanged. We are only copying the section.

ASSIGNMENT

Write a short program that will allow the user to type in a single sentence of text. The program should then print on screen the number of words in the sentence, the number of nonspace characters in the sentence, and the average number of characters per word. Use the GET statement to input the sentence rather than the INPUT statement. Use the MID$ statement to do the counting. A sample solution may be found in Appendix A of this manual.

LESSON 12

Mathematics

The Basic language supports a number of higher math functions such as SIN, COS, TAN, etc. Many other functions can be derived using the functions supplied. This lesson describes each of these functions as they are used in the Basic language but does not attempt a course in higher mathematics.

A mastery of the Basic functions described in this lesson is not required to write good Basic programs. If you do not completely understand all the material presented in this lesson, do not be the slightest bit concerned. For those familiar with algebra and trigonometry, this lesson describes the Basic language elements necessary to do useful work in those disciplines.

12-1. ABSOLUTE VALUE

The ABS statement is used to determine the absolute value of a number. The absolute value of any number is that number with the signs removed. Consider this example:

```
100 F = -2376.45
110 G = ABS(F)
120 PRINT G
```

Although variable F is assigned a negative number in line 100, the value assigned to G will appear to be positive since it does not carry a sign indicating whether it is positive or negative.

The SGN function does not alter the value of a number. Rather it will return a value indicating the sign of a number. The value returned will be a −1 if the number is negative, a +1 if the number was positive, and 0 if the number was zero. Consider the example following:

```
100 F = -385
110 ON SGN (F)+2 GOTO 200,300,400
200 PRINT "NEGATIVE"
210 END
300 PRINT "ZERO"
310 END
400 PRINT "POSITIVE"
410 END
```

In this example we assign a negative number to variable F. Line 110 is an ONGOTO statement that detects if a number is negative, positive or zero and jumps to the appropriate print statement. Note that the result of the SGN statement in line 110 will be −1, 0, or +1. By adding 2 to this result it will become 1, 2 or 3 which is usable in an ONGOTO statement.

12 - 2. THE SQR FUNCTION

The SQR function returns the square root of its numeric argument. Be aware that if this function is applied to any negative number, the Basic error message ?ILLEGAL QUANTITY will appear on screen. Consider this example:

```
100 F = SQR(25)
110 PRINT F
```

Line 100 assigns the square root of 25 to the variable F and line 110 prints F (in this case 5) on the screen.

12 - 3. TRIGONOMETRIC FUNCTIONS

There are four trigonometric functions for which a Basic keyword exists as shown below:

FUNCTION	KEYWORD
SINE	SIN
COSINE	COS
TANGENT	TAN
ARCTANGENT	ATN

The value returned as a result of any of these functions is expressed in radians. Consider the following example:

```
100 F = COS (14)
110 PRINT F
```

Line 100 assigns the cosine of 14 to the variable F and line 110 prints F on screen.

12 - 4. LOGARITHM

The Basic language used on the Commodore 64 computer contains two statements that pertain to logarithmic calculations. They are the LOG and EXP statements.

The LOG statements will calculate the natural logarithm (base e - 2.71828) of any positive number greater than zero as shown below:

```
100 F = LOG(32.8)
110 PRINT F
```

The natural logarithm of 32.8 (3.49042852) will be stored in variable F. If a LOG function is applied to a negative number or zero, the Basic error message ?ILLEGAL QUANTITY will appear.

The EXP statement reverses this process by calculating the original number given the natural logarithm. Consider the example below:

```
100 F = EXP(3.49042852)
110 PRINT F
```

Line 100 calculates the constant e (2.71828) raised to the power of 3.49042852 and stores the result (32.8) in the variable F. Note that applying the EXP function to any number larger than 88.0296919 will cause the Basic error message ?OVERFLOW to appear on screen.

Many times mathematical calculations are required using common logarithms (base 10) rather than natural logarithms (base e - 2.71828). Common logarithms can be derived from the natural logarithm function using the following formula:

```
Common logarithm of X = LOG(X)/LOG(10)
```

12 - 5. PI

The number PI (3.1416) is available from the keyboard and may be used in any arithmetic expression. The up arrow (↑) key produces the PI numeric when shifted. On screen, the normal PI symbol will not appear but rather a graphic symbol (▚). This symbol represents the number 3.1416 to the Commodore 64 computer.

12 - 6. DERIVING TRIGONOMETRIC FUNCTIONS

In topic 12-3 we discussed the trigonometric functions supplied with the Basic language. Many other functions can be derived using different combinations of the functions provided. The table below gives formulas that may be used to produce additional funtions.

FUNCTION	BASIC EQUIVALENT
SEC(X)	1/COS(X)
CSC(X)	1/SIN(X)
COT(X)	1/TAN(X)
ARCSIN(X)	ATN(X/SQR(−X*X+1))
ARCCOS(X)	−ATN(X/SQR(−X*X+1))+PI/2
ARCSEC(X)	ATN(X/SQR(X*X−1))
ARCCSC(X)	ATN(X/SQR(X*X−1))+(SGN(X)−1*PI/2)
ARCOT(X)	ATN(X)+PI/2
SINH(X)	(EXP(X)−EXP(−X))/2
COSH(X)	(EXP(X)+EXP(−X))/2
TANH(X)	EXP(−X)/(EXP(X)+EXP(−X))*2+1
SECH(X)	2/(EXP(X)+EXP(−X))
CSCH(X)	2/(EXP(X)−EXP(−X))
COTH(X)	EXP(−X)/(EXP(X)−EXP(−X))*2+1
ARCSINH(X)	LOG(X+SQR(X*X+1))
ARCCOSH(X)	LOG(X+SQR(X*X−1))
ARCTANH(X)	LOG((1+X)/(1−X))/2
ARCSECH(X)	LOG((SQR(−X*X+1)+1/X)
ARCCSCH(X)	LOG((SGN(X)*(SQR(X*X+1/X)))
ARCCOTH(X)	LOG((X+1)/(X−1))/2

12 - 7. SCIENTIFIC NOTATION

Scientific notation is a method of stating numbers that are either very large (24385926) or very small (0.00000237). In scientific notation a number is made up of 3 parts: the mantissa, the letter E, and the exponent.

The mantissa is simply a floating-point number, the letter E indicates that the number is in exponential form, and the exponent is the power of 10 that the mantissa is raised to. Consider the number 28000. This number could be restated in scientific notation as:

2.8E4

This number would be read as two point eight times ten to the fourth power. Basically what we have done is shift the decimal point in 28000 four places to the left.

We can also use scientific notation to express numbers smaller than one. For example: the number 0.0025 would be expressed as 2.5E-3 in scientific notation. The mantissa (2.5) is multiplied times ten to the −3 power. Notice that in this example the decimal point was shifted 3 places to the right.

There is a limit to the size of numbers that the Basic language can handle even using scientific notation. Any number larger than 1.70141183E38 will cause the Basic error message ?OVERFLOW ERROR to appear.

12-8. DEFINING FUNCTIONS

In writing programs you may find that a particular mathematical expression is repeated a number of times in a program. To save memory you can simply define this function one time and use it thereafter. The Basic statements that allow this are DEF and FN. You must name the function. Thereafter you may call the function by name much as you would a variable. Consider this example:

```
100 DEF FN C   (R)=3.14*R↑2
110 PRINT FN C(23.54)
```

In this example we defined a function C to be equal to the formula PI times R squared. This formula calculates the area of a circle from a given radius (R). After defining the function we can call it as we did in line 110. We must supply the number in parenthesis (in this case 23.54) that we want the function performed on.

In this way any mathematical function may be defined once and called from anywhere in the program as many times as needed.

12-9. THE INT FUNCTION

Most mathematics will be performed on floating-point numbers. However, there are times when integer numbers might be more desirable in the result. We can convert floating-point numbers to integer format using the INT function. This function rounds off a number down to the nearest whole number by dropping any fractional parts. Consider the following example:

```
100 F=32.587
110 G=INT(F)
120 PRINT G
```

In line 110 the number 32.587 will be rounded down to the integer value 32 which is stored in variable G. The contents of G are then printed on screen.

12-10. RANDOMIZATION

One of the more popular uses for the Commodore 64 computer is game playing. But up to this point nothing we have discussed would lead anyone to believe that computers were very creative. How then, do the games keep from becoming repetitive? How does the computer avoid dealing the same Blackjack hand over and over?

The RND function generates a random number between 0 and 1 that may be used to determine what card is dealt or where the alien ships will show up next on the screen. The RND statement, like many other Basic statements, is followed by a numeric argument in parenthesis. The important thing about this argument is not what number it contains but rather whether the number is positive, negative or zero.

NEGATIVE NUMBER: If the number in the argument is negative, it is used to reseed the random number generator. The seed is the number the computer performs calculations on to produce random numbers. It is interesting to note that the number returned by the RND statement will always be the same for a given negative number. For example: the statement RND(−3) will always produce the number 4.48217179E−08. Obviously this is not very random. Negative arguments are used mainly to reseed the random number generator.

POSITIVE NUMBERS: If the numeric argument is positive, a random number between 0.0 and 1.0 will be generated using the stored seed. The interesting thing about positive numbers is that by using the RND function several times, a sequence of numbers will be developed that has a repeatable pattern. All that is needed to repeat the sequence is to reseed with the same negative number and then use a positive argument to call up the random numbers. Consider this example:

```
100 FOR N=1 TO 2
110 X=RND(-3)
120 FOR L=1 TO 5
130 PRINT RND(+2)
140 NEXT L
150 PRINT
160 NEXT N
```

In this example the same random number sequence is run twice by reseeding with −3 each time (line 110) and calling a positive RND STATEMENT FIVE TIMES (line 130). The pattern sequence of five numbers will be identical each time run.

RND USING ZERO: The RND(0) statement operates a little bit differently in that it uses a clock that is built into the Commodore computer for the random number seed. This then is the only truly random number function available. There is no way to predict what number will be generated. The only thing we can be assured of is that the result will be between 0.0 and 1.0.

Rarely will you find a number that is between one and zero useful. Usually you will want a random number that falls in a certain range between two whole numbers. The following formula will generate a random number (N) between a lower limit (X) and an upper limit (Y).

```
N=RND(0)*(Y-X)+X
```

Let's use this formula to generate five numbers between 1 and 52 as shown in the example below:

```
100 FOR L=1 TO 5
110 N=RND(0)*(52-1)+1
120 PRINT N
130 NEXT L
```

This example will generate five random numbers between 1 and 52 and print them on the screen. However, the numbers are all floating-point numbers with long decimal fractions on them. We can remove these fractions using the INT function we learned earlier.

```
100 FOR L=1 TO 5
110 N=INT(RND(0)*(53-1)+1)
120 PRINT N
130 NEXT L
```

This example generates five random whole numbers. With this program we could build a primitive card game. Note that in line 110 the upper limit was changed from 52 to 53 when the INT function was added. This is because the INT function always rounds DOWN to the next whole number. If the upper limit was left at 52, the number 52 would never be generated.

ASSIGNMENT

Write a short program to calculate two random whole numbers between 1 and 6 representing the roll of a pair of dice. Print the numbers on the screen and allow the user to reroll the dice as many times as he likes by pressing any key on the keyboard.

LESSON 13

System Utilities

This lesson covers some odd Basic statements that really do not fit with any of the other lesson topics. Most of these statements do independant tasks such as check the time or check how much memory remains for use.

13 - 1. THE RUN STATEMENT

There are two basic ways to start a program. Use of the GOTO statement is one of them. Simply enter a GOTO command with the line number where you want the program to begin execution. The other means of starting a program is by entering the RUN command. This causes the computer to go to the first program line number in memory and begin execution.

The main difference between the two methods is that the RUN command resets all the variables to zero. Any data that was held in a variable will be cleared out and lost. You may specify a line number in a RUN command but unlike the GOTO statement, it will also work without one.

13 - 2. THE CLR STATEMENT

The CLR statement has no argument. It is used solely to clear out all variables from the program. Any data held in any variables or variable arrays will be lost forever. User defined functions are erased. The CLR statement will free up the memory used by the variables for reuse.

13 - 3. THE FRE STATEMENT

There are 38911 bytes of memory in the Commodore 64 computer available to hold Basic language programs. If your program tries to use more memory space than is available, the Basic error message OUT OF MEMORY will appear on screen and the program will stop. The FRE statement allows you to find out how much memory you have left at any one time.

The FRE statement does have a numeric argument in parenthesis but it does not matter what number you put into it. The FRE statement will return either a positive number indicating remaining memory, or a negative number to which you must add the number 65536 to determine remaining memory. The following example program will always calculate remaining memory whether the returned number is positive or negative.

```
100 X=FRE(0)-(FRE(0)<0)*65536
100 PRINT "MEMORY REMAINING=";X
120 PRINT"MEMORY USED=";38911-X
```

Note in line 100 the expression X=FRE(0)-(FRE(0)<0)*65536. The FRE(0)<0 portion will equal −1 if the result is negative and 0 if the result is positive. So the result of multiplying 65536 by that will equal 0 if the FRE(0) result was positive or −65536 if it was negative. Our original line 100 would then result in either FRE(0)−(−65536) if the result was negative or FRE(0)−0 if the result was positive. In this way we can determine remaining memory whether the FRE(0) statement returns a negative or positive number.

13-4. THE REMSTATEMENT

The REM statement doesn't do anything at all. REM stands for REMark and that is what this statement is used for. You may enter REM statements anywhere you like in your programs and they will not alter the program a bit. However, by using REM statements you can make your programs much easier to read and understand. Let's add one to the example from topic 13-3.

```
 90 REM PROGRAM TO CHECK MEMORY
100 X=FRE(0)-(FRE(0)<0)*65536
110 PRINT "MEMORY REMAINING=";X
120 PRINT "MEMORY USED=";38911-X
```

You may type any text you like after the REM keyword and it will be printed out whenever you list the program. In this way REMs can be used to serve as a reminder of what you had in mind when you were writing each section of the program.

13-5. KEEPING TIME

The Commodore 64 computer contains an onboard clock called the jiffy clock. This clock is set to zero when the computer is first powered up and is updated 60 times a second. However, it does not run at all when loading from or saving to the cassette tape device or disk drive.

There are two Basic language statements that are used to get information from this clock: the TI statement and the TI$ statement.

The TI statement simply returns the number of 1/60 of a second intervals that have occurred since power up.

The TI$ statement is much more useful than the TI statement. The TI$ statement returns a string that contains the number of hours, minutes and seconds that have occurred since power up.

```
100 PRINT TI$
```

This example statement will print a six digit number on screen. The left two digits of the number give the number of hours that have passed since you first turned on your computer. The middle two digits give the minutes and the final two digits give the seconds. So there is actually a digital clock built into the computer. In fact, you can actually set this clock to any time you like by assigning the string a value just as you would any other string. The following example will set the clock.

```
100 INPUT"ENTER HOURS";A$
110 INPUT"ENTER MINUTES";B$
120 INPUT "ENTER SECONDS";C$
130 TI$=A$+B$+C$
140 PRINT TI$
```

Once a numeric value is assigned to TI$ in this manner, the clock will then update TI$ on a regular basis. The example below prints TI$ on screen along with the conventional colon between the hours, minutes and seconds.

```
140PRINT LEFT$(TI$,2);":";MID$(TI$,3,2);":";RIGHT$(TI$,2)
150 FOR N=1 TO 900
160 NEXT
170 PRINT"[up]"
180 GOTO 140
```

Lines 150 and 160 make up a delay loop that keeps line 140 from printing the time more than once each second. The loop will execute 900 times before going on to line 170.

Line 170 is a major key to making this program work. After printing the time in line 140, the system naturally moves the cursor position down one line. Since we want to print the time on the same line repeatedly, we have to devise a means of moving the cursor back up one line. This is not as hard as it sounds.

Recall that the cursor can be controlled using the cursor control keys (bottom row - last two keys on the right). These cursor control key commands can be embedded within quotes in a PRINT statement. Simply press the SHIFT key and the CRSR ↑ KEY WHILE INSIDE THE QUOTES WHEN YOU ENTER THE PRINT statement. This will appear on screen as a reverse video capital Q but it will cause the cursor to move up one line when the program tries to print it. There will be a more detailed discussion of this concept in a later lesson.

LESSON 14

Program Storage

Unfortunately, when you turn off the computer any programs you have entered are lost. Without some means of program storage you would have to retype the entire program each time you used it. Fortunately, the Commodore 64 computer supports program storage on either cassette tape or floppy disc. This allows you to SAVE programs on either of these media and LOAD the programs back into the computer later on.

This lesson describes the Basic language commands used to SAVE and LOAD programs on cassette tape or floppy disc.

14 - 1. SAVING PROGRAMS ON CASSETTE TAPE

The SAVE command copies the program currently in memory onto the cassette tape or floppy disc. Once the save is completed the program remains in memory entirely unchanged. Programs are usually given a name in order to make the program easier to find later on. The name can be up to 16 characters long.

The SAVE command is usually entered in the direct program mode. A simple SAVE command is shown below:

```
SAVE "PROGRAM NAME",1
```

Note that the PROGRAM NAME is enclosed in quotation marks. The number 1 is shown following the name and separated from it by a comma. This is the device number. This number designates which device the program will be saved to. The Datasette recorder is device number 1 while the disc drive unit is device number 8.

Actually the device number may be left out of the statement. In that case the computer would automatically assume that the Datasette recorder was the desired device.

Once the SAVE command has been entered the computer will print the instruction PRESS PLAY & RECORD ON TAPE. Press

both the play and record keys on the Datasette recorder. The screen will blank while the SAVE operation is performed. This may take several minutes. When completed the READY advisory will appear on the screen. At this point, the program is stored on tape and also still held in the computer memory.

A secondary address may be added to the SAVE command as shown below:

```
SAVE A$,1,1
```

A secondary address of 1 will cause the program to save so that when reloaded it will load into the same memory locations it occupied originally. A secondary address of 2 will cause an end-of-tape marker to be stored on the tape after the program. When loading, this end-of-tape marker signals the computer that there are no further programs on that tape. A secondary address of 3 will cause the actions of 1 and 2 to both be performed. Note that in this example the program name was held in A$ rather than quotes. This is fine as long as A$ does not exceed the 16 character limit for program names.

14 - 2. SAVING PROGRAMS ON DISC

Saving programs on disc is similar to saving them on tape. The device number must be included in the SAVE command. The disc drive unit is designated device number 8. A typical SAVE command would appear as follows:

```
SAVE "PROGRAM NAME",8
```

Once this command is executed, the advisory SAVING PROGRAM NAME will appear on screen while the program is being copied onto the disc. Once the save operation is completed, the advisory READY appears on screen.

At times it may be desirable to save a program on disc in place of a previous version of the same program. The example below adds the characters @0: as the first three characters of the program name.

```
SAVE"@0:PROGRAM NAME",8
```

This causes the disc operating system (DOS) to erase the previous version of the program with that name from the disc. The version currently held in the computer is then saved in its place.

14 - 3. VERIFYING PROGRAMS

The advantage in saving programs on disc or tape is that you can save your programs, shut off the computer, and come back days or weeks later and load those programs back into the computer. Most of the time this works quite well. Magnetic recording is not absolutely reliable however. Small inperfections in the disc or tape material can cause some of the saved program data to be lost.

The VERIFY command allows you to check the recorded program to make sure it was correctly stored before you shut off the computer. This command causes a byte-by-byte comparison between the program in the computer and the program that is on tape or disc. The example below will verify a program saved on disc:

```
VERIFY "PROGRAM NAME",8
```

When this command is executed, the advisory SEARCHING FOR PROGRAM NAME will appear on screen. Once the program is found the check is performed. If the program on disc matches the program in the computer exactly, the advisory OK READY will appear on screen. If the programs do not match, the advisory message ?VERIFY ERROR READY will appear. In the event of an error, simply resave the program.

To verify programs saved on tape, simply drop the device number from the disc example command. The instruction PRESS PLAY ON TAPE will appear on screen when the command is executed. The resulting advisories will be the same as those used when verifying disc programs.

14 - 4. LOADING PROGRAMS FROM TAPE

The LOAD command allows you to load programs from tape or disc into the computer. When loading from tape it is important to rewind the tape to a point before the beginning of a program. The command shown below will load a program by name:

```
LOAD "PROGRAM NAME"
```

When this command is entered, the instruction PRESS PLAY ON

TAPE will appear on screen. Press the PLAY key on the Datasette recorder. The computer will scan the tape to find a program by the name specified in the LOAD command. If an end-of-tape marker is found before the program is, the error message ?FILE NOT FOUND will appear on screen.

Once the program is located, the advisory FOUND PROGRAM NAME will appear. At this point, the program has been located but it has not been loaded. To load the program, press the Commodore logo key (C=) on the keyboard. When the load sequence is completed, the READY advisory will be displayed.

You may also load programs without specifying a name using the command below. This command will load the first program found on the tape.

```
LOAD ""
```

The same operation as LOAD"" may be accomplished by pressing the SHIFT and RUN/STOP keys at the same time.

14 - 5. LOADING PROGRAMS FROM DISC

Loading programs from disc is very much like loading from tape. Again, the device number (8) must be added to the LOAD statement as shown below:

```
LOAD "PROGRAM NAME",8
```

When this statement is executed the advisory SEARCHING FOR PROGRAM NAME will appear on screen. If no program is found under that name, the ?FILE NOT FOUND error occurs.

If the program is located, the LOADING advisory prints on screen followed by READY when the loading is completed.

The disc operating system does provide some capabilities that were not available when using tape. Most involve the use of special characters in the program name contained in the LOAD statement.

The disc contains an index of the name of all programs and files stored on the disc. You can load this disc directory just as you would a Basic program by using the LOAD"$". The dollar sign ($) tells the disc operating system to transfer the directory to the computer. Entering the LIST command causes the directory to print out on screen showing the name of each program stored on the disc.

The asterisk (*) is used to load programs by a partial name. For example:

```
LOAD "TA*",8
```

This command will load the first program in the directory that starts with a TA such as TAnk or TAste. When used alone, the asterisk causes the last program accessed to be loaded. If no programs have yet been loaded, the asterisk causes the first program in the directory to load.

The question mark (?) serves as a wild card character in program names. When the question mark is used in the load command, any program that matches the name specified in the load command except for the characters covered by the question mark will be loaded. For example:

```
LOAD "TA??",8
```

This command will load the programs TASK, TANK OR TAPS, whichever is listed first in the program directory.

LESSON 15

More Screen Printing

Lesson 4 of this tutorial described some simple techniques to print data on the screen. This lesson expands on that capability by showing the techniques to add the use of color, control the position of text on screen, and use the Basic PRINT statement to design better screen displays.

The PRINT statement is probably the most powerful Basic command available. It almost comprises a language of its own. It is possible to embed commands within the quote marks found in a PRINT statement. These commands will be executed when the PRINT statement tries to print them. Many of the operations described in this lesson are based on this concept.

15 - 1. CLEARING THE SCREEN

At times it is desirable to completely clear the screen before printing text on it. This can be done by embedding a screen clear command within quotes. This command is entered by pressing the SHIFT key and the CLR/HOME Key at the same time while typing within the quote marks of a PRINT statement. This will appear on screen as an inverse video capital S character. Consider the example below:

```
100 PRINT "[clear]"
110 PRINT "SCREEN CLEARED"
120 END
```

Note that although we have simply written clear within brackets in the print statement in line 100, this will appear on screen as a reverse video graphics character inside quotation marks.

15 - 2. CURSOR CONTROL

Recall from an earlier lesson that there are two cursor control keys located at the lower right-hand side of the keyboard. These keys

may be embedded in quotes just as the CLR/HOME key was in the previous example. These embedded cursor commands can be used to move the print cursor left, right, up or down so that the next thing printed by your program can appear anywhere on screen that you like.

In the example below, the cursor is moved down two spaces after each word is printed. Keep in mind that the cursor will move down one space automatically after any PRINT statement that does not end in a semicolon. Because of this, by embedding one down cursor control command, you will cause the cursor to skip a line. Likewise, by embedding one up cursor control command you would cause the cursor to overprint the same line. Recall that line 100 clears the screen.

```
100 PRINT "[clear]"
110 PRINT "FIRST PRINT[cursor down 1]"
120 PRINT "SECOND PRINT"
130 END
```

Again note that since the reverse video graphic character cannot be reproduced in this manual, we have simply spelled out cursor down 1 in brackets in line 110. When entering this line simply press the CRSR ↑ key once after entering the FIRST PRINT text.

15 - 3. PRINTING IN COLOR

Recall that the numbered keys 1 through 8 cause the color of printed text to change when used with the CTRL key or the Commodore logo key (C=). In this way up to 16 colors are possible.

These color command may also be embedded in quotes. Simply press the CTRL or C= key and the numbered key that corresponds to the color of choice while typing within the quote marks of a PRINT statement. A reverse video symbol will appear in the statement when listed but as in the previous examples in this lesson the symbol will be executed rather than printed. Consider the following example:

```
100 PRINT "[ctrl/3]RED"
110 PRINT "[ctrl/7]BLUE"
120 PRINT "[ctrl/6]GREEN"
```

Once a color is set, printing will continue in that color until the next color control command is encountered.

15 - 4 PRINTING IN REVERSE VIDEO

The numbered keys 0 and 9 control the reverse video function when used with the CTRL key. These commands can also be embedded in quotes as shown in the example below:

```
100 PRINT "NORMAL VIDEO"
110 PRINT "[ctrl/9] REVERSE VIDEO"
120 PRINT "[ctrl/0]NORMAL AGAIN"
```

Note that CTRL9 activates reverse video and CTRL0 turns it off.

15 - 5. SETTING BACKGROUND AND BORDER COLOR

Notice that on the video screen the main portion of the screen is white. This area is termed the BACKGROUND. The edge of the screen is called the BORDER and throughout the Basic Tutorial program it appears in a number of different colors. The color of each of these two areas can be changed quite easily.

The color of each area is determined by a number stored in a memory address in the computer. Address 53280 controls the border color while address 53281 controls background color. By changing the number stored in either address you can change the color.

There are 16 colors available and each is represented by a number as shown in the following table.

COLOR	NUMBER
BLACK	0
WHITE	1
RED	2
CYAN	3
PURPLE	4
GREEN	5
BLUE	6
YELLOW	7
ORANGE	8
BROWN	9
LIGHT RED	10
GRAY 1	11
GRAY 2	12
LIGHT GREEN	13
LIGHT BLUE	14
GRAY 3	15

The POKE statement allows you to alter the contents of any memory location. For example:

```
POKE 53280,2
```

This statement pokes the number 2 into memory location 53280. Notice that the desired value (2) follows the address (53280) and is separated from it by a comma. This example would change the border color to red since 2 is the number for red and 53280 is the address that controls border color.

In the example below, all the numbers representing colors are poked into the border color address (53280) one at a time. Lines 120 and 130 create a delay loop to allow enough time between color changes so that each color can be seen.

```
100 FOR N=0 TO 15
110 POKE 53280,N
120 FOR P=1 TO 500
130 NEXT P
140 NEXT N
150 END
```

By changing the address in line 110 to 52381 you can create the same effect on background color.

```
100 FOR N=0 TO 15
110 POKE 52381,N
120 FOR P=1 TO 500
130 NEXT P
140 NEXT N
150 END
```

15 - 6. THE SPC STATEMENT

The SPC statement allows you to print up to 255 spaces on screen. This can be useful when formatting text for screen displays. The number of spaces to be printed is specified by the number held in parenthesis following the SPC keyword. Consider the example below:

```
600 PRINT SPC(6)"NAME";SPC(16)"PHONE"
610 END
```

It is important to note that the cursor will not merely skip 16 spaces before printing PHONE as a TAB statement would. Blanks will actually be printed in each of the 16 spaces. Any text that might be in those spaces will be overwritten and effectively erased.

15 - 7. THE POS STATEMENT

The POS statement returns a number between 0 and 79 indicating where the cursor is located on the screen line. If the value returned is greater than 40 it indicates the next screen line. POS, of course, stands for position. The argument of the POS statement is a dummy. This means that any number may be used in parenthesis with no effect at all. The value returned will be the same no matter what value is used in the argument. Consider the example below:

```
100 FOR N=1 TO 50
110 PRINT "A";
120 IF POS(0)=30 THEN 140
130 NEXT N
140 PRINT "[ctrl/1]END"
```

In this example lines 100-130 set up a loop to print the letter A 50 times. Line 120, however, breaks the loop when the print cursor reaches position 30. Note also that a color control is used in line 140 to make END print in black.

LESSON 16

Data Handling Techniques

One of the most useful capabilities of the personal computer is its ability to deal with large amounts of data. This data can be stored, sorted and searched in a number of ways to put this information in a more useful format. A common example of this would be a mailing list. Names, addresses and phone numbers are stored in the computer as data. This data can then be searched for a particular name. Once the name is found, the associated address and phone number can be printed on the screen. This lesson describes some of the more basic techniques involved in storing and using data.

16-1. VARIABLE ARRAYS

An array is a list of data items referred to by a single variable name. A list of phone numbers could be considered an array with each number an element of the array.

Individual elements of the array can be accessed by use of a subscript. A subscript is a number in parenthesis following a variable name. This number locates the element within the variable array. For example:

```
A(1)=4289239
A(2)=3357765
A(3)=4280754
A(4)=4276880
```

In this case, there is an array of four numbers stored in variable A. As long as the number of elements of an array does not exceed 10, these arrays can be specified by simple assignment statements such as those shown following:

```
100 A(1)=4289239
110 A(2)=3357765
120 A(3)=4280754
130 FOR N=1 TO 3
140 PRINT A(N)
150 NEXT N
```

Lines 100-120 assign numbers to each of three elements of the variable A. Lines 130-150 simply print out these numbers.

If there are more than 10 elements in an array, a DIM (dimension) statement must be used to reserve space within memory for the data. The largest number of elements possible is 32767. The DIM keyword is followed by the variable name and the number of elements to be reserved. The example below dimensions the variable A for 20 elements.

```
100 DIM A(20)
110 FOR N=1 TO 20
120 A(N)=N*15
130 NEXT N
135 PRINT "[clear screen]"
140 FOR P=1 TO 20
150 PRINT A(P);TAB(2);
160 NEXT P
```

In this example lines 110-130 fill the variable array with multiples of 15. Line 135 clears the screen. Lines 140-160 print out the contents of the array on a continuous line with 2 spaces between each number.

An important characteristic of DIM statements is that they can only be executed one time in a program. If the program attempts to execute a DIM statement a second time, the REDIM 'D ARRAY error message appears. DIM statements can be reexecuted, if necessary, by first executing a CLR statement. Any data held in the array would be lost however.

16 - 2. MULTIDIMENSIONAL ARRAYS

The Commodore Basic language supports arrays of multiple dimensions. Two columns of 20 numbers is an example of a two

dimension array. The DIM statement for such an array would look like this:

```
DIM B(2,20)
```

The first number in parenthesis is the number of columns and the second number indicates the number of elements in each column. To access any element of the array, both the column and the element in the column must be specified. The statement PRINT B(2,17) would print out the seventeenth element in column two.

In the example below the variable B is set for two dimensions (3 columns of 20 elements). Lines 210-240 make up a loop to fill the array with multiples of 15 in column one, 25 in column two, and 42 in column three. Line 250 clears the screen and lines 260-280 print out the three columns.

```
200 DIM B(3,20)
210 FOR N=1 TO 20
220 B(1,N)=N*15
230 B(2,N)=N*25
235 B(3,N)=N*42
240 NEXT N
250 PRINT"[clear screen]"
260 FOR L=1 TO 20
270 PRINT B(1,L),B(2,L),B(3,L)
280 NEXT L
```

Arrays can be dimensioned for up to 255 dimensions. An example of a three dimensional array might look like this:

```
DIM B(5,3,20)
```

This example could be thought of as five pages of three columns of 20 elements.

Arrays can also be used for string and integer variables by simply adding the data type character $ or % to the DIM statement (DIM A$(20)).

Although it is theoretically possible to DIM an array with 255 dimensions containing 32767 elements each, this array would contain a staggering total of 8,355,585 elements. Floating-point variables require 5 bytes of memory to store each element of an array. String variables require 3 bytes per character and integer variables require 2 bytes to store each element. An array contain-

ing 8,355,585 floating-point numbers would require 41,777,925 bytes of memory. Unfortunately, the Commodore 64 computer only provides 38,911 bytes of memory. If a program attempts to execute a DIM statement that exceeds the 38,911 byte limit, an OUT OF MEMORY error message will appear on screen.

16 - 3. READ AND DATA STATEMENTS

Arrays provide a powerful tool for handling data because they are very easy to search through and because it is easy to assign a value to any one element without disturbing any of the others. Arrays do have a disadvantage however. Any time a Basic program line is entered into the computer, all data held in any arrays is lost. For this reason, it is often useful to store data in DATA statements and load data from the DATA statements into the array using a READ statement.

DATA statements are simply lists of data elements that follow a DATA keyword. The data elements are separated from each other by commas. Numbers, characters, even strings can be stored in DATA statements. A typical DATA statement is shown below:

```
DATA JOHN, MARY, BILL, JIM, KATHY
```

All DATA statements in a program are treated as one continuous list starting with the first item in the lowest program line number and continuing through the last item in the DATA statement with the highest program line number.

DATA statements are not actually executed like other types of statements. They are referred to for information by other portions of the program. For this reason they are often found at the very end of a program although their location does not matter. If the data items are strings that contain punctuation such as commas, periods or semicolons, each data item should be enclosed in quote marks.

The READ statement is used to fill variables from the list of items held in the DATA statements. Each time a READ statement is executed another item is pulled off the list held in the DATA statements. If a READ statement is executed after the last item has already been read from the list, the OUT OF DATA error will occur. The type of variable that the READ statement is filling must match the type of data available from the list. If not, the error message ?SYNTAX ERROR will occur. Consider the example below:

```
100 DIM A$(20)
105 PRINT"[clear screen]"
110 FOR N=1 TO 20
120 READ A$(N)
130 PRINT A$(N)
140 NEXT N
150 DATA JOHN, JAMES, ROBERT, WILLIAM, BETTY,
SUSAN, DOUGLAS, NANCY
160 DATA GEORGE, KATHERINE, JUANITA, HAROLD,
ARTHUR, DAVID, GREGORY
170 DATA EVELYN, HERBERT, MICHAEL, ELIZABETH, FR
ANCINE
```

In this example, each of the twenty names held in the DATA statements was read into the A$ variable array. The A$ array is then printed on the screen.

A data pointer is used to locate which item from the data list is to be read next. When this pointer reaches the last item on the list it will prevent any further READ statements from executing. In fact, if another READ statement is attempted, the OUT OF DATA error message will appear on screen. This data pointer can be reset to the first item on the list by the RESTORE command. The data can then be reread by another segment of the program. The RESTORE command has no argument. It is a simple one word command.

16 - 4. SEARCHING AND SORTING ARRAYS

One of the disadvantages of storing data in DATA statements is that there is no way to change any of the data from within a program. Any changes have to be made manually. Data stored in arrays can be changed quite easily however. Presented in this topic is a program to sort names held in an array so that the names are stored in alphabetical order. This particular sort routine is commonly known as a BUBBLE sort.

Recall from an earlier lesson that we can compare two strings using relational operators such as $<$, $=$, and $>$. One string is considered to be less than another string if it would normally precede the string in alphabetical order. We will use this concept in our bubble sort program which is shown below. Assume that the names loaded into the A$ array in the last topic are still there.

```
100 FOR PASS=1 TO 19
110 FLAG=0
120 PRINT"[home]"
130 FOR N=1 TO 19
140 IF A$(N) > A$(N+1) THEN GOSUB 300
150 PRINT A$(N);"      "
160 GOSUB 400
170 NEXT N
180 PRINT A$(20)
190 IF FLAG=0 THEN END
200 NEXT PASS
210 END

300 BUFFERS$=A$(N)
310 A$(N)=A$(N+1)
320 A$(N+1)=BUFFER$
330 FLAG=1
340 RETURN

400 FOR T=1 TO 50
410 NEXT T
420 RETURN
```

Lines 100-200 make up a PASS loop. This loop establishes the number of times the program will cycle through the array to alphabetize the list of names. The greatest number of passes required in a bubble sort is always one less than the number of items in the list.

Line 110 zeroes the FLAG variable. The flag is used to detect when a pass is made where no names had to be swapped. This would indicate that the sort was completed and all names were in alphabetical order.

Line 120 contains a HOME command within quotes so that for each pass the print cursor is reset to the top of the screen.

Lines 130-170 make up a compare loop which is nested within the pass loop. Line 140 of this loop compares the name located at element N of the array with the name that immediately follows it in the array. If the two names are already in alphabetical order no action is taken. If the names are not in order the exchange subroutine contained in lines 300-340 is called to swap the two names and set the FLAG variable to 1.

Line 150 prints element N of the array along with some trailing spaces to erase any characters left behind by longer names pre-

viously printed on that line. The compare loop is repeated for each of the items in the loop except the last one which has no item following it.

Line 180 prints the last item in the array since the compare loop only prints up to item 19.

Line 160 calls a simple time delay loop located in lines 400-420. This loop along with all of the print statements in this program are provided so that you can see this sort routine operate step by step. They are unnecessary in a normal sort routine.

Line 190 ends the program if the flag is still 0. This would indicate that no exchanges were made during this pass so all items in the list must be in order. If an exchange was made during each pass the elements will all be in order after 19 passes anyway and line 210 will end the program.

The exchange subroutine is contained in lines 300-340. Line 300 copies element N of the array into a buffer string. The element following N is copied into N by line 310. Line 320 stores the contents of the buffer string in element N + 1. This effectively swaps the two array elements. Line 330 sets the FLAG variable to 1 indicating that a swap has been performed.

As this example makes apparent, the various names bubbled up the list to their appropriate alphabetical positions. In actual use, this routine would not contain print statements and time delay loops. The computer is quite capable of sorting a list of several hundred names in just a few seconds.

ASSIGNMENT

Write a telephone directory program in five parts as follows:

Part One

Dimensions a two dimensional string array to hold 50 names, 50 addresses and 50 phone numbers. Fills entire array with zeroes. This program section is contained in line numbers 100-199.

Part two

Clears the screen and prints a menu presenting four options:

1. ENTER NAMES
2. SEARCH NAMES
3. ALPHABETICAL SORT
4. QUIT

Keyboard is monitored and causes a jump to program line 1000 if ENTER NAMES is selected, 2000 if SEARCH NAMES is selected, 7000 if ALPHABETICAL SORT is selected and ends program if QUIT is selected. This section of program to occupy lines 200-499.

Part Three

ENTER NAMES- searches array to find first array element that does not contain a name. Allows continuous entry of names, addresses and phone numbers into array beginning at first empty element. If MENU is entered as a name it is not stored in array but rather causes a jump to the main menu section causing it to display. This program section may occupy any program lines from 1000-1999.

Part Four

SEARCH NAMES- allows user to enter a name. Array is then searched for that name. If found, the corresponding address and phone number, along with the entered name is displayed on screen. If the array search is not successful an advisory message to that effect is displayed on screen. Use may then enter an alternate name. If MENU is entered as a name it will cause a jump to the main menu. This section of program occupies lines 2000-2999.

Part Five

ALPHABETICAL SORT- This section of the program clears the screen and asks the user to standby while an alphabetical sort of the array is performed. A count is made to determine how many names are contained in the array. Those names, along with their corresponding addresses and phone numbers are then sorted into alphabetical name order. The user is then advised that the sort is complete. After a short delay, the main menu redisplays. This program section occupies lines 7000-7999.

Once your program is working to your satisfaction, save it on disc for later use. A sample program solution may be found in Appendix A of this manual.

LESSON 17

Disc Files

Lesson 14 described the various methods of saving programs on disc. While this works fine, it is often desirable to save data such as that held in data statements or variable arrays on disc. The program can be contained in the computer and data can be moved from disc to the program and from the program to disc as necessary. By switching data in and out of the computer this way, extremely large blocks of data can be stored and used by a relatively small program.

These data files may be one of three types: sequential, random or relative. This lesson describes the use of sequential files.

Sequential files are somewhat like data statements. Data must be sent to sequential files one item at a time and data read from sequential filés is read one item at a time from beginning to end. The first item that was sent to the file will be the first one read when the information is retrieved. Sequential files can hold numbers or string data (up to 80 characters per item). Any number of items can be sent to a file.

17 - 1. SENDING DATA TO SEQUENTIAL FILES

Before sending data to the disc to be stored we must give the file a name and open a channel of communication to be the disc drive. The OPEN statement does just that.

```
OPEN 1,8,8,"FILE NAME,S,W,"
```

The first number (1) following the OPEN keyword is the file number. This can be any number between 1 and 127 and it is used to identify the file that is being used.
The second number (8) is the device number and it will always be 8 for disc drives.
The third number (8) is the channel number. For sequential files this number can be anything from 2 to 14. This is the channel used to communicate with the disc drive.

Notice that these numbers are separated by commas. Immediately following is a string held in quotes. This string contains a file name, file type and direction. The file name can be up to 16 characters and is used in the disc directory to identify the file. The file type in this case is S for sequential. The direction is W (write) when sending data to the disc and R (read) when reading from the disc.

Once the file is opened, we can send data to the disc using the PRINT# statement. This statement must call the same file number as the OPEN statement did.

```
PRINT#1,A$(7)
```

This statement will send item 7 from the A$ array to disc file number one.

The example below assumes that the names stored in A$ array in lesson 16 are still there. This example will send each of the twenty names to a sequential file named FILE TEST and numbered 1.

```
500 OPEN 1,8,8,"FILE TEST,S,W"
510 FOR N=1 TO 20
520 PRINT#1,A$(N)
530 NEXT N
540 CLOSE 1
550 END
```

When executed this example will store all 20 names in disc file #1. This file will be listed in the disc directory as FILE TEST. In this example line 500 opens the file to write. Lines 510-530 make up a loop that sends each of the 20 names in the A$ array to the disc. Line 540 closes the file. This is very important. If files are left open data can be lost and other files may even be destroyed.

17 - 2. READING DATA FROM SEQUENTIAL FILES

The INPUT# statement is used to read an item from the disc file. Again the same file number used in the open statement must be used in the INPUT# statement.

```
INPUT#1,A$(7)
```

This example would pull the next available item from the disc file and store it in element 7 of the A$ array. The INPUT# statement can handle strings or numeric data up to 80 characters in length.

Files must be opened and closed when reading files just as they were when writing files. However, the OPEN statement will have the letter R (read) for the direction character.

In the example below line 600 clears all variables including the A$ array. Line 610 dimensions the A$ array and line 620 clears the screen. Line 630 opens the file. Lines 640-670 make up a loop to read each name from the disc file into the A$ array and print them on the screen.

```
600 CLR
610 DIM A$(20)
620 PRINT"[clear screen]"
630 OPEN 1,8,8,"FILE TEST,S,R"
640 FOR N=1 TO 20
650 INPUT#1,A$(N)
660 PRINT A$(N)
670 NEXT N
680 CLOSE1
690 END
```

Line 680 closes file number 1. Again, it is extremely important to properly close all files accessed so that no disc files are damaged.

17 - 3. UPDATING SEQUENTIAL FILES

Many times it is necessary to send data to a file already created. If the OPEN statement given in topic 17-1 is used again, a new file will be created with the same name. This will cause two files on the disc, both with the same name. This is very undesirable.

We can avoid this by adding the characters @0: to the file name in the OPEN statement as shown below:

```
OPEN 1,8,8,"@0:FILE TEST,S,W"
```

Data can now be sent to the existing file number 1 FILE TEST.

ASSIGNMENT

Load the telephone directory program written for the lesson 16 assignment. Add program lines to this program to add sequential disc filing capability to this program in five parts as follows:

Part One

MENU- clears the screen and print a menu that presents four options:

1. FILE ARRAY.
2. LOAD ARRAY.
3. UPDATE EXISTING FILE.
4. RETURN TO MAIN MENU.

This section of program will monitor the keyboard and jump to the appropriate section of program in response to menu selections as follows:

FILE ARRAY	4000
LOAD ARRAY	4500
UPDATE EXISTING FILE	5000
RETURN TO MAIN MENU	200

This program section occupies lines 3000-3500.

Part Two

FILE ARRAY- This program section clears the screen and allows the user to enter a file name and file number. The contents of the array are then saved in a new sequential disc file under the entered file name and number. An advisory message indicating that the file operation is completed appears on screen after the file operation has been performed. After a short delay, the file menu redisplays. This program section occupies lines 4000-4499.

Part Three

LOAD ARRAY- This program section clears the screen and allows the user to enter the name and number of an existing sequential disc file. The specified file is loaded from disc into the array. Once loading is completed, an advisory to that effect appears on screen. After a slight pause, the file menu redisplays. This program section occupies lines 4500-4999.

Part Four

UPDATE EXISTING FILE- This program section clears the screen and allows the user to enter the name and number of an existing sequential disc file. The contents of the array are then saved to that disc file replacing whatever was stored there. Once filing is completed, an advisory to that effect appears on screen. After a slight delay the file menu redisplays. This program section occupies lines 5000-5499.

Part Five

Revise the main menu created in lesson 16 to list a FILE AREA option on the menu. If selected, this option should cause a jump to line 3000 of the program.

Once your updated program is working to your satisfaction, save it on disc for later use. A sample solution is provided in Appendix A of this manual.

LESSON 18

Disc Utilities

The disc drive unit contains a small computer of its own that performs all the necessary housekeeping functions required to store programs and data files on the disc. By sending commands to this computer we can cause it to perform a variety of different utility functions. Programs or files can be erased, copied or renamed quite easily.

Although disc commands are not actually considered a part of the Basic language, a familiarity with these commands is an essential part of programming the Commodore 64 computer. This lesson will present those commands necessary to use the disc drive effectively.

18 - 1. THE COMMAND CHANNEL

Recall from an earlier lesson that there are 16 channels available to communicate with the disc drive. Channels 0 and 1 are reserved for use by the system when loading or saving programs. These two channels should never be used in an OPEN statement. Channels 2-14 may be used for sending data to and from the disc drive. Channel 15 is the COMMAND channel. All disc commands should be sent to the disc drive using channel 15.

In order to send a command to the disc drive, channel 15 must first be opened. The easiest way to do this is with the statement:

```
OPEN 15,8,15
```

Recall that the first number following the OPEN keyword is the FILE number. For most disc commands which file number is used is not important. The second number is the DEVICE number. This is usually 8 for disc drives. The last number is the channel number. This number is always 15 when sending disc commands.

Disc commands are sent to the disc drive using the Basic PRINT # statement as shown :

```
100 OPEN 15,8,15
110 PRINT#15,"COMMAND"
120 CLOSE 15
```

Line 100 opens the command channel. Line 110 sends whatever command is selected to the disc drive. The command is held within quote marks following the PRINT#15,. Line 120 closes the file.

18 - 2. FORMATTING NEW DISCS

Before a disc can be used for the first time it must be FORMATTED. The disc drive erases the entire disc, puts timing and block markers on it, and creates the disc directory.

The NEW command is used to format discs. This command is not the same as the Basic language statement NEW which erases all programs in the computer. The NEW disc command appears below:

```
110 PRINT#15,"NEW0:NAME, ID CODE"
```

Note that the characters zero (0) and colon (:) follow the word NEW. The zero is the drive number (usually zero). Any text put in as the NAME will appear in the disc directory as the title of the disc. The ID CODE can be any two characters. This code further identifies the disc.

Once the command channel is opened and the NEW command sent, the disc drive will format the new disc. This usually takes about 90 seconds. After the drive stops turning, make sure the file is properly closed by a CLOSE 15 statement.

18 - 3. ERASING FILES

The SCRATCH command is used to erase unwanted files from the disc. The SCRATCH command and the name of the file to be erased are sent to the disc drive by a PRINT#15 statement. Consider the example statement below:

```
100 PRINT#15,"SCRATCH0:NAME"
```

Again note that the SCRATCH command is followed by the drive number (zero) and a colon. Once this statement is sent to the disc drive, the file named will be erased and the area it occupied will be made available for other programs or files.

18 - 4. RENAMING FILES

Renaming files is almost as easy as erasing them. The RENAME

command is used just as the SCRATCH command was except that two names will follow it instead of just one name.

```
110 PRINT#15,"RENAME0:NEW NAME=OLD NAME"
```

Notice that the NEW NAME and the OLD NAME are separated by an equals sign (=). The file itself will not be changed but the file name will be changed to the text that immediately follows the colon in the PRINT#15 statement.

18 - 5. COPYING FILES

At times it might be desirable to copy a file so that two identical files exist on the same disc. In this way if one area of the disc is damaged there is still a chance that the backup version of the program or file can be recovered.

The COPY command allows you to make a copy of any program or file on the disc. It will not copy programs from one disc to another. The COPY command is shown below:

```
110 PRINT#15,"COPY0:NEW NAME=0:OLD NAME"
```

Notice that like the RENAME command the two names are separated by an equal sign. However, in this case a disc drive number (0) and colon also appear with the equal sign. This statement will cause the file specified by OLD NAME to be duplicated and stored under NEW NAME.

18 - 6. COMBINING FILES

It is possible to splice files together using the COPY command introduced in the last topic. The format for this command is shown below:

```
110 PRINT#15,"COPY0:NEW FILE=0:FILE A,0:FILE B
```

This command will create a new file made by adding file B to the end of file A. Up to four separate files may be combined in this way. Note that each file in the series must be separated by the characters ,0: to keep the various file names straight.

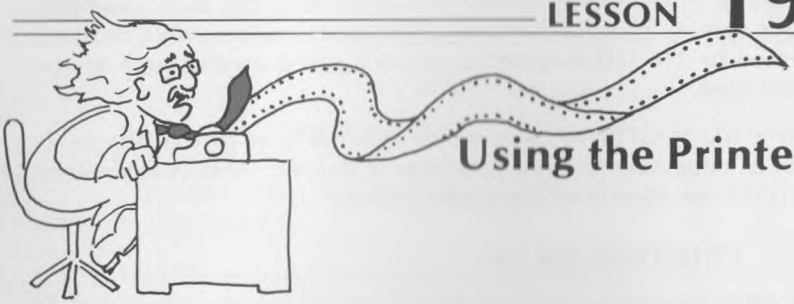
18 - 7. VALIDATING DISCS

After a disc has been used to save, scratch, rename and copy programs and files a number of times, there are likely to be a number of gaps of unused disc area scattered in between files as a

result of the various operations performed. These gaps cannot be used because they are usually too small to hold the amount of data in most files.

The VALIDATE command will cause the disc drive to go through the entire disc and clean house. All the files will be lined up in order and any files that were never properly closed by a CLOSE statement will be deleted. As a result, more storage area will be made available for files. The format for the VALIDATE command is shown below:

```
100 PRINT#15,"VALIDATE"
```



LESSON 19

Using the Printer

While the video display normally does a good job of displaying computer output, it does have a few glaring shortcomings. Most of these disadvantages have to do with the size limitations of the screen. The video display of the Commodore 64 computer is limited to 25 lines of 40 characters. In some applications this just won't do.

For example, when reviewing a Basic program of several hundred lines for errors, the 25 line display forces you to constantly relist various portions of the program. By using a printer as an output device you could print out the entire program and have it right in front of you.

The VIC-1525 graphics printer can print any character that can be produced by the Commodore 64 computer and displayed on screen. Additionally, it provides a line length of 80 characters and no vertical line limitations at all. This lesson describes some of the Basic programming techniques required to print text on the VIC-1525 graphics printer.

19-1. OPENING PRINTER FILES

The printer is recognized as a peripheral device just as the disc drive is. Recall that the disc drive was usually device number 8. The printer is usually either device number 4 or number 5 depending on the position of the self-diagnostic switch on the back of the printer.

Recall also that in order to send data to a disc drive a file had to first be established using an OPEN statement. Use of the printer is pretty much the same in this respect. The format for opening printer files is shown below:

```
OPEN 8,4
```

The first number after the OPEN keyword is the file number (in this case 8). The file number can be anything between one and 255.

This number will be used in all subsequent commands to the printer. Separated from the file number by a comma is the device number (in this case 4). This number must correspond to the switch position selected by the self-diagnostic switch located on the back of the printer case.

After a file is OPENed, commands may be sent to the printer as often as necessary to print whatever is desired. Always remember to CLOSE the open print file when finished.

19 - 2. PRINTING DATA

The PRINT# statement is used to send data to the printer to be printed much as the PRINT statement is used to print data on the screen. The file number must be specified in the PRINT# statement however. An example is shown below:

```
100 OPEN 8,4
110 PRINT#8,"ORBYTE SOFTWARE"
120 CLOSE 8
```

This example will cause the printer to print ORBYTE SOFTWARE one time starting at the left side of the paper.

String variables can be printed on the printer just as easily as literal text enclosed in quote marks. Consider the following example:

```
300 A$="WINSTON CHURCHILL"
310 OPEN 9,4
320 PRINT#9,A$
330 CLOSE 9
```

This example will print WINSTON CHURCHILL on the paper.

19 - 3. UPPER CASE/GRAPHICS MODE

Recall that the Commodore 64 keyboard provides two character sets that can be printed on screen - UPPER CASE/GRAPHICS and UPPER/LOWER CASE. These two modes are also available on the VIC-1525 printer along with two other modes - DOUBLE WIDTH, and REVERSE FIELD. Printer modes are selected by sending printer control codes to the printer via PRINT# statements. Initially the printer is in the UPPER CASE/GRAPHICS mode so there is no need to send a control code to enter this mode when the equipment is first turned on. In this mode all characters will be printed using the UPPER CASE/GRAPHICS character set and will appear on paper just as they would on screen.

Printer control codes are always numbers contained in a CHR$ statement. The numeric code for the UPPER CASE/GRAPHICS mode is 145. By sending a CHR$(145) to the printer this mode is selected. Consider the example below:

```
200 A$="JOHN HANCOCK"
210 OPEN 7,4
220 PRINT#7,CHR$(145)A$
230 CLOSE 7
```

Notice that there is no punctuation between the CHR$(145) and A$ in line 220 of this example. This bit of program will print the name JOHN HANCOCK in upper case letters.

19 - 4. UPPER/LOWER CASE MODE

Selecting the UPPER/LOWER CASE mode is just a bit different from selecting any of the other modes in that a secondary address of 7 must be added to the OPEN statement to ensure proper operation. The control code for UPPER/LOWER CASE mode is CHR$(17). The example below illustrates this.

```
200 A$="JOHN HANCOCK"
210 OPEN 3,4,7
220 PRINT#3,CHR$(17)A$
230 CLOSE 3
```

This example will print the name john hancock in lower case letters.

19 - 5. DOUBLE WIDTH MODE

The VIC-1525 printer has the ability to print double wide characters. This can be very effective for titles, paragraph headings, or anywhere a larger type can be used to attract attention. The DOUBLE WIDTH mode is selected by sending a CHR$(14) control code to the printer.

```
200 A$="JOHN HANCOCK"
210 OPEN 3,4
215 PRINT#3,CHR$(14)"NAMES"
220 PRINT#3,CHR$(15)CHR$(17)A$
230 CLOSE 3
```

In this example line 215 causes **NAMES** to print out double width. The name held in A$ will then print out in lower case letters.

DOUBLE WIDTH mode can be used in combination with any of the

other modes. If in UPPER/LOWER CASE mode for example, upper and lower case letters can still be printed but they will appear double width. DOUBLE WIDTH mode is turned off by sending the control code CHR$(15) to the printer as shown in line 220 of the example.

19 - 6. REVERSE FIELD MODE

Recall that when printing on screen characters may be printed in reverse video. The VIC-1525 printer can do the same thing using the REVERSE FIELD mode. Like DOUBLE WIDTH mode, REVERSE FIELD also has two control codes. CHR$(18) turns it on and CHR$ (146) turns it off.

```
100 OPEN 6,4
110 PRINT#6,CHR$(18)"REVERSE FIELD"
120 PRINT#6,CHR$(146)"NORMAL"
130 CLOSE 6
```

Line 110 prints REVERSE FIELD white on black while line 120 prints NORMAL as normal characters.

19 - 7. CONTROLLING PRINT POSITION

There are three more functions of the printer that can be controlled by sending control codes to the printer. All have to do with controlling where on the paper the printer will print the data.

LINE FEED Sending control code CHR$(10) to the printer causes a line feed to be performed. The paper is moved up one line so that the print head prints on the next line. The print head does NOT return to the left margin of the paper however.

CARRIAGE RETURN Sending the control code CHR$(13) causes a carriage return to be performed. In this case, the paper is also moved up one line but the print head also returns to the left margin of the paper. This function is actually exactly like a carriage return on a normal typewriter.

TAB POSITION The normal TAB function used when printing on screen does not work on the printer. However, the same thing can be accomplished using the control code CHR$(16) followed by a two digit number in quotes representing the number of spaces to tab.

```
200 A$ = "JOHN HANCOCK"
210 OPEN 7, 4
220 PRINT# 7,CHR$(16)"20"A$
230 CLOSE 7
```

In this example line 220 will print the contents of A$ beginning 20 spaces from the left margin of the page.

19 - 8. LISTING PROGRAMS

One of the main advantages of having a printer is the ability to print out entire programs on paper using the Basic LIST command. However, this LIST command normally causes the program to print out on the video screen. In order to print out a program we must first transfer control from the computer to the printer.

The CMD command performs this transfer of control. Once it is sent to the printer, the line to the printer is left open and the printer is said to be listening. At that point the Basic commands LIST and PRINT will be executed by the printer instead of the computer.

```
100 OPEN 3,4
110 CMD3
120 PRINT"PRINTER IN CONTROL"
130 LIST
```

In this example line 100 opens file number 3 to the printer. Line 110 transfers control to the printer using the same file number. Line 120 is a single PRINT statement. Normally it would cause PRINTER IN CONTROL to be printed on screen. Since control was transferred to the printer, the statement is printed out on the printer instead.

For the same reason, line 130 causes the program to list out on the printer instead of on the screen. However, when any LIST command is executed the computer stops the program when the listing function is completed and the system is left in the direct mode. This causes a problem because the lines to the printer and file 3 are left open. The line can be closed by entering the statement PRINT#3 in direct mode. File 3 can be closed by entering the statement CLOSE 3 in the direct mode.

ASSIGNMENT

Load the telephone directory program developed in the assignments for lessons 16 and 17. Add a program section that counts the number of names in the array file. The program then prints those elements of the array that contain names/addresses/phone numbers out on the VIC-1525 printer in a double column format similar to that below:

Name	Name
Address	Address
Phone number	Phone number

Revise the main menu to add the PRINTOUT menu selection. Revise the keyboard monitor to jump to program line 6000 if printout is selected from main menu. The printout section occupies program lines 6000-6999.

When the program works to your satisfaction, save it on disc for later use. It would be a good idea to printout a complete listing of the program on the VIC-1525 printer at this time.

A sample program solution is provided in Appendix A of this manual.

LESSON 20

Improving Your Programs

When you write your first few programs it will seem that 38911 bytes of memory ought to be enough to satisfy the most industrious programmer. Actually many of the most interesting programs around are less than a hundred lines long. But inevitably you will start to write longer programs of ever increasing complexity. Soon you will find that the 38911 byte limitation that seemed so generous at first becomes a bit more restrictive. Just a few more bytes would allow you to add that one last funtion to make your program really useful.

Luckily those few more bytes are there. Scattered throughout your program are hundreds of little two or three bytes inefficiencies that added together can amount to several hundred bytes of wasted memory space.

This lesson describes some of the ways you can save memory and program more efficiently. Some of these techniques can save a lot of memory and some save only a few bytes. But many times a few bytes are all that you need.

20 - 1. USING MULTISTATEMENT LINES

Throughout this tutorial all program examples have been presented with one Basic statement on each program line. As a rule this makes programs easy to understand and easy to debug (find the errors in).

Unfortunately, this is not very memory efficient. For one thing, line numbers occupy space in memory. If several Basic statements could use the same line number, the memory that would have been used for each line number could be saved. Actually, several Basic statements can be combined on a line by separating each statement with a colon (:). The computer recognizes the colon as the end of a Basic statement.

Program lines cannot be longer than two screen lines (80 characters) no matter how many statements are on the line. Consider the following example:

```
100 REM LOAN PAYMENT CALCULATOR
110 PRINT"[clear screen]"
120 INPUT"ENTER LOAN AMOUNT";LN
130 INPUT"ENTER NUMBER OF YEARS";YRS
140 INPUT"ENTER INTEREST RATE";APR
150 I=APR/1200
160 N=YRS*12
170 B=I/(1-(I+1)↑-N)
180 PY=(INT(LN*B*100))/100
190 PRINT"MONTHLY PAYMENTS=$";PY
200 END
```

This program calculates monthly payments for a mortgage. The memory required to store this program could be reduced by simply combining some of the statements as shown below:

```
100PRINT"[clear screen]":REM LOAN PAYMENT CALCUL
ATOR
110 INPUT"ENTER LOAN AMOUNT";LN
120 INPUT"ENTER NUMBER OF YEARS";YRS
130 INPUT"ENTER INTEREST RATE";APR
140 I=APR/1200:N=YRS*12:B=I/(1-(I+1)↑-N)
150  PY=(INT(LN*B*100))/100:PRINT"MONTHLY  PAYME
  NTS=$";PY:END
```

In this example the number of lines was reduced from 11 to 6 by combining statements using the colon. As far as program operation is concerned nothing has changed.

One caution when using multistatement lines - IF/THEN statements do not execute anything on their line if the IF test is not true. For example:

```
200 IFA=100 THEN B=7:C=A*B:D=C*A
```

If A does not equal 100 the program will immediately go to the next line number. All other operations on line 200 will be ignored.

20 - 2. USING SUBROUTINES TO SAVE MEMORY

After a program has been written and debugged to the point where

it operates to your satisfaction, it is a good idea to review the program for recurring groups of Basic statements.

If the same group of two or three (or more) Basic statements appears several places in the program, you can save a good bit of memory by placing those statements in a single subroutine and calling that subroutine as needed. In this way a simple GOSUB statement can entirely replace the group of statements at each place where they appear in the program. GOSUB statements require very little memory. Consider the program below:

```
100 PRINT"SCREEN ONE"
110 PRINT F7 FORWARD - F1 BACK"
120 GET A$:IF A$="" THEN 120
130 IF A$=CHR$(136) THEN 160
140 IF A$=CHR$(133) THEN 100
150 GOTO 120
160 PRINT"SCREEN TWO"
170 PRINT"F7 FORWARD - F1 BACK"
180 GET A$:IF A$="" THEN 180
190 IF A$=CHR$(136) THEN 220
200 IF A$=CHR$(133) THEN 160
210 GOTO 180
220 PRINT"SCREEN THREE"
230 END
```

This program prints three different screens. Function keys f1 and f7 are monitored and used to control which screen is displayed. Lines 110-150 are virtually repeated in lines 170-210. These two line groups could be replaced by one subroutine except for the fact that they each call out different line numbers. It is possible to work around this using a variable to carry line numbers as shown below:

```
100 PRINT"SCREEN ONE"
110 GOSUB 500: ON X GOTO 100,120
120 PRINT"SCREEN TWO"
130 GOSUB 500: ON X GOTO 100,140
140 PRINT"SCREEN THREE"
150 END

500 PRINT"F7 FORWARD - F1 BACK"
510 GET A$:IF A$="" THEN 510
520 IF A$=CHR$(136) THEN X=2:RETURN
530 IF A$=CHR$(133)THEN X=1:RETURN
540 GOTO 510
```

Lines 110 and 130 call the subroutine at lines 500-540. This subroutine will print F7 FORWARD-F1 BACK on the screen and then monitor the keyboard. The variable X is set to 1 if f1 is pressed and 2 if f7 is pressed. In either case a RETURN is performed. Line 110 or 130 then uses an ONGOTO statement to convert the value held in X to a line number.

In revising this program, the total number of lines was reduced from 14 to 11 for a significant savings in memory used.

20 - 3. OTHER MEMORY SAVERS

The use of REM statements throughout a program while it is in development can be a great aid in making the program more understandable. Subroutines are easier to identify and it is good practice to use REM statements to clarify your programs.

Unfortunately, once the program is finished they are so much excess baggage and can cause you to run out of memory space before the program is completed. Remove them.

Another way to save a good bit of memory is to remove all spaces from each program line. While spaces make the program listing much easier on the eyes, they have no effect on program operation and each of them uses 1 byte of memory.

20 - 4. ABBREVIATING KEYWORDS

It may come as a bit of a surprise but you do not actually have to type each letter of a Basic keyword in order to enter it in a program. Keywords may be abbreviated.

For example, the abbreviation for the Basic keyword PRINT is the question mark(?). If this symbol is entered where you would normally type PRINT the computer will recognize it as the same thing. Most keywords are abbreviated by typing the first letter of the keyword and then holding the SHIFT key while typing the second letter.

When a keyword abbreviation is typed it will appear on screen just as you typed it. However, if the line is later LISTed, the keyword will appear on screen completely spelled out. There is actually no memory savings here, but you will find you can enter programs much faster by using keyword abbreviations.

```
PRINT"GEORGE WASHINGTON"
?"GEORGE WASHINGTON"
```

Entering the text on the first line will have the same effect as entering the text on the second line. However, if you gave each a program line number and entered them, they would both look like the first line when listed.

A complete listing of Basic keywords and their abbreviations may be found in APPENDIX D of the Commodore 64 User's Guide that came with your computer.

ASSIGNMENT

Load the telephone directory program developed in lessons 16, 17 and 19. Add a separate section to this program using the FRE function(lesson 13) to calculate the amount of memory the entire program uses and print the result on screen. Run this program section and record the result.

Revise the telephone directory program using multistatement lines and subroutines to reduce memory requirements. Delete unnecessary spaces from program lines. Make sure that the functional operation of the program is not changed.

Once the revised telephone directory program runs to your satisfaction, save it on disc for later use. Run the memory calculator on this revised version and compare the results to those obtained from the original version. How much memory was saved?

Example solutions are provided in Appendix A of this manual.

APPENDIX A

This appendix provides sample solutions to the programming assignments that occur throughout this tutorial. For most programming tasks there is no one "right" way to accomplish a programming goal. Try to develop a program to accomplish the assigned task on your own first. Then compare your solution to those listed in this appendix.

Lesson 4 Assignment

The assignment at the end of lesson 4 was to write a simple program to print the names and ages of the members of your family in the center of the screen. A program such as that shown below would accomplish this.

```
100 PRINT TAB(15)"JOHN DOE  28"
110 PRINT TAB(15)"NANCY DOE  32"
120 PRINT TAB(15)"JOHN DOE JR.  4"
```

By using the TAB function you can start printing any number of spaces to the right of the left hand margin. In this case about 15 spaces would cause the names to be centered since there are 40 spaces to a screen line.

Lesson 7 Assignment

The lesson 7 assignment asked that you write a Basic program to calculate the loan balance on a $50000 loan at 12% interest. A total of 108 monthly payments of $514.30 have been made on the loan. Based on the formula given in lesson 7, the program below will do this.

```
100 A=50000
110 I=.12/12
120 P=514.30
130 N=108
140 BALANCE=(1/(1+I)↑-N)*(P*(((1+I)↑-N-1)/I)+A)
150 PRINT"BALANCE =$";BALANCE
```

Lesson 10 Assignment

The assignment in lesson 10 consisted of two parts. The loan balance program you wrote was to be made interactive and the loan balance and loan payment programs were to be accessed from a menu. The example below makes the loan balance payment interactive.

```
100 INPUT"ENTER LOAN AMOUNT";A
110 INPUT"ENTER INTEREST RATE";I
120 I=I/1200
130 INPUT"ENTER MONTHLY PAYMENT";P
140 INPUT"ENTER NUMBER OF PAYMENTS MADE";N
150 BALANCE=(1/(1+I) ↑ -N)*(P*(((1+I)↑-N-1)/I)+
A)
160 PRINT"BALANCE =$";BALANCE
170 END
```

Note that line 120 divides the entered interest rate by 1200. This allows interest to be entered in the normal whole number manner (12 instead of 0.12) and as annual interest rather than monthly.

The example below prints a menu on screen and includes the two loan calculation programs already written.

```
100 PRINT"PRESS KEY FOR DESIRED SELECTION"
110 PRINT TAB(10)"F1.  CALCULATE LOAN PAYMENT"
120 PRINT TAB(10)"F7.  CALCULATE LOAN BALANCE"
130 GET A$
140 IF A$=""THEN130
150 IF A$=CHR$(133)THEN 1000
160 IF A$=CHR$(136)THEN 2000
170 GOTO130

1000 INPUT"ENTER LOAN AMOUNT";A
1010 INPUT"ENTER ANNUAL INTEREST";I
1020 I=I/1200
1030 INPUT"ENTER NUMBER OF YEARS";N
1040 N=N*12
1050 PMT=A*I/(1-(1+I)↑-N)
1060 PRINT"PAYMENT=$";PMT
1070 END

2000 INPUT"ENTER LOAN AMOUNT";A
2010 INPUT"ENTER INTEREST RATE";I
```

```
2020 I=I/1200
2030 INPUT"ENTER MONTHLY PAYMENT";P
2040 INPUT"ENTER NUMBER OF PAYMENTS MADE";N
2050   BALANCE=(1/(1+I)↑-N)*(P*(((1+I)↑-N-1)/I)
+A)
2060 PRINT"BALANCE =$";BALANCE
2070 END
```

In this example lines 100-170 contain the code for printing the menu on screen and monitoring the keyboard. Lines 1000-1070 contain the coding for the loan payment calculator and lines 2000-2070 calculate loan balances.

Lesson 11 Assignment

The lesson 11 assignment was an exercise in string handling. The assignment program was to allow the user to input a single sentence using the GET statement. The program was to then count the number of words and characters in the sentence and calculate the average number of words in the sentence. The example program below would accomplish this.

```
100 GET K$
110 IF K$=""" THEN 100
120 IF K$="."THEN 200
130 PRINT K$;
140 A$=A$+K$
150 GOTO 100

200 PRINT K$
210 L=LEN(A$)
220 WD=1
230 CHAR=0
240 FOR N=1 TO L
250 IF MID$(A$,N,1)=" " THEN 400
260 CHAR=CHAR+1
270 NEXT N
280 PRINT"NUMBER OF CHARACTERS =";CHAR
290 PRINT"NUMBER OF WORDS =";WD
300 PRINT"AVERAGE CHARACTERS PER WORD =";
CHAR/WD
310 END
400 WD=WD+1
410 GOTO 270
```

In this example, lines 100-150 make up a keyboard monitor to input the sentence. Lines 100 and 110 loop until a key is pressed. Line 120 causes a jump out of the monitor to line 200 if the period key is pressed indicating the end of the sentence. If the key pressed was not a period, line 130 prints the character on screen and line 140 adds it to the string variable A$. Line 150 causes a jump back to line 100 to get the next keystroke.

Once the sentence is completely entered into the A$ variable, line 200 prints the period on screen. Line 210 sets variable L equal to the length of A$.

Line 220 sets the variable WD equal to 1. This variable will be used to hold the number of words in the sentence. We will do this by counting the number of spaces in the sentence. Since the number of words in the sentence will be one more than the number of spaces, WD is initially set to 1.

Line 230 sets the variable CHAR to 0. This variable will be used to hold the number of nonspace characters in the sentence.

Lines 240 to 270 make up a FOR/NEXT loop that will operate from 1 to L. Recall that the variable L contains the number of characters contained in A$.

Line 250 checks character N of A$ to see if it is a space. If so, line 400 adds 1 to the WD variable and line 410 causes a jump to line 270 to increment the loop. If the character examined by line 250 was not a space, line 260 increments the CHAR variable.

Once all characters have been examined by the loop in lines 240-270, the total number of words will be held in WD and the total number of nonspace characters will be held in CHAR. Line 280 prints CHAR on screen and line 290 prints WD. Line 300 calculates the average number of characters per word by dividing CHAR by WD. This result is also printed on screen. Line 310 ends the program.

Lesson 12 Assignment

The assignment in lesson 12 was to write a program that would generate a random result similating the roll of a pair of dice each time any key on the keyboard was pressed. The program example below should accomplish this.

```
100 PRINT
110 GOSUB 200
120 A=N
130 GOSUB 200
140 B=N
```

```
150 PRINT A,B
160 GET A$
170 IF A$=""" THEN 160
180 GOTO 100

200 N=INT(RND(0)*(7-1)+1)
210 RETURN
```

In this example line 100 prints a blank line on screen to separate each dice roll result. Line 110 calls the subroutine at line 200. This subroutine calculates a random whole number between 1 and 6 and stores the result in variable N. Line 120 sets variable A equal to N. Variable A is used to store the value of dice number one.

Lines 130-140 call the subroutine again to calculate the dice number two result which is stored in variable B. Line 150 prints the contents of variables A and B on the screen.

Line 160-180 make up a keyboard monitor loop. If any key is pressed, line 180 causes a jump back to line 100 to reroll the dice.

Lesson 16 Assignment

The assignment in lesson 16 was in five parts. The example solution for each part is discussed separately below:

Part One

The program below will accomplish part one of the assignment in lesson 16:

```
100 DIM A$(3,50)
110 FOR N=1 TO 50
120 A$(1,N)="0"
130 A$(2,N)="0"
140 A$(3,N)="0"
150 NEXT N
```

Line 100 dimensions the A$ array for three columns of 50 strings. In our program A$(1,N) will hold names, A$(2,N) will hold addresses, and A$(3,N) will hold telephone numbers.

Lines 110-150 fill all elements of the array with zeroes. This is done so that we can detect the first empty slot in the array by finding the first zero.

Part Two

In part two of the lesson 16 assignment a menu was to be printed on screen and a keyboard monitor constructed. The program lines below should accomplish this.

```
200 PRINT"[clear screen]"
210 PRINT TAB(15)"MENU"
220 PRINT
230 PRINT"PRESS NUMBER OF DESIRED FUNCTION"
240 PRINT
250 PRINT"     1. ENTER NAMES."
260 PRINT"     2. SEARCH NAMES."
270 PRINT"     3. ALPHABETICAL SORT."
280 PRINT"     4. QUIT."

400 GET K$
410 IF K$=""  THEN 400
420 K=VAL(K$)
430 ON  K  GOTO  1000,2000,7000,490
440 GOTO 400
490 END
```

Lines 200-280 simply print the menu on screen. Lines 400-410 make up a keyboard monitor loop. If any key is pressed the loop is broken and line 420 converts the keystroke stored in K$ into a numeric value which is stored in variable K. This value is used in the ONGOTO statement in line 430 to jump to the appropriate program section.

Line 440 causes a jump back to the keyboard monitor loop in lines 400-410 if the previous keystroke did not render a value of 1,2,3 or 4 which was usable by the ONGOTO statement. Line 490 ends the program if the QUIT option was selected from the menu.

Part Three

The program below allows the user to enter names into the array as described in the lesson 16 assignment.

```
1000 PRINT"[clear screen]"
1010 FOR N=1 TO 50
1020 IF A$(1,N)="0" THEN 1100
1030 NEXT N
1040 PRINT"ARRAY FULL."
1050 FOR N=1  TO  2000
1060 NEXT N
1070 GOTO 200
```

```
1100 PRINT"[clear screen]"
1105 B$="0"
1110 INPUT"ENTER NAME:";B$
1120 IF B$="MENU" THEN 200
1130 A$(1,N)=B$
1135 B$="0"
1140 INPUT"ENTER ADDRESS:";B$
1150 IF B$="MENU" THEN 200
1160 A$(2,N)=B$
1165 B$="0"
1170 INPUT"ENTER PHONE NUMBER:";B$
1180 IF B$="MENU" THEN 200
1190 A$(3,N)=B$
1200 IF N=50 THEN 1040
1210 N=N+1
1220 GOTO 1100
```

Lines 1000-1030 search through the name elements A$(1,N) to find the first element that contains a zero. This would be the first empty element suitable for storing a name. The number of this element is stored in variable N and a jump is made to line 1100.

If no element in the name array contains a zero, then it must be completely filled with names. Line 1040 prints the advisory message ARRAY FULL on screen. Lines 1050-1060 make up a delay loop to allow enough time for the advisory to be read before line 1070 causes the main menu to redisplay. The duration of this delay can be adjusted by changing the value held in line 1050 (currently 2000).

Lines 1100-1190 clear the screen and input names, addresses and phone numbers into the first empty slot in the array (element N from lines 1000-1030). Note that each item is first stored in a buffer string (B$). The B$ variable is checked to see if MENU was entered. If so, a jump to the main menu is performed and no element of the array is disturbed. If B$ was not MENU it is stored in the appropriate array element.

Line 1200 checks to see if the last element of the array (50) was just filled. If so, a jump to line 1040 is performed to notify the user that the array is full. If not, line 1210 increments variable N and line 1220 causes a jump back to line 1100. Another name can then be entered. In this way the user can continue to enter names until either the array is filled or MENU is entered as a name.

Note the relationship between the array elements for names, addresses and phone numbers. Picture the array as three columns

of 50 elements each. Column one contains the names, column two contains the addresses and column three contains the phone numbers. If the third element in column one contains the name HAROLD, we want the third element in column two to contain HAROLD's address and the third element in column three to contain HAROLD's phone number. In lines 1130, 1160 and 1190 we enter the data into each array column specifying the same column element by using the variable N. This relationship is important. It's O.K. to move Harolds name from element three to element 25 for instance, but his address and phone number must also move to element 25 of their respective columns in order for the relationship to be maintained.

Part Four

Part four of the assignment in lesson 16 asked you to write a program section to search the array for a particular name. The example solution below will do this.

```
2000 PRINT"[clear screen]"
2010 INPUT"ENTER NAME:";B$
2020 PRINT
2030 IF B$="MENU"  THEN 200
2040 FOR N=1  TO 50
2050 IF A$(1,N)=B$  THEN 2100
2060 NEXT N
2070 PRINT"NO SUCH NAME ON FILE"
2080 GOTO 2010

2100 PRINT A$(1,N)
2110 PRINT A$(2,N)
2120 PRINT A$(3,N)
2130 PRINT
2140 GOTO 2010
```

Line 2000 clears the screen and line 2010 allows the user to enter a name. Line 2030 causes a jump to the main menu if the user entered MENU as a name.

Lines 2040-2060 search the name column of the array for a match between the entered name and the names on file. If an exact match is found, a jump to line 2100 is performed. If no match is found, line 2070 prints the advisory message NO SUCH NAME ON FILE on the screen. Line 2080 then causes a jump back to 2010 allowing the user to enter another name.

Lines 2100-2120 print the name that was found along with the corresponding address and phone number. When this is done line 2140 causes a jump back to 2010 allowing the user to enter the next name.

Part Five

Part five of lesson 16 asked for a program section that would perform an alphabetical sort of all the names in the array. The example solution below will do this.

```
7000 PRINT"[clear screen]"
7010 PRINT"STANDBY WHILE SORTING"
7020 FOR S=1 TO 50
7030 IF A$(1,S)="0" THEN 7050
7040 NEXT S
7050 S=S-1

7100 FOR PASS=1 TO S-1
7110 FLAG=0
7120 FOR C=1 TO S-1
7130 IF A$(1,C)>A$(1,C+1) THEN GOSUB 7500
7140 NEXT C
7150 IF FLAG=0 THEN 7200
7160 NEXT PASS
7200 PRINT"ALPHABETICAL SORT COMPLETE"
7210 FOR N=1 TO 2000
7220 NEXT N
7230 GOTO 200

7500 B$(1)=A$(1,C)
7510 B$(2)=A$(2,C)
7520 B$(3)=A$(3,C)
7530 A$(1,C)=A$(1,C+1)
7540 A$(2,C)=A$(2,C+1)
7550 A$(3,C)=A$(3,C+1)
7560 A$(1,C+1)=B$(1)
7570 A$(2,C+1)=B$(2)
7580 A$(3,C+1)=B$(3)
7590 FLAG=1
7600 RETURN
```

Since we do not want to sort the empty array elements up to the beginning of the array, we need to count how many elements are actually filled with names and limit our sort to just those elements.

Lines 7000-7010 clear the screen and print a STANDBY WHILE SORTING advisory message on screen. Line 7020-7050 calculate the number of array elements that are filled by locating the first element in the array name column that contains a zero (line 7030) and then subtracting one (line 7050).

Lines 7100-7600 make up a bubble sort very similar to the one illustrated in lesson 16. The only notable difference is that in this sort when a swap is performed (lines 7500-7600), the addresses and phone numbers must be swapped along with the names.

When the sort is completed, line 7200 prints the advisory message ALPHABETICAL SORT COMPLETE on screen. After a slight delay (lines 7210-7220), line 7230 causes the main menu to redisplay.

Lesson 17 Assignment

The assignment given in lesson 17 was in five parts. Each part will be described separately in the paragraphs below.

Part One

Part one of the lesson 17 assignment asked for a program section to be added to the telephone directory program that would print a file area menu on screen and provide an accompanying keyboard monitor. The example solution below accomplishes this.

```
3000 PRINT"[clear screen]"
3010 PRINT TAB(13)"FILE MENU"
3020 PRINT
3030 PRINT"PRESS NUMBER OF DESIRED FUNCTION"
3040 PRINT
3050 PRINT"     1. FILE ARRAY."
3060 PRINT"     2. LOAD ARRAY."
3070 PRINT"     3. UPDATE EXISTING FILE"
3080 PRINT"     4. RETURN TO MAIN MENU."

3100 GET K$
3110 IF K$=""  THEN 3100
3120 K=VAL(K$)
3130 ON K GOTO 4000,4500,5000,200
3140 GOTO 3100
```

This file menu and monitor is very similar to the main menu contained in lines 200-490 of the lesson 16 assignment.

Part Two

Part two of the lesson 17 assignment asked for the creation of a program section that would store the contents of the array in a new sequential disc file. The program below does this.

```
4000 PRINT"[clear screen]"
4010 INPUT"ENTER FILE NAME";N$
4020 N$="0:"+N$+",S,W"
4030 INPUT"ENTER FILE NUMBER";F
4040 OPEN F,8,8,N$
4050 FOR N=1 TO 50
4060 PRINT#F,A$(1,N)
4070 PRINT#F,A$(2,N)
4080 PRINT#F,A$(3,N)
4090 NEXT N
4100 CLOSE F
4110 PRINT"FILING COMPLETE"
4120 FOR N=1 TO 2000
4130 NEXT N
4140 GOTO 3000
```

Line 4000 clears the screen and line 4010 allows the user to enter the file name into variable N$. Line 4020 adds the 0: and ,S,W characters to this name to make up the proper disc command for opening a new sequential file.

Line 4030 allows the user to enter a file number into variable F and line 4040 opens the file using the F and N$ variables in the OPEN statement.

Lines 4050-4090 send each element of the array to the disc file. Line 4100 closes the file. Lines 4110-4140 print the advisory FILING COMPLETED and after a short pause cause a jump to line 3000, redisplaying the file menu.

Part Three

Part three of the lesson 17 assignment asks for a program section that will load the array from a sequential disc file.

```
4500 PRINT"[clear screen]"
4510 INPUT"ENTER NAME OF FILE TO LOAD";N$
4520 N$="0:"+N$+",S,R"
4530 INPUT"ENTER FILE NUMBER";F
```

```
4540 OPEN F,8, 8, N$
4550 FOR N=1  TO  50
4560 INPUT#F,A$(1,N)
4570 INPUT#F,A$(2,N)
4580 INPUT#F,A$(3,N)
4590 NEXT N
4600 CLOSE F
4610 PRINT"FILE LOADING COMPLETE"
4620 FOR N=1 TO 2000
4630 NEXT N
4640 GOTO 3000
```

Lines 4500-4510 clear the screen and allow the user to enter the name of the desired file into N$. Line 4520 adds to N$ the characters required to make up a proper disc command for reading a sequential file. Line 4530 allows the user to enter the desired file number and line 4540 opens the file.

Lines 4550-4590 load the array from the specified sequential file and line 4600 closes the file. Lines 4610-4640 print the advisory FILE LOADING COMPLETED and after a short delay cause the file menu to redisplay.

It should be noted that the file name and number entered by the user must correspond to an exising disc sequential file.

Part Four

Part four of the lesson 17 assignment called for a program segment that would update an existing sequential file.

```
5000 PRINT"[clear screen]"
5010 INPUT"ENTER NAME OF UPDATE FILE";N$
5020 N$="@0:"+N$+",S,W"
5030 INPUT"ENTER UPDATE FILE NUMBER";F
5040 OPEN F,8,8,N$
5050 FOR N=1  TO  50
5060 PRINT#F,A$(1,N)
5070 PRINT#F,A$(2,N)
5080 PRINT#F,A$(3,N)
5090 NEXT N
5100 CLOSE F
5110 PRINT"FILE UPDATE COMPLETED"
5120 FOR N=1  TO  2000
5130 NEXT N
5140 GOTO 3000
```

This program simply saves the data from the array into an existing sequential disc file, replacing whatever was there. Note that this program is virtually identical to the one created in part two of the lesson 17 assignment. The primary difference is in line 5020 where the at symbol (@) is added to N$ so that the open statement in line 5040 will allow update of a file already on disc.

Part Five

Part five of the lesson 17 assignment simply asked that you revise the main menu program contained in lines 200-490 to access the filing features added in the lesson 17 assignment. One program line must be added and another revised.

Line 290 is added to the menu as follows:

```
290 PRINT"      5. FILE AREA"
```

Line 430 in the keyboard monitor section must be revised to allow access to the file area when the 5 key is pressed.

```
430 ON K GOTO 1000,2000,7000,490,3000
```

Lesson 19 Assignment

The lesson 19 assignment called for the addition of a printout feature to the telephone directory program that was developed in the lesson 16 and 17 assignments. The sample solution program shown below should accomplish this.

```
6000 FOR S=1 TO 50
6010 IF A$(1,S)="0" THEN 6100
6020 NEXT S
6100 OPEN 4,4,7
6110 PRINT#4,CHR$(17)
6120 FOR N=1 TO S STEP2
6130 PRINT#4,A$(1,N)CHR$(16)"40"A$(1,N+1)
6140 PRINT#4,A$(2,N)CHR$(16)"40"A$(2,N+1)
6150 PRINT#4,A$(3,N)CHR$(16)"40"A$(3,N+1)
6160 PRINT#4
6170 NEXT N
6180 CLOSE 4
6190 GOTO 200
```

In this example we want to printout only those array elements that contain names and addresses. Lines 6000-6020 determine the number of elements that contain names.

Line 6100 opens the printer file with a secondary address of 7 so that the UPPER/LOWER CASE mode can be accessed. Line 6110 sends a CHR$(17) control code to the printer to put it in UPPER/LOWER CASE mode.

Line 6120 sets the print loop to run from element 1 of the array to the element determined by lines 6000-6020. This loop is incremented in steps of two so that we can achieve the double column format desired.

Line 6130 prints the name element specified by variable N and the following element as well on the same line. Note the CHR$ (16) control code which tabs the print head to the fortieth space before printing the second element. Lines 6140 and 6150 print the address and phone elements respectively in a similar fashion. Line 6160 causes a blank line to be printed as a spacer.

Line 6180 closes the print file and 6190 causes a return to the main menu.

By adding a line 300 to the main menu as shown below, the PRINTOUT function will appear in the main menu.

```
300 PRINT"      6. PRINTOUT."
```

Additionally, line 430 must be revised so that when the 6 key is pressed the printout function is performed.

```
430 ON K GOTO 1000,2000,7000,490,3000,6000
```

Lesson 20 Assignment

The programming assignment for lesson 20 included the creation of a memory calculator using the FRE function described in lesson 13. The program lines below will calculate how much memory the program currently in the machine occupies.

```
8000 PRINT"[clear screen]"
8010 X=FRE(0)-(FRE(0)<0)*65536
8020 PRINT"MEMORY USED =";38911-X
8030 END
```

The lesson 20 assignment also asked that you revise the telephone directory program using multistatement lines and subroutines to save memory. The following pages contain a listing of the program as written in the sample solutions and a revised version that uses these memory saving techniques. These two programs are exactly equivalent as far as program operation is concerned. However, the original program is much easier to read and debug while the revised version uses 600-700 bytes less memory.

TELEPHONE DIRECTORY PROGRAM

```
 99 REM DIMENSION ARRAY AND FILL WITH ZEROES
100 DIM A$(3,50)
110 FOR N=1 TO 50
120 A$(1,N)="0"
130 A$(2,N)="0"
140 A$(3,N)="0"
150 NEXT N
199 REM PRINT MENU
200 PRINT "▼"
210 PRINT TAB(15)"MENU"
220 PRINT
230 PRINT"PRESS NUMBER OF DESIRED FUNCTION."
240 PRINT
250 PRINT"        1. ENTER NAMES."
260 PRINT"        2. SEARCH NAMES."
270 PRINT"        3. ALPHABETICAL SORT."
280 PRINT"        4. QUIT."
290 PRINT"        5. FILE AREA."
300 PRINT"        6. PRINTOUT."
399 REM KEYBOARD MONITOR
400 GETK$
410 IF K$="" THEN 400
420 K=VAL(K$)
430 ON K GOTO 1000,2000,7000,490,3000,6000
440 GOTO 400
490 END
499 REM ENTER NAMES PROGRAM SECTION
1000 PRINT "▼"
1010 FOR N=1 TO 50
1020 IF A$(1,N)="0" THEN 1100
1030 NEXT N
1040 PRINT"ARRAY FULL."
1050 FOR N=1  TO  2000
1060 NEXT N
1070 GOTO 200
1099 REM  ENTER NAMES, ADDRESSES, PHONE NUMBERS
1100 PRINT "▼"
1105 B$="0"
1110 INPUT"ENTER NAME:";B$
1120 IF B$="MENU" THEN 200
1130 A$(1,N)=B$
1135 B$="0"
1140 INPUT"ENTER ADDRESS:";B$
1150 IF B$="MENU"  THEN 200
1160 A$(2,N)=B$
1165 B$="0"
```

```
1170 INPUT"ENTER PHONE NUMBER:";B$
1180 IF B$="MENU" THEN 200
1190 A$(3,N)=B$
1200 IF N=50 THEN 1040
1210 N=N+1
1220 GOTO 1100
1999 REM SEARCH NAMES PROGRAM SECTION
2000 PRINT"■"
2010 INPUT"ENTER NAME:";B$
2020 PRINT
2030 IF B$="MENU" THEN 200
2040 FOR N=1 TO 50
2050 IF A$(1,N)=B$ THEN 2100
2060 NEXT N
2070 PRINT"NO SUCH NAME ON FILE."
2080 GOTO 2010
2100 PRINT A$(1,N)
2110 PRINT A$(2,N)
2120 PRINT A$(3,N)
2130 PRINT
2140 GOTO 2010
2999 REM PRINT FILE MENU
3000 PRINT"■"
3010 PRINT TAB(13)"FILE MENU"
3020 PRINT
3030 PRINT"PRESS NUMBER OF DESIRED FUNCTION."
3040 PRINT
3050 PRINT"        1. FILE ARRAY."
3060 PRINT"        2. LOAD ARRAY."
3070 PRINT"        3. UPDATE EXISTING FILE."
3080 PRINT"        4. RETURN TO MAIN MENU."
3099 REM FILE KEYBOARD MONITOR
3100 GET K$
3110 IF K$="" THEN 3100
3120 K=VAL(K$)
3130 ON K GOTO 4000,4500,5000,200
3140 GOTO 3100
3999 REM SAVE ARRAY TO NEW SEQUENTIAL FILE
4000 PRINT"■"
4010 INPUT"ENTER FILE NAME:";N$
4020 N$="0:"+N$+",S,W"
4030 INPUT"ENTER FILE NUMBER:";F
4040 OPEN F,8,8,N$
4050 FOR N=1 TO 50
4060 PRINT#F,A$(1,N)
4070 PRINT#F,A$(2,N)
4080 PRINT#F,A$(3,N)
```

```
4090 NEXT N
4100 CLOSE F
4110 PRINT"FILING COMPLETED"
4120 FOR N=1 TO 2000
4130 NEXT N
4140 GOTO 3000
4499 REM LOAD ARRAY FROM EXISTING SEQUENTIAL FILE
4500 PRINT"■"
4510 INPUT"ENTER NAME OF FILE TO LOAD";N$
4520 N$="0:"+N$+",S,R"
4530 INPUT"ENTER FILE NUMBER:";F
4540 OPEN F,8,8,N$
4550 FOR N=1 TO 50
4560 INPUT#F,A$(1,N)
4570 INPUT#F,A$(2.N)
4580 INPUT#F,A$(3,N)
4590 NEXT N
4600 CLOSE F
4610 PRINT"FILE LOADING COMPLETED"
4620 FOR N=1 TO 2000
4630 NEXT N
4640 GOTO 3000
4999 REM SAVE ARRAY TO EXISTING SEQUENTIAL FILE
5000 PRINT"■"
5010 INPUT"ENTER NAME OF UPDATE FILE";N$
5020 N$="@0:"+N$+",S,W "
5030 INPUT"ENTER UPDATE FILE NUMBER:";F
5040 OPEN F,8,8,N$
5050 FOR N=1 TO 50
5060 PRINT#F,A$(1,N)
5070 PRINT#F,A$(2,N)
5080 PRINT#F,A$(3,N)
5090 NEXT N
5100 CLOSE F
5110 PRINT"FILE UPDATE COMPLETED"
5120 FOR N=1 TO 2000
5130 NEXT N
5140 GOTO 3000
5999 REM PRINTOUT ARRAY ON VIC—1525 PRINTER
6000 FOR S=1 TO 50
6010 IF A$(1,S)="0" THEN 6100
6020 NEXT S
6100 OPEN4,4,7
6110 PRINT#4,CHR$(17)
6120 FOR N=1 TO S STEP2
6130 PRINT#4,A$(1,N)CHR$(16)"40"A$(1,N+1)
6140 PRINT#4,A$(2,N)CHR$(16)"40"A$(2,N+1)
```

```
6150 PRINT#4,A$(3,N)CHR$(16)"40"A$(3,N+1)
6160 PRINT#4
6170 NEXT N
6180 CLOSE 4
6190 GOTO 200
6999 REM SORT ARRAY ALPHABETICALLY
7000 PRINT"■"
7010"STANDBY WHILE SORTING"
7020 FOR S=1 to 50
7030 IF A$(1,S)="0" THEN 7050
7040 NEXT S
7050 S=S-1
7100 FOR PASS=1 TO S-1
7110 FLAG=0
7120 FOR C=1 TO S-1
7130 IF A$(1,C) > A$(1,C+1) THEN GOSUB 7500
7140 NEXT C
7150 IF FLAG=0 THEN 7200
7160 NEXT PASS
7200 PRINT"ALPHABETICAL SORT COMPLETE"
7210 FOR N=1 TO 2000
7220 NEXT N
7230 GOTO 200
7500 B$(1)=A$(1,C)
7510 B$(2)=A$(2,C)
7520 B$(3)=A$(3,C)
7530 A$(1,C)=A$(1,C+1)
7540 A$(2,C)=A$(2,C+1)
7550 A$(3,C)=A$(3,C+1)
7560 A$(1,C+1)=B$(1)
7570 A$(2,C+1)=B$(2)
7580 A$(3,C+1)=B$(3)
7590 FLAG=1
7600 RETURN
7999 REM CALCULATE AMOUNT OF MEMORY USED
8000 PRINT"■"
8010 X=FRE(0)-(FRE(0)< 0)*65536
8020 PRINT"MEMORY USED =";38911-X
8030 END

READY.
```

REVISED TELEPHONE DIRECTORY PROGRAM

```
100  DIMA$(3,50):FORN=1TO50:A$(1,N)="0":A$(2,N)="0":A$(3,
N)="0":NEXTN
200 PRINT"■";TAB(15)"MENU":PRINT:PRINT"PRESS NUMBER
OF DESIRED FUNCTION.":PRINT
210 PRINT"        1. ENTER NAMES."
220 PRINT"        2. SEARCH NAMES."
230 PRINT"        3. ALPHABETICAL SORT."
240 PRINT"        4. QUIT."
250 PRINT"        5. FILE AREA."
260 PRINT"        6. PRINTOUT."
400 GETK$:IFK$=""THEN400
410 K=VAL(K$):ONKGOTO1000,2000,7000,490,3000,6000:GOTO400
490 END
1000 PRINT"■"
1010 FORN=1TO50:IFA$(1,N)="0"THEN1100
1020 NEXTN
1040 PRINT"ARRAY FULL.":FORN=1TO2000:NEXTN:GOTO200
1100 PRINT"■":B$="0":INPUT"ENTER NAME:";B$:IFB$="MEN
U"THEN200
1130 A$(1,N)=B$:B$="0"
1140 INPUT"ENTER ADDRESS:";B$:IFB$="MENU"THEN200
1160 A$(2,N)=B$:B$="0"
1170 INPUT"ENTER PHONE NUMBER:";B$:IFB$="MENU"THEN
200
1190 A$(3,N)=B$:B$="0"
1200 IFN=50THEN1040
1210 N=N+1:GOTO1100
2000 PRINT"■"
2010 INPUT"ENTER NAME:";B$:PRINT:IFB$="MENU"THEN200
2040 FORN=1TO50: IFA$(1,N)=B$THEN2100
2060 NEXTN
2070 PRINT"NO SUCH NAME ON FILE.":GOTO2010
2100 PRINTA$(1,N):PRINTA$(2,N):PRINTA$(3,N):PRINT:GOTO2010
3000 PRINT"■";TAB(13)"FILE MENU":PRINT:PRINT"PRESS NUM
BER OF DESIRED FUNCTION"
3040 PRINT
3050 PRINT"        1. FILE ARRAY."
3060 PRINT"        2. LOAD ARRAY."
3070 PRINT"        3. UPDATE EXISTING FILE."
3080 PRINT"        4. RETURN TO MAIN MENU."
3100 GETK$:IFK$=""THEN3100
3120 K=VAL(K$):ONKGOTO4000,4500,5000,200:GOTO3100
4000 PRINT"■":INPUT"ENTER FILE NAME";N$:N$="0:"+N$+"
,S,W"
```

```
4030 INPUT"ENTER FILE NUMBER:";F:GOSUB9000:PRINT"FILING
COMPLETED"
4120 FORN=1TO2000:NEXTN:GOTO3000
4500 PRINT"■":INPUT"ENTER NAME OF FILE TO LOAD";N$:N$=
"0:"+N$+",S,R"
4530 INPUT"ENTER FILE NUMBER:";F:OPENF,8,8,N$
4550 FORN=1TO50:INPUT#F,A$(1,N):INPUT#F,A$(2,N):INPUT#F,A
$(3,N):NEXTN:CLOSEF
4610 PRINT"FILE LOADING COMPLETED.":FORN=1TO2000:NEX
TN:GOTO3000
5000 PRINT"■":INPUT"ENTER NAME OF UPDATE FILE:";N$N$=
"@0:"+N$+",S,W"
5030 INPUT"ENTER UPDATE FILE NUMBER:";F:GOSUB9000:PRIN
T"FILE UPDATE COMPLETED."
5120 FORN=1TO2000:NEXTN:GOTO3000
6000 FORS=1TO50:IFA$(1,S)="0"THEN6100
6020 NEXTS
6100 OPEN4,4,7:PRINT#4,CHR$(17)
6120 FORN=1TOSSTEP2
6130 PRINT#4,A$(1,N)CHR$(16)"40"A$(1,N+1)
6140 PRINT#4,A$(2,N)CHR$(16)"40"A$(2,N+1)
6150 PRINT#4,A$(3,N)CHR$(16)"40"A$(3,N+1):PRINT#4:NEXTN:
CLOSE4:GOTO200
7000 PRINT"■STANDBY WHILE SORTING":FORS=1TO50:IFA$(1
,S)="0"THEN7050
7040 NEXTS
7050 S=S-1:FORPASS=1TOS-1:FLAG=0:FORC=1TOS-1
7130 IFA$(1,C)>A$(1,C+1)THEN GOSUB7500
7140 NEXTC:IFFLAG=0THEN7200
7160 NEXTPASS
7200 PRINT"ALPHABETICAL SORT COMPLETE.":FORN=1TO200
0:NEXTN:GOTO200
7500 B$(1)=A$(1,C)
7510 B$(2)=A$(2,C)
7520 B$(3)=A$(3,C)
7530 A$(1,C)=A$(1,C+1)
7540 A$(2,C)=A$(2,C+1)
7550 A$(3,C)=A$(3,C+1)
7560 A$(1,C+1)=B$(1)
7570 A$(2,C+1)=B$(2)
7580 A$(3,C+1)=B$(3)
7590 FLAG=1:RETURN
8000 PRINT"■":X=FRE(0)-(FRE(0)<0)*65536:PRINT"MEMORY
USED =";38911-X:END
9000 OPENF,8,8,N$
9010 FORN=1TO50:PRINT#F,A$(1,N):PRINT#F,A$(2,N):PRINT#F,A
$(3,N):NEXTN:CLOSEF
9030 RETURN
READY.
```

A Letter From Professor Orbyte

To my students:

Congratulations! You have accepted the challenge of learning the fundamentals of BASIC programming by completing BASIC, A TUTORIAL. At this time you should have a good working knowledge of the BASIC keywords, structures, format and techniques, as well as a better understanding of your Commodore 64 computer.

Because of the effort and time you have devoted to BASIC, A TUTORIAL, I would like to award you with an honorary diploma of achievement personally signed and certified. To receive this document, simply fill out the form at the bottom of this page and send it to:

Professor Orbyte
Orbyte Software
P.O. Box 2686
Waterbury, CT 06720

and your diploma will be sent to you.

Professor Orbyte

CUT ON DOTTED LINE

PLEASE TYPE OR PRINT CLEARLY

NAME _____

ADDRESS _____

CITY _____

STATE _____ ZIP _____

To insure the receipt of your diploma your warranty/registration card must be on file at Orbyte Software.

I, _____, have completed BASIC,
SIGNATURE

A TUTORIAL and fulfilled the assignment requirements and would like to receive an honorary diploma from Professor Orbyte.

Glossary of Basic Terminology

The following is an alphabetical list of the keywords, statements, commands and functions discussed in BASIC, A TUTORIAL. The function type, format and purpose of each of these are included.

ABS

TYPE: Function-Numeric
FORMAT: ABS (expression)
PURPOSE: The ABS function returns the absolute value of the expression.

AND

TYPE: Operator
FORMAT: (expression)AND(expression)
PURPOSE: AND tests the truth or falsehood of the two expressions and returns the logical value TRUE or FALSE.

ASC

TYPE: Numeric Function
FORMAT: ASC(string)
PURPOSE: ASC returns the ASCII value of the first character of the string.

ATN

TYPE: Function-Numeric
FORMAT: ATN (number)
PURPOSE: The ATN function returns the arctangent of the number.

CHR$

TYPE: String Function
FORMAT: CHR$ (numeric expression)
PURPOSE: CHR$ returns a character equivalent to the Commodore ASCII code of the numeric expression.

CLOSE

TYPE: Statement
FORMAT: CLOSE file number
PURPOSE: CLOSE stops the datafile specified by the file number from flowing to a device.

CLR

TYPE: Command
FORMAT: CLR
PURPOSE: CLR erases all variables from memory and avails space for new variables.

CMD

TYPE: Statement
FORMAT: CMD file-number, [String]
PURPOSE: CMD switches the output device from the monitor to the specified file.

CONT

TYPE: Command
FORMAT: CONT
PURPOSE: The CONT command re-starts a program that has been stopped by either the STOP or END statements or the RUN/STOP key.

COS

TYPE: Function
FORMAT: COS (number)
PURPOSE: The COS function calculates the cosine of the number.

DATA

TYPE: Statement
FORMAT: DATA list
PURPOSE: DATA statements store information within a program.

DEF FN

TYPE: Statement
FORMAT: DEF FN name (variable) = expression
PURPOSE: DEF FN sets up a user-defined function. This function saves program space.

DIM

TYPE: Statement
FORMAT: DIM variable (subscript), variable (subscript). . .
PURPOSE: DIM allocates space for an array of variables.

END

TYPE: Statement
FORMAT: END

PURPOSE: The END statement halts execution of the program and displays the READY message.

EXP

TYPE: Numeric function
FORMAT: EXP (numeric expression)
PURPOSE: EXP calculates the value of e (logarithmic base) raised to the specified number.

FN

TYPE: Numeric Function
FORMAT: FN name (number)
PURPOSE: FN references the specified previously defined formula name, substitutes the specified number in its place and calculates the formula.

FOR . . . TO . . . [STEP . . .]

TYPE: Statement
FORMAT: FOR variable = starting expression TO limit expression STEP increment expression
PURPOSE: FOR . . . TO allows you to use a variable as a counter.

FRE

TYPE: Function
FORMAT: FRE (variable)
PURPOSE: The FRE function notifies of the amount of RAM available to the program and its variables.

GET

TYPE: Statement
FORMAT: GET variable list
PURPOSE: The GET statement reads each key typed by the user. It is best to read the keys as strings.

GET#

TYPE: (Input/Output) I/O Statement
FORMAT: GET# file number, variable list
PURPOSE: GET# works as does the GET statement except it reads characters from a device or file specified rather than from the keyboard.

GOSUB

TYPE: Statement
FORMAT: GOSUB line number
PURPOSE: The GOSUB statement is similar to the GOTO statement except it remembers where it came from in the program and returns to the next logical command following the GOSUB.

GOTO

TYPE: Statement
FORMAT: GOTO line number
PURPOSE: The GOTO statement allows a program to execute lines out of numerical order.

IF . . . THEN

TYPE: Statement
FORMAT: IF expression THEN line #
IF expression GOTO line#
IF expression THEN statements
PURPOSE: The IF . . . THEN statement evaluates conditions and takes different actions depending on the outcome of these conditions.

INPUT

TYPE: Statement
FORMAT: INPUT "PROMPT"; variable list
PURPOSE: The INPUT statement causes a "prompt" and ? to be displayed on screen, allowing the user to enter information into the computer.

INPUT#

TYPE: (Input/Output) I/O Statement
FORMAT: INPUT# file number, variable list
PURPOSE: The INPUT# statement retrieves file-stored data as a whole variable of up to 80 characters in length rather than one at a time as in the GET# statement.

INT

TYPE: Integer Function
FORMAT: INT (numeric)
PURPOSE: The INT function returns the integer value of the expression.

LEFT$

TYPE: String Function
FORMAT: LEFT$ (string, integer)
PURPOSE: This function returns a string comprised of the left-most specified number (integer) of characters of the specified string.

LEN

TYPE: Integer Function
FORMAT: LEN (string)
PURPOSE: This function returns a string comprised of the left-most specified number (integer) of characters of the specified string.

LIST

TYPE: Command
FORMAT: LIST first line - last line
PURPOSE: The LIST command allows you to look up lines in the program currently in the computer's memory. If no specific line numbers are given, the entire program is listed.

LOAD

TYPE: Command
FORMAT: LOAD"file name", device
PURPOSE: The LOAD command reads the contents of a program from tape or disk into memory.

LOG

TYPE: Floating-Point Function
FORMAT: LOG (numeric)
PURPOSE: The LOG function returns the natural logarithm of the numeric.

MID$

TYPE: String Function
FORMAT: MID$(string, numeric, numeric)
PURPOSE: MID$ returns a sub-string taken from within a larger string, the starting position of the substrings specified by the first numeric and the length of the substring specified by the second numeric.

NEW

TYPE: Command
FORMAT: NEW
PURPOSE: The NEW command clears all variables in the program currently in memory, therefore deleting the program.

NEXT

TYPE: Statement
FORMAT: NEXT counter
PURPOSE: The NEXT statement is used to establish the end of a FOR . . . NEXT loop.

NOT

TYPE: Logical Operator
FORMAT: NOT expression
PURPOSE: The NOT operator is used to reverse the normal true/false result of relational expressions, affecting only the expression to the right of it.

ON

TYPE: Statement
FORMAT: ON variable GOTO/GOSUB line #, line#
PURPOSE: The ON statement is used to specify lines (s) depending on the value of the variable.

OPEN

TYPE: (Input/Output) I/O Statement
FORMAT: OPEN file-number, device, address, "file name, type, mode".
PURPOSE: The OPEN statement opens a channel for input and/or output to a peripheral device.

OR

TYPE: Logical Operator
FORMAT: operand OR operand
PURPOSE: The OR operator connects two or more relations and returns a true or false value.

PEEK

TYPE: Integer Function
FORMAT: PEEK (numeric)
PURPOSE: The PEEK function returns in integer which is read from a memory locator.

POKE

TYPE: Statement
FORMAT: POKE location, value
PURPOSE: The POKE statement writes a one-byte binary value into a specified memory location or input/output register.

POS

TYPE: Integer Function
FORMAT: POS (dummy)
PURPOSE: The POS function notifies of the current cursor position.

PRINT

TYPE: Statement
FORMAT: PRINT variable
PURPOSE: The PRINT statement is used to write data to the screen.

PRINT#

TYPE: (Input/Output)I/O Statement
FORMAT: PRINT# file number, variable
PURPOSE: The PRINT# statement is used to write data items to a logical file. Output goes to the device-number used in the OPEN statement.

READ

TYPE: Statement
FORMAT: READ variable, variable
PURPOSE: The READ statement is used to fill variable names from constants in DATA statements.

REM

TYPE: Statement
FORMAT: REM remark
PURPOSE: The REM statement makes your programs more easily understood when LISTed as it allows you to write remarks on what each section of the program is about.

RESTORE

TYPE: Statement
FORMAT: RESTORE
PURPOSE: The RESTORE statement is used to begin reREADing the first DATA constant of a program.

RETURN

TYPE: Statement
FORMAT: RETURN
PURPOSE: The RETURN statement is used to exit from a subroutine.

RIGHT$

TYPE: String Function
FORMAT: RIGHT$ (string, numeric)
PURPOSE: The RIGHT$ function returns a substring, the length specified by the numeric, taken from the right-most end of the string argument.

RND

TYPE: Floating-Point Function
FORMAT: RND(numeric)
PURPOSE: The RND function returns a random floating-point from 0 to 1.

RUN

TYPE: Command
FORMAT: RUN line number
PURPOSE: RUN is used to start the program currently in memory. If a line number is specified, the program will start from that line. If not, it will start from the beginning.

SAVE

TYPE: Command
FORMAT: SAVE "file name", device number, address
PURPOSE: The SAVE command is used to store the program currently in memory onto a tape or diskette file.

SGN

TYPE: Integer Function
FORMAT: SGN numeric
PURPOSE: The SGN function returns an integer value depending on the sign of the numeric argument.

SIN

TYPE: Floating Point Function
FORMAT: SIN numeric
PURPOSE: The SIN function gives the sine of the numeric argument.

SPC

TYPE: Special Function Format
FORMAT: SPC (numeric)
PURPOSE: The SPC function controls the format of the data by printing the number of SPaCes specified by the numeric.

SQR

TYPE: Floating-Point Function
FORMAT: SQR (numeric)
PURPOSE: The SQR function returns the square root value of the numeric.

STATUS

TYPE: Integer Function
FORMAT: STATUS
PURPOSE: Returns a completion STATUS for the last input/output operation performed on an open file.

STEP

TYPE: Statement
FORMAT: STEP expression
PURPOSE: STEP defines an increment value for the loop counter variable in a FOR statement.

STOP

TYPE: Statement
FORMAT: STOP
PURPOSE: The STOP statement is used to stop execution of the current program and return to direct mode.

STR$

TYPE: String Function
FORMAT: STR$(numeric)
PURPOSE: STR$ gives string representation of the numeric value of the argument.

SYS

TYPE: Statement
FORMAT: SYS memory location
PURPOSE: The SYS statement allows you to mix a BASIC program with a machine language program.

TAB

TYPE: Special Function
FORMAT: TAB(numeric)
PURPOSE: The TAB function moves the cursor to a specified screen position.

TAN

TYPE: Floating-Point Function
FORMAT: TAN(numeric)
PURPOSE: The TAN function returns the tangent of the numeric value.

TIME

TYPE: Numeric Function
FORMAT: TI
PURPOSE: The TI function reads the interval timer which starts when the computer is powered.

TIME$

TYPE: String Function
FORMAT: TI$
PURPOSE: The TI$ function acts as a real clock when the computer is powered.

VAL

TYPE: Numeric Function
FORMAT: VAL(string)
PURPOSE: The VAL function returns a numeric value representing the string argument.

Shift - run/stop - attempt to load - run a program from cassette

For further details contact your local dealer or:

ORBYTE™ SOFTWARE

BASIC

A TUTORIAL

ORBYTE SOFTWARE