THE ELEMENTARY COMMODORE 128

LEARN HOW TO PROGRAM IN BASIC 7.0

BY

WILLIAM B. SANDERS

The Elementary Commodore 128

A Guide to Programming in BASIC 7.0

by
William B. Sanders, Ph.D.
San Diego State University

TM

Library of Congress Cataloging in Publications Data
Sanders, William B. 1944-
The Elementary Commodore 128
1. Computer Programming
I. Sanders, William B. 1944-
II. Title
ISBN 0-931145-09-0

Manufactured in the United States of America

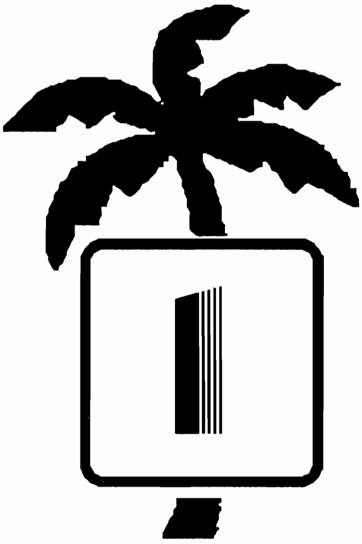
Table of Contents

Chapter 1	Introduction	1
Chapter 2	Let's Go!	15
Chapter 3	Moving Along	31
Chapter 4	Branching Out to New Frontiers	53
Chapter 5	Program Organization	75
Chapter 6	Inside the Mind of Your Commodore 128	103
Chapter 7	Sound and Music	117
Chapter 8	Using Graphics	129
Chapter 9	All About Files	161
Chapter 10	Working Your Printer	185
Chapter 11	Program Hints and Help	203
Glossary	BASIC 7.0	227
Appendix A	ASCII Charts	243
Appendix B	Screen Storage Maps	247
Product Guide		248

Acknowledgements

This book is the result of spending a lot of time with the Commodore 128. To a large extent this was due to the gracious help of Susan West of Commodore Business Machines who supplied a pre-release version of both the Commodore 128 and 1571 disk drive along with documentation. Likewise, Jim Gracely of Commodore was helpful in supplying technical assistance. The most help came from the bunch of people who make up Commodore User Groups in the San Diego area. Included in those who offered advice, hints, tips and noise are Tony Payne, Darlene Fuller, Don Johnson, Gerry White, Barbara Proudy, David Skillman, Al Wilson, Jane Campbell, and Larry and Claudia Carlson. Linda Edwards contributed directly to the chapter on graphics by creating a number of the examples that highlight BASIC 7.0's new graphic statements, functions and commands. Linda is also Microcomscribe's senior editor, and in that capacity, she took care of the book's details despite the author's best efforts to misspell every other word.

My family were kind enough not to accuse me of ignoring them too much, and I love them all. My wife Eli and sons Bill and David are my real treasures. Our dog Jingle pulled me away from my writing so that she could take me for my daily walk. These retreats from a room full of computers provided evidence that there is life beyond the cathode ray tube after all.



Introduction

This book is intended to help you operate your new Commodore 128 Personal Computer, get started programming and make life easier with your computer. It is not for professional programmers or more advanced applications. It is only the first step, and it is for beginning programmers on the Commodore 128 Personal Computer. Everything will be kept on an introductory level, but by the time you are finished, you should be able to write and use programs. The C-128 is actually three computers in one; 1) The C-128 mode, 2) the C-64 mode and 3) the CP/M mode, but we will focus on programming in the C-128 mode. The System Guide that comes with your computer gives you a good start on getting going, and you should use it as a handy reference. This book is going to focus on the actual programming and how to get the most out our BASIC 7.0 in the C-128 mode.

To best use ELEMENTARY C-128 it is suggested that you start at the beginning and work your way through step-by-step. I have tried to arrange the book so that each part and section logically follows the one preceding it. Skipping around might result in your not understanding some important aspect of the computer's operation. The only exception to this rule is the last chapter where I have put a number of suggestions for programs you might want to buy for applied purposes such as word processing and communications. When you're finished with this chapter, it would be a good idea to take a quick peek at some of the programs described in the last chapter to see if any of them fit your needs while you're learning about your C-128. You don't have to purchase any of the programs but, depending on your interests and needs, you will find some of them very useful.

Software and Hardware

Software consists of the programs which tell the computer to do different things. Whatever goes into the computer's memory is software. It is analogous to the mind or ideas. Hardware is the physical elements of the computer. The chips, keyboard, disk drive and circuit boards are hardware. Treating the hardware as the brain, any idea which goes into the hardware is the software. Software is to computers as records are to stereos. Software operates either in Random Access Memory (RAM) or Read Only Memory (ROM). (Firmware is hardware with "burned in" software.)

RAM RAM is the part of the computer's memory into which you can enter information in the form of data and programs. The more memory you have, the larger the program and more data you can enter. Think of RAM as a warehouse. When you first turn on your computer, the warehouse is just about empty but as you run programs and enter information, the warehouse begins filling up. The larger the warehouse, the more information you can store there; when it is full, you have to stop. Your C-128 has 128K of RAM. When you first turn on your system, it will indicate,

122365 Bytes Free

Only 8707 bytes of RAM are being used when you power up. (Your BASIC 7.0 is using most of it). The "K" for computerists refers to kilobytes or thousands-of-bytes , but the actual number is 1024 bytes. Your C-128 can access one megabyte (millions-of-bytes) of memory For now, all you need to know about bytes is that they are a measure of storage in computers. The more bytes, the more room you have. Think of them in the same way you would gallons, inches or meters - simply a unit of measure.

ROM A second type of computer memory is ROM, meaning "Read Only Memory." This type of memory is "locked" into your computer's chips. The difference between ROM and RAM is that whenever you turn off your computer, all information in RAM evaporates, but ROM keeps all of its information. Don't worry, though, you can save whatever is in RAM on diskettes and tape and get it back. We'll see how that is done later.

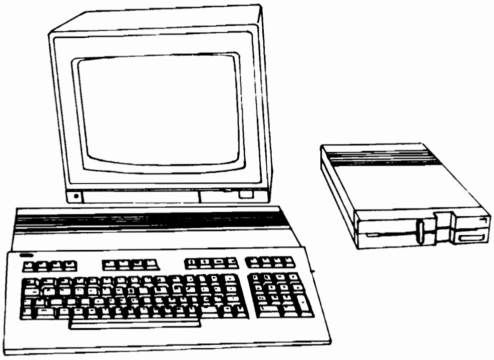
Now that you know a few terms and enough not fear your computer, let's get it cranked up and running. If you already have your computer all hooked up and working properly, you can skip the next section and go directly to the "Power On!" section of this chapter. If you don't have it set up, see your *System Guide* for instructions.

=Before you buy a printer...=

Before you buy a printer, decide on your needs and then look at the features of the different kinds before buying! And by all means, ask to see a demonstration on an C-128. Never let a salesperson convince you a certain printer will work without seeing a demonstration. Even a salesperson with the best intentions (e.g. they think a certain printer is the best for your needs) may not realize that the model cannot be interfaced to your machine. Only a demonstration is sufficient to remove all doubts! Your C-128 has a built in serial (Asynchronous Communications or RS-232) interface, but you can get a Parallel Printer attachment for your C-128, so you can use either parallel or serial printers. My own preferences lean toward parallel interfaces since there is a wider selection of parallel printers on the market, and they cost less than serial ones. Also, if you use the serial port for a printer, you cannot use it for other devices that require serial connections.

=Caution=

NEVER insert or remove cables or interfaces to your computer while the POWER IS ON! Even if you are rich and can afford to buy new chips every time you blow them by messing with the hardware on your C-128 while the power is on, you might give yourself the SHOCK of a lifetime by doing so.



Other Add-Ons. Besides the disk drive, TV/monitor, and printer, most new users do not have anything else to hook up at this point, so you can skip on to the next section. However, if you plan on expanding your C-128 or have other gadgets you bought with your system, you had better read the following section.

Modem A Modem is a device which allows your computer to communicate with other computers over telephone lines. These devices usually require that you hook up your telephone to a part of the modem, or place the phone in an acoustic sender/receiver. Not only can the modem be used to call up computer bulletin boards, but you can access such information centers as The Source and Compuserve to get everything from weather reports to airline tickets! The speed at which a modem can transfer information is called the "baud rate", with the higher values meaning higher speed. The least expensive are 300 baud modems, but I recommend at least a 1200 baud modem if you plan to put it to serious use. The 1200 baud modems cost a little more, but they are four times faster, and you can save the difference in price on phone bills.

More wonderful gadgets There are numerous other add-ons and interfaces to make the C-128 into a multifaceted machine. Special interfaces will allow you to access and use a variety of peripherals such as various disk drive systems, printers and devices made for other computers. So while the C-128 is a terrific microcomputer all by itself, it is fully expandable to make it even better.

Power On! System Check-Out

Now that you have your C-128 all set to go, you simply plug it in -- along with your TV or monitor, disk drive and printer -- turn on the power and let her rip! On the right-hand side of your computer near the back is a switch marked ON and OFF. Turn it to the ON position and turn on your TV/monitor. If everything is connected, your screen will first display the following:

```
        COMMODORE BASIC V7.0 122365 BYTS FREE
          (C)1985 COMMODORE ELECTRONICS, LTD.
                (C)1977 MICROSOFT CORP.
                  ALL RIGHTS RESERVED

READY.
```

Adjust your monitor or TV so that you have a comfortable contrast you can easily read. Directly below the READY. message is a little blinking square. It is called the "cursor," indicating your computer is waiting for you to press some keys and tell it what to do. Press the RETURN key several times and the cursor will move down the side of the screen. The message on top will scroll off the top of your screen. Your cursor is now at the bottom of the screen. To get it to the top, press the key marked F4 key and the key marked SHIFT on the right hand portion of your keyboard. Now the cursor will pop to the upper left hand corner. (Or you can just enter **SCNCLR** and press the **RETURN** key.) That done, you know your keyboard and computer are all set. We will return to the keyboard in a bit, but first let's check out your printer, disk drive, and/or cassette tape recorder. To see if your printer is working correctly, put in the following statement EXACTLY as it appears below: (<RETURN> means to press the key marked RETURN.)

```
10 OPEN4,4  <RETURN>
20 PRINT#4,"MY PRINTER IS WORKING!"
<RETURN>
30 PRINT #4 <RETURN>
40 CLOSE4   <RETURN>
```

Make certain you have written the line as it appears above. If there are even minor differences, change it so that it is precisely the same. Put the ribbon and some paper into your printer. Now turn on your printer and make certain it is "ON LINE". On some printers, a green light next to ON LINE will appear. Other printers use different notations to indicate ON LINE. For example, the C. ITOH 8510 dot matrix printer uses "SEL" to indicate it is ON LINE. Look at your printer manual to find the equivalent to ON LINE. Now enter

```
    RUN <RETURN>
```

If your printer is attached properly, it will print out the message, MY PRINTER IS WORKING! when you press the RETURN key. If a Syntax error or some other error message jumps on the screen, it means that you wrote the little test line improperly; so go back and do it again. If the system "hangs up" - the screen goes blank and nothing happens - check to make sure the printer is turned on, has the paper correctly placed in the printer and is ON LINE. If it still doesn't work, turn off the printer and the computer and review the steps for hooking up your printer.

Operating Disk Drives

Assuming your system is working correctly, let's look at a diskette.

Make sure your 1571 (or 1541) drive is connected and plugged in an electrical outlet. Turn on your drive by flipping the switch in the back. Place your DEMO disk in your drive and press the key marked F3 view the contents of the disk. (You may also enter the word DIRECTORY or CATALOG and press the RETURN key to get the same results, but let's not be dumb about this whole thing.) You will be shown the contents of the disk, but since there are so many programs on the DEMO disk, the first files will scroll off the top of the screen and you cannot see them. Press the F3 key again, but before the first material disappears, press the CONTROL and S keys simultaneously. That will stop the scrolling. Now press any other key to see the rest of the disk contents. You will notice two types of files; SEQ and PRG. The PRG files are PRoGram files. They can be DLOADed and RUN. The SEQ files are SEQuential files that must be read by a program file. (Don't worry about SEQ files now. We'll get to them in the chapter on files further down the road.)

To practice getting a file off your disk and up and running, enter the following:

```
DLOAD "DEMO <RETURN>
```

Your disk will whirr for a while than then print the READY. signal and prompt. Enter RUN <RETURN> and you will see a very good demo of what you can do with BASIC 7.0. (A shortcut for DLOAD is to press the SHIFT and F2 keys simultaneously.)

Since we're going to be creating our own programs, let's see about getting our own disk ready for use. This involves a process called **formatting** disks.

Use the following procedure:

Step 1: Place a blank diskette into your drive. There should be a notch in the diskette. Orient the diskette so that the notch is on the left. (When you place the diskette in the drive correctly, it the notch will be right next to the word DRIVE on your disk drive.)

Step 2: Close the drive by moving the drive arm downward over the slot.

Step 3: Key in the following:

```
HEADER "DISK1" <RETURN>
```

Your computer will respond with

```
ARE YOU SURE?
```

Enter 'Y' for Yes and your disk will be formatted.

Look at your directory (remember DIRECTORY), and you will see an inverse bar with the name of your disk. You are now all set to save programs you write to the disk.

(**Note**: Once a disk is formatted, you should **NOT** format it again unless you want to remove all programs from the diskette.)

=Disk Access Shortcut #8502=

Instead of having to enter DLOAD and then RUN, you can just enter RUN and the name of the program on disk, thusly,

```
    RUN "DEMO" <RETURN>
```

That cannot be done on the Commodore 64, and since old habits are hard to break, feel free to do it the old way. (You new guys don't have all the bad habits yet; so just RUN the program.)

LOADing and RUNning programs from tape

The procedure for loading and running programs from tape is quite simple. The following steps show you how:

STEP 1. Make sure your tape recorder is connected and rewind it to the beginning. If you have a tape with programs on it, use it to test loading. (A game cassette will work fine.) If you do not have a tape with a program on it, enter the following program: Note: The ENTER is usually referred to as ENTER. Now that you have used the ENTER key for a while, we will use the term ENTER whenever you are supposed to press the ENTER key.

```
NEW <RETURN>
10 PRINT "<YOUR NAME>"  <RETURN>
20 END <RETURN>
SAVE "ME" <RETURN>
```

Rewind tape and then press REC and PLAY keys simultaneously on your recorder. When the recorder stops and the READY. prompt comes on your screen, press STOP and rewind your tape. Turn off your computer.

STEP 2. Turn on your computer and when you get the cursor, write in the following:

```
LOAD "ME" <RETURN>
```

STEP 3. Press the PLAY button on your tape recorder. Your recorder will spin for a while and then stop and you will get READY. and your cursor.

STEP 4. At this point your program is all loaded and ready to go. Enter the word RUN, and your program will then execute. If you used our example program, your name will simply be printed on the screen. Rewind your tape now so that it will be ready for the next time.

=What Every Cassette User Should Know=

Get a disk drive. Even a used 1541 drive is better than a cassette tape. If you go without eating for two weeks, you can easily afford one. (You should check with your physician, first, however.)

Tape to Disk Transfer

If you have a lot of programs on tape, and you want to transfer them to disk, all you have to do is LOAD "PRG" and then DSAVE "PRG". That will save the program on disk for you. This is especially a good idea for those of you who have upgraded your system from an entry level one to a disk system. Programs will DLOAD from and DSAVE to disk a lot faster than tape. You can also make back-up programs to tape. You just DLOAD "PRG" from your disk, and SAVE "PRG" to tape. Since tapes are a lot cheapter than diskettes, this is an inexpensive way to store back-ups.

The Commodore 128 Keyboard

If you are familiar with a typewriter keyboard, you will see most of the same keys on your Commodore 128. For the most part, they do almost the same thing as your typewriter keys. If you type in the word COMPUTER, hitting the same keys you would on a typewriter, the word COMPUTER appears on the screen just as it would on paper in a typewriter. The upper-case (capital letters) and lower-case letters work exactly the same as a typewriter. On the Commodore 128, you can toggle the SHIFT LOCK by pressing the "SHIFT LOCK" key. All keys will now be capitalized except the number keys will remain the same. (That is, they will not print the "shifted" characters as on a typewriter.) When you want single upper-case characters or the symbols on the upper portion of the keys, simply press the SHIFT key and a letter to get upper-case as you would on a typewriter. This is the same as the CAPS LOCK key in the upper left hand portion of your keyboard.

Your screen can be either 40 or 80 columns if you have a monitor. (The letters are fatter in 40 columns and easier to see.) Using a television set, you can only get 40 columns; so all of our examples will use 40 columns. Of course, you cannot type just anything on the screen. If you start typing away, you'll get a ?SYNTAX ERROR every time you press RETURN unless you put in the proper commands. Otherwise, though, think of your keyboard as you would a typewriter keyboard. (Note: In most of the programming examples, we will be using upper-case only.)

Special Computer Keys. While most of the keys on your Commodore 128 look like those on a typewriter, many do not, and they are important to know about. Color codes on the C-128 keyboard reflect pressing a combination of keys. The most important are the Fn (green) and Alt (blue) keys. Whenever they are pressed with any of the other keys with the same color code function on the colored background appear. The following keys are peculiar to your computer; you will soon get used to them even though they will be a bit mysterious at first:

CONTROL On the left hand side of your keyboard is the "Control Key." By pressing the CONTROL key and certain other keys, you are able to get special characters or functions. As we encounter the need for various control characters they will be illustrated and explained. The following are some more useful ones:

> **Selected Control Codes:**
> CONTROL-1 through 8 changes text color to color indicated on number keys.
> CONTROL-9 : Turns on reverse video.
> CONTROL-0 : Turns off reverse video.
> CONTROL-X : Sets and clears tabs.

RETURN The RETURN key is something like the carriage return on a typewriter. (It's the biggest key on your keyboad.) In fact, you may see it referred to as a "Carriage Return" or "CR" in computer articles. It works like a typewriter's carriage return, because the cursor bounces back to the left-hand side of the display screen after you press it.

However, there are other uses for the RETURN key which will be discovered as you get into programming.

CURSOR CONTROL KEYS On the center-right hand top portion of your keyboard are four cursor control keys. These move the cursor in the direction indicated by the arrows. (Also, in the lower right hand sector of the keyboard are two more such keys. They are for the Commodore-64 mode, but they work in the C-128 mode as well)

CLR/HOME : If unSHIFTed, it laces cursor in upper left hand corner of screen. SHIFTed CLR/HOME clears screen and "homes" cursor.

INST/DEL: UnSHIFTed deletes character to left cursor. SHIFTed, the key will open line at cursor for new text to be inserted.

ESC In editing, quick line changes and movement are possible. In Chapter 2, we will go over the ESC combinations in the section on using your editor.

TAB KEY In the upper left hand corner of your keyboard, tabs to the right. Tab stops are made and cleared with CONTROL-X. Hit it a few times to see it jump.

FUNCTION KEYS F1-F8. When you enter BASIC the function keys' functions are listed to the screen when you enter KEY and press RETURN. They will save a lot of time for common functions, and later on we will see how to change the function keys to be used for whatever commands we want.. (If you ever want to see what the function keys do, write in KEY <RETURN>; so it would be redundant to list them all here.)

RUN/STOP This key stops a program or listings. It issues a "break", but it will not affect the program.

RESTORE. This is used with the RUN/STOP key and restore the screen to default conditions. This is a "panic" button combination short of hitting the reset key located next to the ON/OFF switch on the side of the machine.

RESET KEY. The reset key is **not** on your keyboard, but on the side of the machine next to the ON/OFF switch. It will cream memory and reset the computer to the start-up condition. It is used to exit the Commodore-64 mode.

ALT and HELP. These keys are accessed by certain applications. We won't be using them.

LINE FEED. Forces a line feed.

NO SCROLL. Works like CONTRL-S by toggling stop/start scroll. Try it with your DEMO directory lisitng and long program listings.

40/80 DISPLAY. Be careful with this key. If you lock it down, it will give 80 column display with monitor only when you reset or start up. If your computer is hooked up to a TV set, you will get a blank screen and think something is broken.

Numeric Keypad. For those of you familiar with a numeric keypad, the 14 keys to the far right of your keyboard works in that fashion. You **can** write BASIC 7.0 programs that will respond to the numeric keypad, but not in the Commodore 64 mode.

=Vat's Dot?=

If you have felt the **F** and **J** keys, and the **5** key on the numeric keypad, you will find a small bump. This 'dot' is to help you position your fingers on the correct keys. For those of you who have not had an accounting or typing class, this may come as a surprise since you only use two of your ten fingers to program, calculate and word process. (If there are 'word processors'; then the accompaning verb would be to 'word process'. Then again, maybe not, but who cares?)

Math Functions on the Keys. Some of the familiar keys have different meanings for the computer than we usually associate with the key symbols. Many are math symbols you may or may not recognize. In the next chapter, we will illustrate how these keys can be operated and discuss them in detail. For now let's just take a quick look at the math symbols.

Symbol	Meaning
+	Add
-	Subtract
*	Multiply (different from conventional)
/	Divide (different from conventional)
^	Exponentiation (power of)

In addition to some of the new representations for math symbols, other keys will be used in a manner to which you are not accustomed. As we go on, we will explain the meanings of these keys, but just to get used to the idea that your Commodore 128 has some special meanings for keys, we'll show you some more here which will have special meanings later.

Symbol	Meaning
$	Used to indicate a string variable and hexadecimal value.
:	Used to indicate "end of statement" in program.
%	Indicates an integer variable.
?	Can be used as PRINT command.

Don't worry about understanding what all of these symbols do for the time being. Simply be prepared to think in "computer talk" about symbols. As you become familiar with the keyboard and the uses and meanings of these symbols, you will be able to handle them easily, but the first step is to be aware that the different meanings exist.

Summary

This first chapter has been an overview of your new machine. You should now know how to hook up the different parts of your Commodore 128 and get it running. Also, you should be able to format a diskette, list the contents of a disk, and load and run a program from disk or from tape. Finally, you should be familiar with the keyboard and know what the cursor means.

At this point there is still much to learn, so don't feel badly if you don't understand everything. As we go along, you will pick up more and more; what may be confusing now will become clear later. Have faith in yourself and in no time you will be able to do things you never thought possible. The next chapter will get you started in learning how to program your

Commodore 128. It is vitally important that you key in and run the sample programs. Also, it is recommended you make changes in them after you have first tried them out to see if you can make them do slightly different things. Both practical and fun (and crazy!) programs are included so that you can see the purpose behind what you will be doing and enjoy it at the same time.

Let's Go!

Introduction

This chapter will introduce you to writing programs in the language known as BASIC 7.0. Commodore-128 BASIC 7.0 is different than some other versions of the language, including the BASIC in the Commodore 64 mode, and if you are already familiar with BASIC, you will find these differences. However, if you are new to the language; then you will find programming in BASIC 7.0 very simple. To get ready, turn on your computer, and when the "READY" sign comes up on your TV, you are all set to begin programming. If something else is on your screen press the RESTORE and RUN/STOP keys simultaneously, and key in NEW to clear memory.

Your very first statement! PRINT

Probably the most often used statement in BASIC is PRINT. Words enclosed in quotation marks following the PRINT statement will be printed to your screen, and numbers and variables will be printed if they are preceded by a print statement. It is used to command your computer to print output to the screen or printer from within a program or in the

Immediate Mode. You may well ask what the difference is between the Immediate and Program mode. Let's take a look.

1. Immediate Mode. The Immediate Mode executes a statement as soon as you press RETURN. For example, try the following:

```
PRINT"THIS IS THE Immediate Mode"<RETURN>
```

If everything is working correctly, your screen should look like this:

```
PRINT "THIS IS THE Immediate Mode"
THIS IS THE Immediate Mode
READY.
```

See how easy that was? Now try PRINTing some numbers, but don't put in the quote marks. Try the following:

```
PRINT 6 <RETURN>
PRINT 54321 <RETURN>
```

As you can see, numbers can be entered without having to use quote marks, but as we will see later, the actual value of the number is placed in memory rather than a "picture" of it.

2. Program Mode. This mode "delays" the execution of the commands until your program is "RUN". All commands that begin with numbers on the left side will be treated as part of a program. Try the following:

```
10 PRINT "THIS IS THE PROGRAM MODE"
```

<RETURN> - nothing happens, right?

Enter the RUN command and your screen should look like this:

```
10 PRINT "THIS IS THE PROGRAM MODE"
RUN
THIS IS THE PROGRAM MODE
```

Your very first program!
Clearing the screen and writing your name.

Let's write a program and learn two new commands. First, the new commands are SCNCLR and END. The SCNCLR statement clears the screen and places the cursor in the upper left hand corner. The END statement tells the computer to stop executing commands. From the Immediate Mode write in the SCNCLR statement to see what happens. Now, let's write a program using SCNCLR, END and PRINT. From now on, press the RETURN key at the end of each line. Throughout the rest of the book, I will no longer be putting in <RETURN> except in reference to entries in the Immediate Mode.

```
10 SCNCLR
20 PRINT "<YOUR NAME>"
30 END
RUN <RETURN>
```

All you should see on the screen is your name, READY. and the blinking cursor. Now, we're going to introduce two shortcuts that will save you time in programming and in memory. First, instead of entering new line numbers, it is possible to put multiple commands on the same line by using a colon ":" between commands. Also, instead of typing in PRINT, you can key in a question mark "?". Try the following program to see how this works.

```
10  SCNCLR
20 ? "<YOUR NAME>" : END
RUN <RETURN>
```

It did exactly the same thing, but you did not have to put in as many lines or write out the word PRINT. Neat, huh? Now, as a rule of thumb, ALWAYS begin your programs with SCNCLR. This will help you get into a habit that will pay off later when you're running all kinds of different programs. There will be exceptions to the rule, but for the most part, by beginning your programs with SCNCLR, you will start off with a nice clear screen rather than a cluttered one. While we're just getting started, it will probably be a good idea to use the colon sparingly. This is because it is easier to understand a program with a minimum number of commands in a single line. Later, when you become more adept at writing programs, and want to figure out ways to save memory and speed up program execution, you will probably want to use the colon a good deal more. Also, we want to make liberal use of the REM statement. After the computer sees a REM statement in a line, it goes on to the next line number, executing nothing until it comes to a statement that can be executed. The REM statement works as a REMark in your program lines so that others will know what you are doing and as a reminder to yourself what you have done. Just to see how it works, let's put it in our little program.

```
10 SCNCLR : REM THIS CLEARS THE SCREEN
20 PRINT "<YOUR NAME>" : END
30 REM THIS MAGNIFICENT PROGRAM WAS
CREATED BY <YOUR NAME>
```

Now RUN he program and you will see that the REM statements did not affect it at all! However, it is much clearer as to what your program is doing since you can read what the commands do in the program listing.

Setting Up A Program Using Line Numbers.

Now that we've written a little program let's take a look at using line numbers. In your first program, we used the line numbers 10, 20 and 30. We could have used line numbers 1, 2 and 3 or 0, 1 and 2 or even 1000, 2000 and 3000. In fact, there is no need at all to have regular intervals between numbers, and line numbers 1, 32 and 1543 would have worked just fine. However, we usually want to number our programs by 10's, starting at 10. You may well ask, "Wouldn't it be easier to number them 1, 2, 3, 4, 5, etc.?" In some ways maybe it would, but overall, it definitely would not! Here's why. Type in the word LIST <RETURN>, and if your program is still in memory it will appear on the screen. Suppose you want to inset a line between lines 20 and 30 that prints your home address. Rather than re-writing the entire program, all you have to do is to enter a line number with a value between 20 and 30 (such as 25) and enter the line. Let's try it, but first remove the END statement in line 20.

```
25 PRINT "<YOUR ADDRESS>"
RUN <RETURN>
```

Aha! You now have your name and address printed on the screen, and all you had to do was to write in one line instead of retyping the whole program. Now if we had numbered the program by 1's instead of 10's you would not have been able to do that since there would be no room between lines numbered 2 and 3 as there was between 20 and 30. You would have to rewrite the whole program. Now with a small program, this would not be much of a problem, but when you start getting into 100 and 1000 line programs, you'll be glad you have space between line numbers! LISTING YOUR PROGRAM. As we just saw, using the word LIST gives us a listing of our program. To make it neat, type in SCNCLR : LIST <RETURN>, and you'll get a listing on a clear screen. However, once you start writing longer programs, you won't want to list everything, but only portions. Let's examine the options available with the LIST command.

What You Write & What You Get

`LIST` Lists entire program
`LIST 20` Only line 20 is listed (or any line number you choose.)
`LIST 20-30` All lines from 20 to 30 inclusive are listed (or any other range of lines you choose).
`LIST -40` Lists from the beginning of the program to line 40 (or any other line number chosen).
`LIST 40-` Lists from line 40 (or any other line number chosen) to the end of the program. Try listing different portions of your program with the options available to see what you get. The following commands will give you some examples of the different options:

```
LIST 25
LIST 20-
LIST -20
LIST 25-30
```

=Irresponsible Programming 101=

Usually you will want to use the LIST command from the Immediate Mode as you write your program. However, you can use it from within a program. Just for fun, add the following line to your program.

```
40 LIST
```

RUN your program and see what happens. Believe it or not, there are some very practical applications we will see in some programs much later in the book. For the time being, though, it's just for fun. Now, back to some serious stuff.

RENUMBER and AUTO

In our example program, we have really made a mess. We started out with nice even line numbers and then by inserting a bunch of lines, the listing looks really sloppy. Not only that, but what would happen if we had to insert a lot of lines and ran out of room between lines? Let's clean things up with the RENUMBER command. Just enter,

```
    RENUMBER <RETRUN>
```

and LIST your program. Everything is now nice and neat, evenly numbered by 10's, and you did not have to re-write the whole program. You can do more with RENUMBER by providing parameters:

```
    RENUMBER SLN,INC,OLN
```

SLN=Starting line number
INC=Increment
OLN=Old starting number

Key in the following examples to see what happens. As you program more, the RENUMBER command will become increasingly important. It is used only in the Immediate Mode; so press the RETURN key after each entry.

```
    RENUMBER 5
    RENUMBER 4,2,5
    RENUMBER ,,8
```

Another helpful utility included in BASIC 7.0 is AUTO. When you enter AUTO and a number, your program lines will AUTOmatically be generated in increments specified by the number. For example, try the following:

```
    AUTO 10
```

Now enter,

```
    10 REM I WONDER WHAT WILL HAPPEN
```

As soon as you press RETURN, your screen will show,

```
    10 REM I WONDER WHAT WILL HAPPEN
    20
```

Your cursor will be waiting for you on line 20; all set to go.

It will save you time, and you can pick up anywhere you want after stopping. Also, it will help you get into the habit of incremented line numbers.

Saving Your Program.

Suppose you write a program, get it working perfectly and then turn off your computer. Since the program is stored in the RAM memory, it will go to Never-Never Land, and you will have to write it in again if you want to use it. Fortunately, it is a simple matter to DSAVE a program to your diskette as we saw to some extent in Chapter 1. Let's use our program for an example of DSAVEing a program to disk. Make sure your program is still in memory by LISTing it, and if it is not, re-write it. Make sure a formatted disk is in the drive and write in the following: (If you are not certain about disk formatting , review the section covering those items in Chapter 1.)

```
    DSAVE "MY PROGRAM" <RETURN>
```

The disk will start whirring and the green light will glow on the 1571 disk drive. This means the disk drive is writing your program to disk. When the red light goes out, press the F3 key to get a directory. You will be presented with a directory of the disk, and if you see MY PROGRAM in the directory that means your program has been successfully saved to disk.

Saving Programs On Tape.

To save a program to tape, put a blank cassette in your tape recorder and rewind it. Press the RECORD button and the PLAY button together on your tape recorder and write in SAVE "MY PROGRAM". The tape recorder will start spinning, the message OK will appear on the screen along with the message SAVING MY PROGRAM. When it is done, the READY prompt will reappear on the screen. Your program is now SAVEed to tape.

Retrieving Your Programs.

The best way to make sure you have SAVEd a program to disk or tape is to completely turn off your Commodore-128, and then turn it on again. Go ahead and do it. Then hit the

F3 key to view your disk directory. You should be able to see your program, (MY PROGRAM) in the directory. Now, enter DLOAD "MY PROGRAM". The disk drive will whirl for a while, and then your program will be loaded and the READY prompt will reappear. LIST and RUN your program to make sure it's the same one you DSAVEd. If it is the same, you know you have successfully DSAVEd it to disk. If you have a tape cassette, all you have to do is to press the PLAY button on your recorder and enter LOAD "MY PROGRAM." The tape will whirl looking for the program, and then load it, responding with a READY when completed. LIST and RUN it to make sure it's the correct one.

=A SAFETY NET=

As you begin writing longer programs, every so many lines, you should SAVE your program to disk or tape. In this way, if your dog accidentally trips over your cord and turns off your computer, you won't lose your program and have to shoot the offending pooch. Saves both programs and dogs.

Now that you have SAVEd and LOADed programs, let's look at another neat trick. Remembering you SAVEd your file under the name MY PROGRAM, let's change the contents of that file. First, add the following line and then LIST your program:

```
    27 PRINT "<YOUR CITY, STATE & ZIP>"
```

Your program is now different than the program you SAVEd in the file MY PROGRAM since you have added line 27. Now write in,

```
    DSAVE "@MY PROGRAM"<RETURN>
```

Clear memory with NEW, DLOAD the file MY PROGRAM and LIST it. As you can see, line 27 is now part of MY PROGRAM. All you have to do to update a program is to DLOAD it, make any changes you want, and then DSAVE it under the same file name using the "@" symbol. However, BE CAREFUL. No matter what program is in memory, that program will be DSAVEd when you enter the DSAVE command; therefore, if your disk has PROGRAM A and you write PROGRAM B, and then SAVE it under the title

PROGRAM A, it will destroy PROGRAM A and the DSAVEd program will actually be PROGRAM B. Also, if you have a really important program, it is a good idea to make a "back-up" file. For example, if you saved your current program under the file names, MY PROGRAM and MY PROGRAM BACK-UP, it would have two files with exactly the same program. To really play it safe, save the program on two different diskettes.

=I TOLD YOU SO DEPT.=

Sooner or later the following will happen to you: You will have several disks or tapes, one of which you want to format or save programs on. You will pick up the wrong diskette or cassette, one with valuable programs on it. There will be no write protect tab on the diskette or cassette, and after you format it or overwrite programs on it and blow away everything you wanted to keep, you will realize your mistake and say, "!&$#"!%&", and kick your dog. You cannot prevent that from happening at least once, believe me. Therefore, to insure that such a mistake is not irreversible, do the following: MAKE BACK-UP's. Take your ORIGINAL and put it somewhere out of reach, and when you accidentally erase a disk or tape, you can make another copy. Remember, if you fail to follow this advice, your dog will have sore ribs. Be kind to your dog.

Using Your Editor

By now you probably entered something and got a ?SYNTAX ERROR, ?SYNTAX ERROR IN 30 (referring to line 30 or any other line where an error is detected) or some other kind of error message, such as REDO FROM START, that told you something was amiss. . This occurs in the Immediate mode as soon as you hit RETURN and in the Program mode as soon as you RUN your program. Depending on the error, you will get a different type of message. As we go along, we will see different messages depending on the operation. For now, we will concentrate on how to fix errors in program lines rather than the nature of the errors themselves. This process is referred to as "editing" programs. (See APPENDIX A for a complete list of error messages.)

Deleting Lines. The most simple type of editing involves inserting and deleting lines. Let's write a program with an error in it and fix it up.

```
NEW<RETURN>
10 SCNCLR
20 PRINT " AS LONG AS SOMETHING CAN"
30 PRINT : "GO WRONG" : REM LINE WITH
 ERROR
40 PRINT "IT WILL"
50 END
RUN <RETURN>
```

If the program is written exactly as depicted above you will get a ?SYNTAX ERROR IN 30. Now, write in,

```
 30  <RETURN>
LIST <RETURN>
```

What happened to line 30?! You just learned about deleting a line. Whenever you enter a line number and nothing else, you delete the line. We already learned how to insert a line; so all you have to do to fix the program is enter the following:

```
30 PRINT "GO WRONG"
```

Now run the program. It should work fine. The error was inserting the colon between the PRINT statement and the words to be printed. Another way you could have fixed the program was simply to re-enter line 30 correctly without first deleting it, but I wanted to show you how to delete a line by entering the line number. USING THE COMMODORE-128 EDITOR. Within your COMMODORE-128 is a trusty editor. To see how to work with your editor, we'll write another bad program and fix it. OK, write the following program and RUN it.

```
NEW <RETURN>
10 SCNCLR
20 PRINT "IF I CAN GOOF UP A PROGRAM "
30 PRINT "I CAN" : FIX IT: REM BAD LINE
40 END
RUN <RETURN>
```

All right, you got a ?SYNTAX ERROR IN 30. To repair it, instead of rewriting line 30 do the following:

STEP 1. LIST your program.

STEP 2. Using the arrow cursor keys at the top of your keyboard, "walk" the cursor to LINE 30.

STEP 3. Now "walk" the cursor to the right until it is just to the right of the first colon.
STEP 4. Press the INST/DEL until the colon and quote mark after CAN" disappear.

STEP 5. Press the right arrow cursor key until the cursor is right over the colon. Now press SHIFT INST/DEL and the colon will jump a space to the right.

STEP 6. Now, simply enter a quotation mark after to "T" in the word "IT" in the space you INSerTed with your editor. Press RETURN and you're all finished.

LIST the program again. Line 30 should now be correct. Now RUN the program. You should have seen the statement, IF I CAN GOOF UP A PROGRAM I CAN FIX IT.

Let's learn more about the editor. Put in the following program: (Remember, in BASIC, we can use question marks to replace PRINT statements. If you LIST the program before you run it, you will see that all of the question marks have magically been transformed to PRINT statements.)

```
10 SCNCLR
20 ? "SOMETIMES I LIKE TO WRITE LONG,
  LONG, LONG, LONG LINES " : WHEW!
30 ?"AND SOMETIMES I LIKE SHORT LINES"
40 END
LIST <RETURN>
{See what happened to the question marks.}
 RUN <RETURN>
```

OK, after you ran the program it went El Bombo. The problem was that we stuck in that WHEW! without a PRINT statement or quote marks after the colon had terminated the line, or, alternatively, a REM statement before WHEW!. To repair it, LIST the program, "walk" the cursor up to line 20 using the arrow keys and starting at line 20 retrace the line up to where the mistake was made. To make it simple, remove the second quote mark, and leaving the colon in place, add a quote mark after the word WHEW!. Since the colon is now inside the quote marks, it will be printed as part of the PRINT statement and be ignored as a line termination statement. Press RETURN. Now RUN the program. Now, let's take a look at a feature of the COMMODORE-128 editor that might cause some problems. Enter the following BUT DO NOT HIT RETURN!!!!:

```
 20 PRINT "I LIKE TO COMPUUUUUUT
```

Whoops! There's a mistake, but you haven't finished the line. No sweat. Just press the arrow keys and back the cursor over the multiple "U's" and re-enter it correctly. However, you find that when you press the arrow key instead of walking the cursor, you get inverse vertical lines or brackets. With the up/down arrows you get big blue dots and inverse "Q's". What's going on!?? Not so elementary, Watson. As we noted in Chapter 1, the COMMODORE-128 gives you the option of printing those inverse characters inside a set of quotation marks, and to make them, you have to press the arrow keys. To make repairs, simply press RETURN and then using the arrow keys walk up and make the repairs. As you will see, the arrow keys are now working fine, even inside the quotation marks. (HINT: Let's face it, it would have been a lot easier simply to press the INST/DEL key a bunch of times to get rid of those offending "U's", but then you would never have learned why your arrow key went nuts inside the quotation marks.)

=Watch Out For 'RUNDY'=

After editing with the COMMODORE-128, I have often entered RUN over the READY prompt, ending up with "RUNDY". Of course, instead of having the program RUN, it gives a ?SYNTAX ERROR. On some computers, as soon as you press RETURN, the remaining characters on the line are forgotten if the cursor has not been passed over them. Therefore, if you are used to other kinds of computers, watch out for RUNDY!

More Editing. Let's do a few more things with your editor before going on. We'll practice some more with inserting characters and numbers, but we will also see how to do edit groups of characters. So, let's see how we can use the editor to do more with "insertions." Try the following little program:

```
10 SCNCLR
20 PRINT "NOW IS THE TIME FOR ALL GOOD
   MEN";
30 PRINT "TO COME TO THE AID OF THEIR
   COUNTRY"
40 END
```

So far so good, but you meant to include women as well as men in line 20. You could retype the entire line, but all you really need to add is AND WOMEN after MEN. Also, it's really boring to have everything in upper case. Let's change the line to include women and make it both upper and lower case:

STEP 1. Press the "COMMODORE" and SHIFT simultaneously, and everything will go to lower case characters.

STEP 2. "Walk" the cursor up to the beginning of the LINE 30 using the arrow keys and then place the cursor to the right of the first quotation mark.

STEP 3. Press the SHIFT and INST/DEL keys to make enough spaces to include "and women," and enter "and women."

STEP 4. To make the sentence correct, place the cursor over the "n" in "now" in LINE 20 and press SHIFT and "N" to capitalize the first letter of the sentence.
After these repairs, you now have upper and lower case, and when you RUN your program it should read;

Now is the time for all good men and women to come to the aid of their country.

You will save yourself a great deal of time if you use the editor rather than retyping every mistake you make. Therefore, to practice with it, there are a several pairs of lines below to repair. The first line shows the wrong way and the second line in the pair shows the correct way. Since "little" things can make a big difference, there are a number of changes to be made. However, as you will soon see, those little mistakes are the ones we are most likely to get snagged on. Practice on these examples until you feel comfortable with the editor - time spent now will save you a great deal later.

Editor Practice

```
50 PRINT PEOPLE ARE SMARTER THAN COMPUTERS
50 PRINT"PEOPLE ARE SMARTER THAN COMPUTERS"

10 PRINT SCNCLR
10 SCNCLR

80 PRINT "A GOOD MAN IS HARD TO FIND"
80 PRINT "A GOOD PERSON IS HARD TO FIND"

40 SCNCLR PRINT "WE'RE OFF!
40 SCNCLR : PRINT "WE'RE OFF!"
```

If you fixed all of those lines, you can repair just about anything. Once you get the hang of it, it's quite simple.

Using the ESC key with Editing

For those of you used to editing on the Commodore 64, you will find the ESC key editing a big time-saver. In Appendix I of your *System Guide* there is a full set of ESC codes, and here we are going to focus only on those that are the most useful and list them in terms of how useful each is.

Esc Key+	Edit Operation
A	Start auto-insert
O	Disable auto-insert
J	Move to start of current line
K	Move to end of current line
D	Delete current line*
I	Insert blank line at cursor position
Q	Erase to end of current line
P	Erase to beginning of current line
@	Clear to end of screen

*Deletes line from screen but not from memory.

To get the hang of the ESC key, go back and try them out on the editor practice you've already done to see if they save you time. When you're finished with that, try the following:

```
    10 PRINT "I LIKE MY 128"
    10 PRINT "I LIKE MY COMMODORE 128"

    10 PRINT "HER'S A PROBLEM HERE
       SOMEWHEREEEE"  END
    10 PRINT "THERE'S A PROBLEM HERE
       SOMEWHERE" : END
```

=Free Tip For Beginners=

Nobody remembers everything about programming. What I have found useful is to put little yellow stick-um tags in my computer books to mark pages with some valuable stuff to look up; like ESC codes. On the tags, I write what the tag marks, and looking stuff up is a lot easier.

Elementary Math Operations

So far all we've done is to PRINT out a lot of text, but that isn't too different than having a fancy typewriter. Now, let's do some simple math operations to show you your computer can compute! Enter the following:

```
    SCNCLR
    PRINT 2 + 2
```

This is what your screen should look like now:

```
    PRINT 2 + 2
    4
```

Big deal, so the computer can add - so can my $5 calculator and my 11 year old kid. Who said computers are smart? The programmer (you) is who is smart. Ok, so let's give it a little tougher problem.

```
    SCNCLR
    PRINT 7.87 * 123.65
```

Still nothing your calculator can't do, but it'd be a little rough on the 11 year old. As we progress, we can include more and more aspects of mathematical problems, and in the next chapter, we will see how we can store values in variables and a lot of things that would choke your calculator. For now, though, all we'll do is to introduce the format of mathematical manipulations. The "+" and "-" signs work just as they do in regular math, and the "x" is replaced by "*" (asterisk) for multiplication and "-" is replaced by the "/" (slash) for division. As we begin dealing with more and more complex math, we will need to observe a certain order in which problems are executed. This is called "precedence." Depending on the operations we use, and the results we are attempting to obtain, we will use one order or another. For example, let's suppose we want to multiply the sum of two numbers by a third number - say the sum of 15 and 20 multiplied by 3. If you entered

```
    3 * 15 + 20
```

you would get 3 multiplied by 15 with 20 added on. That's not what you wanted. The reason for that is precedence - multiplication precedes addition. To help you remember the precedence, let's write a little program you can run and then play with some math problems in the Immediate mode to see the results and refer to your "Precedence Chart" on the screen. (This little program is quite handy; so save it to disk or tape to be used later.)

```
    10 SCNCLR
    20 PRINT "1. - (MINUS SIGNS FOR NEGATIVE
       NUMBERS - NOT SUBTRACTION)"
    30 PRINT "2.  ↑ (EXPONENTIATION)"
    40 PRINT "3. * / (MULTIPLICATION AND
       DIVISION)"
    50 PRINT "4. + - (ADDITIONS AND
       SUBTRACTIONS)"
    60 PRINT "NOTE: ALL PRECEDENCE IS FROM
       LEFT TO RIGHT"
    70 PRINT "YOUR COMPUTER FIRST EXECUTES
       THE NUMBERS IN PARENTHESES, WORKING
       ITS WAY FROM THE INSIDE OUT IN
       MULTIPLE PARENTHESES."
```

Try some different problems and see if you can get what you want. RE-ORDERING PRECEDENCE. Once you get the knack of the order in which math operations work, there is a way to simplify organizing math problems. By placing two or more numbers in PARENTHESES, it is possible to move them up in priority. Let's go back to our example of adding 15 and 20 and then multiplying by 3, but this time we will use parentheses.

```
    PRINT 3 * (15 + 20)
```

Now since the multiplication sign has precedence over the addition sign, without the parentheses, we would have gotten 3 times 15 plus 20. However, since all operations inside parentheses are executed first, your computer FIRST added 15 and 20 and then multiplied the sum by 3. If more than a single set of parentheses is used in an equation, then the innermost is executed first, working its way out.

=The Parentheses Dungeon=

To help you remember the order in which math operations are executed within parentheses, think of the operations as being locked up in multi-layer dungeon. Each cell represents the innermost operation, and the cells are lined up from left to right. Each "prisoner" is an operation surrounded by walls of parentheses. To escape the dungeon, the prisoner must first get out of the innermost cell, then the prisoner goes to his right and releases any other prisoners in their cells. Then they break out of the "cell-block" and finally out into the open. Unfortunately, since operations are "executed", this is a lethal analogy for our poor escaping "prisoners." Do some of the examples and see if you can come up with a better analogy.

The following examples show you some operations with parentheses.

```
PRINT 20 + (10 * (8 - 4))
PRINT (12.43 + 92) / (3 ↑ (11 - 3))
PRINT (22 * 3.1415) * (22 * 3.1415)
PRINT ((16 / 4) * (3 + 5)) / 18
PRINT 19 + 2 * (51 / 3) - (100 / 14)
```

Now, try some of these problems in the proper format expected by your computer:

Multiply the sum of 4 , 9 and 20 by 15.

Multiply 35 by 35 and the result by pi {SHIFT ↑} The vertical arrow/pi key is located between the asterisk and RESTORE keys. (You realize that this will compute the area of a circle with a radius of 35, and to find the area of any other circle, just change 35 to another value.) Pi (SHIFT ↑) is treated just like any other number you enter, but to save time, you only need a single key. Pretty neat, huh?

Add up the charges on your long distance calls and divide the sum by the number of calls you made. This will give you the average expense of your calls. Remember, though, you have to do this in one set of statements in a single line. Do the same thing with your check book for a month to see the average (mean) amount for your checks.

Summary

This chapter has covered the most basic aspects of programming, and at this point you should be able to use the editor in your Commodore 128, write commands in the Immediate and Program (deferred) modes. Also, you should be able to manipulate basic math operations. However, we have only just begun to uncover the power of your computer, and at this stage, we are treating it more as a glorified calculator than a computer. Nevertheless, what we have covered in this chapter is extremely important to understand, for it is the foundation upon which your understanding of programming is to be built. If there are parts you do not understand, review them before continuing. If you still do not understand certain operations after a review, don't worry. You will be able to pick them up later, but it is still important that you try and get everything to do what it is supposed to do and what you want it to do. The next chapter will take us into the realm of computer programming and increase your understanding of your Commodore 128 considerably. If you take it one step at a time, you will be amazed at the power you have at your fingertips and how easy it is to program. Also, we will be leaving the realm of calculator-like commands and getting down to some honest-to-goodness computer work. This is where the fun really begins.

Moving Along

Introduction

In the last chapter, we saw how to get started in executing statements in both the Immediate and Program mode. From now on we will concentrate our efforts on building from the foundation set in Chapter 2 in the Program mode, tying various statements together in a program. We will, however, use the Immediate mode to provide simple examples and to give you an idea of how a certain statement works. Also, as we learn more and more statements, it would be a good idea if you started saving the example programs on your disk or cassette so that they can be used for review and a quick "look-up" of examples. Use file names that you can recognize, such as VARIABLE EXAMPLE or HOW TO SUBROUTINES, and REMEMBER each file has to have a different name; so be sure to number example file names(e.g. ARRAYS 1, ARRAYS 2, etc.)

Variables

Perhaps the most single important computer function is in the use of variables. Basically, a variable is a symbol that can have more than a single value. If we say, for example, X = 10, we assign the value of 10 to the variable we call "X". Try the following:

```
X = 10 <RETURN>
READY.
PRINT X <RETURN>
```

Your computer responded,

```
10
```

Now type in,

```
X=86.5 <RETURN>
READY.
PRINT X <RETURN>
```

This time you got,

```
86.5
```

Each time you assign a value to a variable, it will respond with the last assigned value when your PRINT that variable. Now try the following:

```
X = 10 <RETURN>
Y = 15 <RETURN>
PRINT X + Y <RETURN>
```

And your COMMODORE-128 responded with,

```
25
```

As you can see, variables can be treated in the same way as math problems using numbers. However, instead of using the numbers, you use the variables. Now let's try a little program using variables to calculate the area of a circle.

```
10 SCNCLR
```

```
20 PI = π
30 REM USE THE "PI" CHARACTER RIGHT NEXT
   TO THE 'RESTORE' KEY.
40 R = 15 :
50 REM R IS THE RADIUS OF OUR CIRCLE
60 PRINT PI * (R * R) :
70 REM LINE 60 GIVES US THE SQUARE OF PI
   TIMES THE RADIUS
80 END
```

When you RUN the program, you will get the area of a circle with a radius of 15. If you change the value of "R" in line 40, it is a simple matter to quickly calculate the area of any circle you want! Since our example "squares" a result, why don't we use our exponential sign "↑". Change line 40 to read:

```
40 PRINT PI * (R ↑ 2) : REM SAME KEY AS
THE "PI" SIGN BUT YOU DON'T SHIFT TO
 PRINT IT.
```

That saves typing, doesn't it? RUN the program again and see if you get the same results. You should. Also, change the value of R to see the different areas of circles.

Variable Names

When you name a variable, the computer only looks at the first two characters. For example, if you name a variable NUMBER, all your computer is interested in is NU. Try the following:

```
NAUGHTY = 44
PRINT NA
```

You got 44 even though you only entered the first two characters of the variable you called NAUGHTY. Now try this next one:

```
NUMBER = 1986
PRINT NUDE
```

The value 1986 is printed because the only characters of interest to the computer are still the first two; so even if you undress NUMBER you still get 1986! Now it may seem the best thing to do is to use variable names with only two characters, and while you're getting used to variables, that's probably not a bad idea. However, as you get into more and more sophisticated programs, it helps to use variable names that are descriptive. For example, the following program uses MEAN as a descriptive variable name:

```
10 SCNCLR
20 A = 15 : B = 23 : C = 38
30 MEAN = (A + B + C ) / 3
40 PRINT MEAN
50 END
```

If the above program were a hundred or more lines long, you would know what the variable MEAN does - it calculates a "mean." Now you'd have to be careful not to have another variable named MEATBALL or some other name beginning with "ME", but it would certainly make it easier to understand what it does. Other considerations in naming variables include not using "reserved words" (i.e. programming statements) or variables, and beginning variable names with a letter. There are only 8 reserved variables, **DS,DS$,ER,ERR$,EL,TI, TI$** and **ST**. Furthermore, all words used as program statements are also reserved. Let's look at some examples of what is and what is not a valid variable name:

`PRINT = 987` (Invalid name since PRINT is a reserved word.)

`R1 = 321` (Valid name since first character is a letter.)

`1R = 55` (Invalid since first character is not a letter.)

`FORT = 222` (Invalid since variable name contains reserved word FOR.)

`PR = 99` (Valid name, for even though reserved word PRINT begins with PR, only part of the reserved word is used in variable name.)

`TO = 983` (Invalid name since TO is a reserved 2 character word.)

`TIE = 99999999` (Invalid since TI is a reserved variable for time.)

`ILOVEAPARADE = 10` (Valid name, but really dumb.)

It is also possible to give values to variables with other variables or a combination of variables and numbers. In our example with the variable MEAN we defined it with other variables. Here are some more examples:

```
T = A * (B + C)
N = N + 1
SUM = X + Y + Z
```

Types of Variables

Real Variables. So far we've only used "real" or "floating point" variables in our examples. Any variable that begins with a capital letter and does not end with a dollar sign ($) or percentage sign (%) is a real variable. The value for a real variable can be from + or -2.93873588E-39 to + or -1.70141183E+38. The "E" is the scientific notation for very big numbers. For the time being, don't worry about it, but if you get a result with such a letter in a numeric result, get in touch with a math instructor. At this juncture, figure you can enter numbers in their standard format from 0.01 to 999,999,999. (If your checkbook debit or income tax payments have a scientific notation in them, leave the country.) Think of real variables as being able to hold just about any number you would need along with the decimal fractions.

Integer Variables. Integer variables contain only "integer" or "whole" numbers - ones without fractions. The following are some examples:

```
AB% = 345
K% = R% + N%
ADD% = ADD% + NUM%
WXY% = 88 + LR%
```

The values of integer variables can range from - 32767 to + 32767, and, like real variables, only the first two characters are read. However, the "%" is always read, no matter how many characters are used. So, a variable named WA% is the same as WAX%. Also, a variable named ABC is different from one named ABC%; therefore, both variables could be used in the same program and each be considered unique. As they have a lower range than real variables, integer variables have limited applications; however, integer variables take up less memory and execute faster than real variables and so they have many useful applications. They can be used in mathematical operations in the same way as are real variables, but since they do not store fractions, operations using division and similar fraction operations must be done with care. Try some of the following operations from the Immediate mode to see how they work:

```
A% = 15 : B% = 21 : C% = B% + A%
PRINT C% <RETURN>
36
LL% = 17 : JJ% = LL% / 5
PRINT JJ% <RETURN>
3
Z% = -11 : XY% = Z% + 51
PRINT XY% <RETURN>
40
```

String Variables. String variables are extremely useful in formatting what you will see on the screen, and like real and integer variables, they are sent to the screen by the PRINT statement. However, rather than printing only numbers, string variables send all kinds of characters, called "strings", to the screen. String variables are indicated by a dollar sign ($) on the end of a variable. For example, A$, BAD$, G$, and PULL$ are all legitimate string variables. (In computer parlance, we use the term "string" for the dollar sign. Thus, our examples would be called "A string", "BAD string", etc.) String variables are defined by placing the "string" in quotation marks, just as we did with other messages we printed out. Let's try out a few examples from the Immediate mode:

```
ABC$ = "ABC"
PRINT ABC$ <RETURN>
G$ = "BURLESQUE"
PRINT G$ <RETURN>
KAT$ ="CAT"
PRINT KAT$ <RETURN>
NUMBER$ = "123456789"
PRINT NUMBER$ <RETURN>
B1$ = "5 + 10 + 20"
PRINT B1$ <RETURN>
```

In the same way that real and integer variables only use the first two characters, a string variable must begin with a letter and use non-reserved words. More importantly, you probably noticed in our examples that numbers in string variables are not treated as numbers, but rather as "words" or "messages." For example, you may have noticed that when you PRINTed B1$, instead of printing out "35" (the sum of 5, 10 and 20), B1$ printed out exactly what you put in quotes, 5 + 10 + 20. Do not attempt to do math with string variables. (In later chapters, we'll see some tricks to convert string variables to numeric -real or integer- variables, but for now just treat them as messages.)

Let's put all of our accumulated knowledge together and write a program that uses variables. We will start a little program that will allow you to subtract a check from your check book and print the amount. This program will be the beginning of something we will later develop to give you a handy little program with which to do check book balancing.

```
10 SCNCLR
20 BALANCE =1234.56
30 REM ANY BALANCE (BA) IS A REAL
VARIABLE
40 CHECK = 29.95
50 REM WHAT YOU PAID FOR SOME SOFTWARE
55 REM CHECK (CH) IS A REAL VARIABLE.
60 B$ = "YOUR BEGINNING BALANCE IS $"
70 C$ = "YOUR CHECK IS FOR $"
80 NB$ = "YOUR NEW BALANCE IS $"
90 REM B$, C$ AND NB$ ARE STRING
VARIABLES
100 PRINT B$;BALANCE
110 PRINT C$;CHECK
120 N = BALANCE - CHECK
130 PRINT NB$; N
140 END
```

Since this is a fairly long program for this stage of the game, make sure you put in everything correctly. For the computer, it is critical that you distinguish between commas, semi-colons, periods, etc. Also, save it to disk. To play with it, change the values in lines 20 and 40. Let's quickly review what we have done.

STEP 1. First we defined the real variables "BALANCE" and "CHECK" (which your COMMODORE-128 read as BA and CH since it only cares about the first two characters.)

STEP 2. Then we defined string variables B$, C$, and

NB$ to use as labels in screen formatting.

STEP 3. Finally, we printed out all of our information using our variables, with one new variable, "N" defined as the difference between BALANCE and CHECK.

Note how we formatted the "OUTPUT" (what you see on your screen) of our PRINT statements. The semi-colon ";" between the variables accomplished two things:

(1) it told the computer where one variable ended and the next began, and
(2) it told the computer to PRINT the second variable right after the first one.

Thus, it took the string variable NB$

```
YOUR NEW BALANCE IS $#
```

and stuck the value of the real variable N right after the dollar sign (exactly where we placed the pound sign #). Later we will go more into the formatting of OUTPUT, but for now let's take a quick look at using punctuation in formatting text. We will use the comma "," and semi-colon ";" and "new line" to illustrate basic formatting. Put in the following little program:

```
NEW <RETURN>
10 SCNCLR
20 A$ ="HERE" : B$="THERE" : C$= "WHERE"
30 PRINT A$;: PRINT B$;: PRINT C$;: REM
   SEMI-COLONS
40 PRINT
50 PRINT A$,: PRINT B$,: PRINT C$,: REM
   COMMAS
60 PRINT
70 REM A 'PRINT' BY ITSELF GIVES A VERT-
   ICAL 'SPACE' IN FORMATTING
80 PRINT A$ : PRINT B$ : PRINT C$ : REM
   'NEW LINES'
90 END
```

Now RUN the program. As you should see, the little differences in lines 30, 40, and 50 made big differences on the screen. The first set is all crammed together, the second set is spaced evenly across the screen, and the third set is stacked one on top of the other. As we saw in the previous program, semi-colons put numbers and strings right next to one another. However, using commas after a PRINTed variable will space output in groups of four across the screen, and using "new lines" in the form of colons or new line numbers will make the output start on a new line. A PRINT statement all by itself will put a vertical "linefeed" between statements. Try the following little program to see how PRINT statements all by themselves can be used.

```
NEW <RETURN>
10 SCNCLR
20 PRINT "WHENEVER YOU PUT IN A PRINT
    STATEMENT";
25  REM NOTE PLACEMENT OF SEMI-COLON
30 PRINT " ALL BY ITSELF, IT GIVES A
   'LINEFEED'."
40 PRINT
50 PRINT "SEE WHAT I MEAN?"
60 END
```

Play with commas, semi-colons, and "new lines" with variables and string variables until you get the hang of it. They are very important and are the source of program "bugs."

=Bugs and Bombs=

We've mentioned "bugs" and "bombs" in programs but never really explained what they meant. "Bugs" are simply errors in programs that either create ?SYNTAX ERRORs or prevent your program from doing what you want it to do. "Debugging" is the process of removing "bugs." "Bombing" is what your program does when it encounters a "bug." This is all computer lingo, and if you use it in your conversations, people will think you really know a lot about computers or have a bug in your personality.

Input and Output (I/O)

Input and output, often referred to as I/O, are ways of putting things into your computer and getting it out. Usually we put

IN information from the keyboard, save it to disk or tape, and then later put it in from the disk drive or cassette recorder. When we want information OUT of the computer, we want it to go to our screen or printer. This is what I/O means. So far, we have entered information IN the computer from the keyboard either in the Program or in the Immediate mode. Using the PRINT statement, we have sent information OUT to the screen. However, there are other ways we can INPUT information with a combination of programming and keyboard commands. Let's look at some of these ways and make our CHECKBOOK program a lot simpler to use.

INPUT

The INPUT statement is placed in a program and expects some kind of response from the keyboard and then a RETURN. (A RETURN alone will also work, but the response is read as "".) It must be part of a program and cannot be used from the Immediate mode. (If attempted from the Immediate mode, there will be an ?ILLEGAL DIRECT ERROR message.) Let's look at a simple example:

```
NEW <RETURN>
10 SCNCLR
20 INPUT X :
25 REM 'X' IS A NUMERIC VARIABLE SO ENTER A
NUMBER
30 PRINT X
40 END
```

RUN the program and your screen will go blank and a "?" along with a blinking cursor will sit there until you enter a number and then the computer will PRINT the number you just entered. Really interesting, huh? Let's try INPUTing the same information but using a slightly different format. The nice thing about INPUT statements is that they have some of the same features as PRINT statements for getting messages on the screen. Look at the following program:

```
NEW <RETURN>
10 SCNCLR
20 INPUT "ENTER YOUR AGE "; X
30 SCNCLR : PRINT : PRINT : PRINT
40 PRINT "YOUR AGE IS "; X
```

Now RUN the program; you will see that the presentation is a little more interesting. Also notice we did not put an END statement at the end of the program. In COMMODORE-128 it is not necessary to enter an END statement, but it is usually a good idea to do so. As we get into more advanced topics, we will see that our program can jump around, and the place we want it to END will be in the middle, and we will need an END statement so that it will not crash into an area we don't want it to go. So, while an END statement really has not been necessary up to now, it is nevertheless a good habit to develop. Let's soup up our program a little more with the INPUT statement.

```
NEW <RETURN>
10 SCNCLR
20 INPUT "ENTER YOUR NAME -> "; NA$
30 PRINT
40 INPUT "ENTER YOUR AGE  -> "; AG%
50 PRINT
60 INPUT "PRESS <RETURN> TO CONTINUE "; RT$
70 SCNCLR: ? : ? : ? : ? : ? :
75 REM USING "?" AS SUBSTITUTES FOR PRINT
80 PRINT NA$; " IS "; AG% ; " YEARS OLD. "
85 REM BE CAREFUL WHERE YOU PUT YOUR
87 REM QUOTE MARKS & SEMI-COLONS IN LINE 80
90 END
```

Now we're getting somewhere. You can enter information as numeric or string variables and the OUTPUT is formatted so you know what's going on. As your programs become larger and more complicated, it is very important to connect your string variables and numeric variables in such a way that it is easy to see what the numbers on the screen mean. Let's face it, a computer wouldn't be very helpful if it filled the screen with numbers, and you did not know what they meant! Line 60 is the format for a pause in your program. RT$ doesn't hold any information, but since INPUT statements expect something from the keyboard and a variable, RT$ (for RETURN) is as good as any.

GETKEY and GETing Information

The GETKEY and GET statements are something like the INPUT statement, except they is executed as soon as you hit a key. GETKEY is the easiest to use since you just have to key in GETKEY and a variable. Using a string variable, the user can hit any key.

```
NEW <RETURN>
10 SCNCLR
20 PRINT "HIT ANY KEY, JACK"
30 GETKEY A$
40 PRINT "YOU HIT=>";A$
```

To see how GET works try the following program. You should note that to be of use, GET must be put into a little "loop" routine.

```
NEW <RETURN>
10 SCNCLR
20 ? : ? : ? : ?
30 PRINT " ENTER A NUMBER FROM 1-9 ";
40 GET N: IF N < 1 OR N > 9 THEN 40
45  REM NOTE FORMAT IN LINE 40
50 ? : ?
60 PRINT " HIT ANY KEY TO CONTINUE ";
70 GET K$: IF K$ = "" THEN 70
75 REM GETKEY WOULD BE SIMPLER IN LINE 70
80 SCNCLR: ? : ? : ? : ?
90 PRINT "YOUR NUMBER IS -->" ; N
100 END
```

Notice that as soon as you hit a key, the GET statement is executed. With an INPUT statement you first enter information and then press the RETURN key before the program executes. The good thing about the GET statement is that it is a faster way to enter and execute from the keyboard, but the problem is that you can only enter a single character before the program takes off again. If you press the wrong key there is no chance to correct the error before pressing the RETURN key as there is with the INPUT statement. (Using more sophisticated routines, we can GET more than a single character, but we will GET to that later.)

READing In DATA

A third way to enter data into a program is with READ and DATA statements. However, instead of entering the data through the keyboard, DATA in one part of the program is READ in from another part. Each READ statement looks at elements in DATA statements sequentially. The READ statement is associated with a variable which looks at the next

DATA statement and places the numeric value or string in the variable. Let's look at the following example:

```
NEW <RETURN>
10 SCNCLR
20 READ NA$ : REM READS NAME
30 READ OC$ : REM READS OCCUPATION
40 READ SN  : REM READS STREET NUMBER
50 READ ST$ : REM READS STREET NAME
60 READ CT$ : REM READS CITY
70 READ SA$ : REM READS STATE
80 READ ZIP : REM READS ZIP CODE
90 PRINT : PRINT : PRINT
100 REM BEGIN PRINTING OUT WHAT 'READ'
102 REM READ IN. (BE CAREFUL TO PUT IN
104 REN EVERYTHING EXACTLY AS IT IS LISTED.)
110 PRINT NA$
120 PRINT OC$
130 PRINT SN; " " ; ST$
140  PRINT CT$ ; ", " ; SA$ ;" "; ZIP
150 END
1000 DATA SHIRLEY SOFT, PROGRAMMER, 8502,
     DISK DRIVE
1010 DATA SILICONE, SOUTH DAKOTA,49152
```

In the DATA statements there is a comma separating the various elements, unless the DATA statement is at the end of a line. If you have one of the elements out of place or omit a comma, strange things can happen. For example if the READ statement is expecting a numeric variable (such as the street address) and runs into a string (such as the street name) you will get an error message. Think of the DATA statements as a stack of strings and numbers. Each time a READ statement is encountered in the program the first element of the DATA is removed from the stack. The next READ statement looks at the element on top of the stack, moving from left to right. Go ahead and SAVE this program and let's put an error in it. (DSAVE it first, though, so you will have a correct listing of how READ and DATA statements work.)

LIST the program to make sure you have it in memory and enter the following line:

```
85 READ EX$
```

Now RUN the program and you should get an ?OUT OF DATA ERROR IN 85. The error occurred because you have a READ statement without enough DATA statements (or elements); so, be sure that 1) there are enough elements in your DATA statements to take care of your READ statements, and 2) the variables in your READ statements are compatible with the elements of the DATA statements. (i.e. Your numeric variables read numbers and string variables read strings.) To repair your program, simply type in:

```
    1020 DATA WORD
```

This will give it something to READ. (Of course you could have DELETEd line 85). If an element is a DATA statement (and is enclosed in quotation marks), all the characters inside the quotes are considered to be a single string element. For example, make the following changes in your program and RUN it.

```
    145 PRINT EX$
    1020 DATA "10 DOWNING ST, LONDON, 45,
    ENGLAND"
```

Both numbers and commas were happily accepted by a READ statement with a string variable since they were all enclosed in quotation marks. Now remove the quote marks and RUN it again. This time it only printed up to the first comma, '10 DOWNING ST' but the string variable EX$ had no problem accepting a numeric character! (However, since it read the '10' as a string, it cannot be used in a mathematical operation.) Experiment with different elements in the DATA statements to see what happens. Also, just for fun, put the DATA statements at different places in the program. You will quickly find that they can go anywhere and are READ in the order of placement in the program.

Loops

The loop structure in computer programming is a real time-saver. It allows your program to go through a procedure several times while only entering the procedure in the program once. Your Commodore 128 has two loop structures, the FOR/NEXT/STEP loop with a built-in increment/decrement, and the DO/UNTIL(WHILE)/LOOP. The FOR/NEXT loop is used with a known number of passes through a loop, and the DO/UNTIL/LOOP structure is best used when a procedure is to be repeated an unknown number of times.

FOR/NEXT/STEP. The FOR/NEXT loop is one of the most useful operations in BASIC programming. It allows the user to instruct the computer to go through a determined number of steps, at variable increments if desired, and execute them until the total number of steps is completed. Let's look at a simple example to get started.

```
NEW <RETURN>
10 SCNCLR
20 NA$ = "<YOUR NAME>"
30 FOR X = 1 TO 10 : REM BEGINNING OF LOOP
40      PRINT NA$
50 NEXT X : REM LOOP TERMINAL
60 END
```

Now RUN the program and you will see your name printed 10 times along the left side of the screen. That's nice, but so what? OK, not too impressive, but we will see how useful this can be in a bit, but first let's look at another simple illustration to show what's happening to "X" as the loop is being executed.

```
NEW <RETURN>
10 SCNCLR
20 FOR X = 1 TO 10
30    PRINT X
40 NEXT X
```

As we can see when the program is RUN, the value of "X" changes each time the program proceeds through the loop. Think of a loop as a child on a merry-go-round. Each time the merry-go-round completes a revolution, the child gets a gold ring, beginning with one and ending, in our example, with 10.

Having seen how loops function, let's do something practical with a loop. We'll fix up our CHECK-BOOK program we've been playing with. In our souped up CHECK BOOK program, we are going to use variables in many ways. First, our FOR/NEXT loop will use a variable. We'll use "X." Second, we will use a variable to indicate the number of loops to be executed. We will use N%, an integer variable. Third, we will use variables for the balance, the amount of the check, and the new balance. This program is going to be a little longer; so be sure to DSAVE it to disk every 5 lines or so. For cassette, SAVE it about every 10 lines.

```
NEW <RETURN>
10 SCNCLR
20 CB$ = "CHECK BOOK"
30 PRINT : PRINT : PRINT CB$
40 INPUT "HOW MANY CHECKS? ->" ; N%
50 INPUT "WHAT IS YOUR CURRENT BALANCE? ->"
   ;BA
60 REM BEGIN LOOP
70 FOR X = 1 TO N%
80     PRINT "YOUR BALANCE IS NOW $";BA
90     PRINT " AMOUNT OF CHECK #";X; "-> ";
100    INPUT CK : REM VARIABLE FOR CHECK
110    BA = BA - CK : REM RUNNING BALANCE
120   NEXT X : REM TOP OF LOOP
130   SCNCLR: REM CLEAR SCREEN WHEN ALL CHECKS
       ARE ENTERED
140   PRINT : PRINT : PRINT
150   PRINT "YOU NOW HAVE $"; BA ; " IN YOUR
       ACCOUNT"
160   PRINT : PRINT "THANK YOU AND COME AGAIN"
170 END
```

Our check book program is coming along, making it easier to use, and that is the purpose of computers. Now, let's look at something else with loops. NESTED LOOPS. With certain applications, it is going to be necessary have one or more FOR/NEXT loops working inside one another. Let's look at a simple application. Suppose you had two teams with 10 members on each team. You want to make a team roster indicating the team number (#1 or #2) and member number (#1 through #10). Using a nested loop, we can do this in the following program:

```
NEW <RETURN>
10 SCNCLR
20 FOR T = 1 TO 2 : REM T FOR TEAM #
30   FOR M = 1 TO 10 : REM M FOR MEMBER #
40    PRINT "TEAM #" ; T ; "PLAYER #"; M
50   NEXT M
60 NEXT T
70 END
```

In using nested loops, it is important to keep the loops straight. The innermost loop (the "M loop" in our example) must not have any other FOR or NEXT statement inside of it. Think of nested loops as a series of fish eating one another, the largest fish's mouth encompassing the next largest and so forth on down to the smallest fish.

Look at the following structure of nested loops:

```
FOR A = 1 TO N
  FOR B = 1 TO N
    FOR C = 1 TO N
      FOR D = 1 TO N
      NEXT D
    NEXT C
  NEXT B
NEXT A
```

Note how each loop begins (a FOR statement is executed) and is terminated (encounters a NEXT statement) in a "nested" sequence. If you have ever stacked a set of different sized cooking bowls, each one fits inside the other; that is because the outer edge of one is larger than the next one. Likewise, in nested loops, the "edge" of each loop is "larger" than the one inside it and "smaller" than the one it is inside. STEPPING FORWARD AND BACKWARDS. Loops can go one step at a time, as we have been using, or they can step at different increments. For example, the following program "steps" by 10.

```
NEW <RETURN>
10 SCNCLR
20 FOR X = 10 TO 100 STEP 10
30   PRINT X
40 NEXT X
```

This allows you to increment your count by whatever you want. You can even use variables or anything else that has a numeric value. For example,

```
NEW <RETURN>
10 SCNCLR
20 K = 5 : N = 25
30 FOR X = K TO N STEP K
40   PRINT X
50 NEXT
```

Go ahead and RUN the program. But WAIT!!, you say. In line 50 you detect a BUG, a typo and big mistake. After the word NEXT, there should be an "X" but there is none, right? Well, actually, in COMMODORE-128 BASIC you really do not need it, and you can save a little memory if you use NEXT statements without the variable name. Even in nested loops, as long as you put in enough NEXT statements, it is possible to run your program without variable names after NEXT statements. However, it is good programming practice to use variable names after NEXT statements, especially in nested loops so that you can keep everything straight. It is also possible to go backwards. Try this program:

```
NEW <RETURN>
10 FOR X = 4 TO 1 STEP -1
20   PRINT "FINISHING POSITION IN RACE =";X
30 NEXT X
```

As we get into more and more sophisticated (and useful) programs, we will begin to see how all of these different features of COMMODORE-128 BASIC are very useful. Often, you may not see the practicality of a statement initially, but when you need it later on, you will wonder how you could program without it!

=Programming Taboo=

Don't jump out of FOR/NEXT loops. *Sometimes there will be occasions where you want to exit a FOR/NEXT loop before the top of the loop. Usually, no harm will come of it, but with longer and more complex programs, this will definitely cause problems. If a loop must be left before the end of a count, use the DO/WHILE or DO/UNTIL structure instead of a FOR/NEXT.*

DO/WHILE/LOOP and DO/UNTIL/LOOP. The DO/LOOP waits either for a condition to begin or end. Actually, the loop will wait UNTIL a condition is true or WHILE a condition is true. True and false are flagged by a zero (0) if false and a minus one (-1) if true. In other words, as long as a value is not zero, it is true. Let's look at a couple of "adding machine" programs using these structures.

DO/WHILE

```
10 SCNCLR : A=1
20 DO WHILE A
25 REM BEGIN LOOP
30 INPUT "AMOUNT=>"; A
40 T=T+A
50 PRINT "RUNNING TOTAL=";T
60 LOOP : REM TOP OF LOOP
70 PRINT "THE END" : END
```

Notice that we had to first "initialize" the variable 'A' in line 10 with some value greater than zero. The loop begins on line 20 with the condition that WHILE A is greater than 0, keep going through the process from lines 30-50. When a zero is entered when prompted for the AMOUNT, the program exits the loop. In line 20 we could have had it read DO WHILE A <> 0, but this way saves a couple of steps.

DO/UNTIL

To see how the DO/UNTIL structure works, just change line 20 to read:

```
20 DO UNTIL A=0
```

The only difference between the two structures is that while one waits UNTIL a zero is found (false) while the other loops WHILE a zero is not present (true.) Depending on the circumstances, one or the other will be preferable.

=In Case You Wondered=

You may have noticed that the lines inside the loops were indented. If you tried that on your COMMODORE-128 you probably found that as soon as you LISTed your program, all the indentations were gone. Unfortunately, that will happen, and without special utilities, there's nothing you can do about it. However, don't worry about it. It is a programming convention for clarity to indent or "tab" loops to make it easier to understand what the program is doing, but they do not affect your program at all.

Counters. Often you will want to count the number of times a loop is executed and keep a record of it in your program for later use. For example, if you run a program that loops with a STEP of 3, you may not know exactly how many times the loop will execute. To find out, programmers use "counters", variables that are incremented, usually by +1, each time a loop is executed. The following program illustrates the use of a counter:

```
10 SCNCLR
20 FOR X=5 TO 50 STEP 5
30 PRINT "LOOP VALUE=";X
40 N=N+1
50 NEXT X
60 PRINT "YOUR LOOP EXECUTED";N;"TIMES"
```

The first time the loop was entered, the value of "N" was 0, but when the program got to line 40, the value of 1 was added to N to make it 1 (i.e. 0 + 1 = 1). The second time through the loop, the value of N began at 1, then 1 was added, and at the top of the loop, line 50, the value of N was 2. This went on until the program exited the loop. Then, after all the looping was finished, presto!, your N told you how many times the loop was executed. Of course, counters are not restricted to counting loops, and they can be incremented by any value, including other variables, you need. For example, change line 40 to read:

```
40 N = N + (X * 4)
```

RUN your program again and your "counter total" will be a good deal higher.

Using the DO/LOOP structure, counters can keep track of how many times a procedure was used. Since there is a wholly unknown number of times such a loop will be performed, counters are even more useful in the DO/LOOP programs. The following illustrates this:

```
10 SCNCLR
20 DO WHILE A$ <> "END"
30 INPUT "NAME PLEASE";A$
40 C=C+1 : REM COUNTER
50 PRINT : PRINT "NAME#";C;"IS ";A$
60 LOOP
```

Summary

This chapter has begun to show you the power of your computer, and we have really began programming. One of the most important concepts we have covered is that of the "variable". The significant feature of variables is that they "vary" (change depending on what your program does). This is true not only with numeric variables, but also with string variables. The various input commands show how we enter values or strings into variables depending on what we want the computer to compute for us. Finally, we have learned how to loop. This allows us, with a minimal amount of effort, to tell the computer to go through a process several times with a single set of instructions. With loops, we can set the parameters of an operation at any increment we want, and then sit back and let our Commodore 128's go to work for us. However, we have only just begun programming! In the next chapter we will begin getting into more commands and operations that allow us to delve deeper into the Commodore 128's capabilities and make our pro-gramming jobs easier. The more commands we know, the less work it is to write a program.

Branching Out To New Frontiers

Introduction

In this chapter we will begin exploring new programming constructs that will geometrically increase your programming ability. We will be examining some more sophisticated techniques, but by taking each a step at a time, you will begin using them with ease. Later, when you are developing your own programs, be bold and try out new statements. One problem new programmers have is a tendency to stick with the simple statements they have learned to get a job done. After all, why use "complicated" statements to do what simpler ones can do. Well, the answer to that has to do with simplicity. If one "complicated" statement can do the work of 10 "simple" statements, which one is actually simpler? As you get into more and more sophisticated programming applications, your programs can become longer and subject to more bugs. The more statements you have to sift through, the more difficult it is to find the bugs; therefore, while it is perfectly OK to write a long program using a lot of simple statements while you're learning, begin thinking about short-cuts through the use of the more advanced statements.

Related to this issue of maximizing your knowledge of different statements is that of letting the computer perform the computing. This may sound strange at first, but often novices will figure everything out for the computer and use it as a glorified calculator. In the last chapter you may remember we set up a counter to count the times a loop was executed when we used a STEP 3 loop. We could have figured out how many loops were executed instead of letting the computer do it with the counter, but that would have defeated the purpose of programming! So, as you learn new statements, see how they can be used to perform the calculations you had to work out yourself.

Branching

So far all of our programs have gone straight from the top to the bottom with the exception of loops. However, if our COMMODORE-128 is to do some real decision making, we must have some way of giving it options. When a program leaves a straight path, it is referred to as either "looping" or "branching." We already know the purpose of a loop, but what is a branch? Well, using the IF/THEN/ELSE and GOTO statements, we will see. (In fact, with the GET statement in the last chapter, we sneaked these statements in.) Consider the following program: (NOTE: By now you should know enough to clear memory with a NEW statement; so I won't keep on insulting your intelligence by putting them at the beginning of each program.)

```
10 SCNCLR
20 PRINT "CHOOSE ONE OF THE FOLLOWING BY
NUMBER"
30 PRINT
40 PRINT "1. BANANAS"
50 PRINT "2. ORANGES"
60 PRINT "3. PEACHES"
70 PRINT "4. WATERMELONS"
80 PRINT
90 INPUT "WHICH? "; X
100 SCNCLR
110 IF X = 1 THEN GOTO 200
120 IF X = 2 THEN GOTO 300
130 IF X = 3 THEN GOTO 400
140 IF X = 4 THEN GOTO 500
150 GOTO 10
155 REM LINE 150 IS A 'TRAP' TO MAKE SURE
```

```
157 REM THE USER CHOOSES 1, 2, 3, OR 4
200 REM ********
210 REM BRANCHES
220 REM ********
230 PRINT "BANANAS" : END
300 PRINT "ORANGES" : END
400 PRINT "PEACHES" : END
500 PRINT "WATERMELONS" : END
```

As you can see, your computer "branched" to the appropriate place, did what it was told and ENDed. Not very inspiring I admit, but it is a clear example. Now, let's try something a little more practical for your kids to play with in their math homework. While we're at it, we will introduce a new statement, SLEEP. It provides a pause in your program for the number of seconds following the SLEEP statement. Our example uses a two second pause.

```
10 SCNCLR
20 AG$=" ADDITION GAME ": PRINT AG$
30 PRINT : PRINT
40 INPUT "ENTER FIRST NUMBER -->" ; A
50 PRINT
60 INPUT "ENTER SECOND NUMBER-->" ; B
70 PRINT
80 PRINT "WHAT IS "; A ; "+" ; B ; : INPUT C
90 IF C = A + B THEN GOTO 200
100 PRINT  : PRINT "THAT'S NOT QUITE IT. TRY
AGAIN." : PRINT
110 SLEEP 2 : REM TWO SECOND PAUSE
110 GOTO 80
200 REM **************
210 REM CORRECT ANSWER
220 REM **************
230 PRINT " THAT'S RIGHT!  VERY GOOD "
240 PRINT
250 PRINT "WOULD YOU LIKE TO DO MORE?
 (Y/N):";
260  GETKEY AN$
270 IF AN$ = "Y" THEN SCNCLR : GOTO 30
280 SCNCLR : PRINT : PRINT : PRINT
290 PRINT "HOPE TO SEE YOU AGAIN SOON" : END
```

As you can see, the more statements we learn, the more fun we can have. Just for fun, change the program so that it will handle multiplication, division, and subtraction.

=What's In A Name ?=

Kids (of all ages) like to have their names displayed. See if you can change the above program so that it asks the child's name; then when the program responds with either a correction or affirmation command, it mentions the child's name. (e.g. THAT'S RIGHT! VERY GOOD, SAM). Use "NA$" as the name variable.

Let's look carefully at our program to learn something about IF/THEN statements. First, note in line 270, the branch is to clear the screen (SCNCLR) if AN$ ="Y". If any other response is encountered it ends the program. You may ask why the program did not branch to line 30 regardless of the response since the "GOTO 30" statement is after a colon, making it a new line. Good point. The reason for that is after an IF statement, when the response or condition is null, the program immediately drops to the next LINE NUMBER. That is, any statements after a colon in a line beginning with an IF statement will only be executed if the condition of the IF statement is met. Secondly, the condition of AN$ is queried as being a "Y" and not simply a Y without quotes. Since the user enters a Y and not a "Y", we assume that the program will accept a Y, but remember AN$ is a "string" and not a numeric variable. Therefore in the setting of the conditional, we must remember what kind of variable we are using. On the other hand, if we used a numeric variable, such as AN or AN%, we could have entered a line such as,

```
IF AN = 1 THEN....
```

It is also possible to have an alternative branch with ELSE. Using ELSE is an exception to the rule that if the 'true' condition of an IF is not met, the program drops to the next line. Thus, if you want one of two branches, you can place a colon (:) and then ELSE for another branch or statement. For example, look at the following program.

```
10 SCNCLR
20 INPUT "CAN YOU SAY 'YEAH'";Y$
30 IF Y$="YEAH" THEN 100 : ELSE GOTO 200
```

```
40 END
100 REM ***********
110 REM BRANCH THEN
120 REM ***********
130 PRINT "YEAH, YEAH, YEAH"
140 END
200 REM ***********
210 REM BRANCH ELSE
220 REM ***********
230 PRINT "WHAT'D YOU SAY THAT FOR?"
240 END
```

Of course, ELSE does not have to branch to a new line, but can execute a statment on its own. For example,

```
10 SCNCLR
20 PRINT "ENTER 1 OR ELSE!"
30 GETKEY A
40 IF A=1 THEN PRINT "ONE" : ELSE PRINT "NOT
   ONE"
```

Relationals

So far we have only used "=" to determine whether or not our program should branch. However, there are other states, referred to as "relationals", that we can also query. The following is a complete list of the relationals we can employ:

=	Equal to
<	Less than
>	Greater than
<>	Not equal to
>=	Greater than or equal to
<=	Less than or equal to

Now let's play with some of these, and then we'll examine them for their full power. Here are some quickie programs:

```
10 SCNCLR
20 INPUT "NUMBER 1-->";A
30 INPUT "NUMBER 2-->";B
40 IF A > B THEN GOTO 100
50 IF A < B THEN GOTO 200
60 IF A = B THEN GOTO 300
```

```
100 PRINT "NUMBER 1 IS GREATER THAN NUMBER 2
: END
200 PRINT "NUMBER 1 IS LESS THAN NUMBER 2"
: END
300 PRINT "NUMBER 1 IS EQUAL TO NUMBER 2"
```

```
10 SCNCLR
20 PRINT "DO YOU WANT TO CONTINUE? (Y/N)";
30 GETKEY AN$
40 IF AN$ <> "Y" THEN END : ELSE GOTO 20
```

```
10 SCNCLR
20 INPUT "HOW OLD ARE YOU? "; AG%
30 IF AG% >= 21 THEN GOTO 100
40 SCNCLR : PRINT
50 PRINT "SORRY, YOU'VE GOT TO BE 21 OR
OLDER TO COME IN HERE!"
60 END
100 REM **********
110 REM OLD ENOUGH
120 REM **********
130 SCNCLR : PRINT : PRINT "DO YOU COME HERE
OFTEN?"
140 PRINT "I'M A VIRGO.  WHAT'S YOUR SIGN?"
```

Ok, you have the idea how relationals can be used with IF/THEN/ELSE statements; note they work with strings as well as numeric variables. However, there is another way to use relationals. Try the following from the Immediate mode:

```
A = 10 : B = 20 : PRINT A = B
```

Your computer responded with a 0, right? This is a logical operation. If a condition is false, your COMMODORE-128 responds with a 0, but if it is true, it responds with a -1. Now try the following little program.

```
10 SCNCLR
20 A = 10
30 B = 20
40 C = A > B
50 PRINT C
```

When you RUN the program, you again get a 0. This is because the variable C was defined as A being greater than B. Since A was less than B the variable C was 0 or "false." Now, let's take it a step further:

```
10 SCNCLR
20 A = 10
30 B = 20
40 C = A > B
50 IF C = 0 THEN PRINT "A IS LESS THAN B"
: END
60 IF C = -1 THEN PRINT "A IS GREATER THAN B"
```

Later, we will see further applications of these logical operations of the COMMODORE-128. For now, though, it is important to understand that a true condition is represented by a "-1" and a false condition by a "0". AND/OR/NOT Sometimes we need to set up more than a single relational. Suppose, for example, that you are organizing your finances into 3 categories of expenses: (1) Under $10; (2) between $10 and $100 and 3) over $100. With our relationals it would be simple to compare input under $10 and over $100. But what if we wanted to do something in between. In this case we might have some difficulty without added statements. The AND, OR and NOT statements allow us to set ranges with our relationals.

AND	If all conditions are met then true.
OR	If one condition is met then true.
NOT	If condition is not met then true.

For example:

```
10 SCNCLR
20 INPUT "ENTER AMOUNT -->$"; A
30 IF A < 10 THEN 100
40 IF A > 10 AND A <= 100 THEN 200
50 IF A > 100 THEN 300
100 REM ****************
110 REM LESS THAN BRANCH
120 REM ****************
130 PRINT " PETTY CASH " : GOTO 400
200 REM *****************
210 REM IN BETWEEN BRANCH
```

```
220 REM ****************
230 PRINT " GENERAL EXPENSES " : GOTO 400
300 REM ******************
310 REM GREATER THAN BRANCH
320 REM ******************
330  PRINT " BIG BUCKS "
400 REM ****************
410 REM WHAT NEXT? BRANCH
420 REM ****************
430 PRINT " DO YOU WISH TO CONTINUE? ";
440 GETKEY AN$
450 IF AN$ < > "Y" AND AN$ < > "N" THEN PRINT
"ANSWER 'Y' OR 'N' PLEASE " : GOTO 400
460 IF AN$ = "Y" THEN 10
470 SCNCLR : PRINT "GOODBYE"
```

In line 40 we set the conditional branch to be BOTH greater than 10 and equal to or less than 100. The variable "A" had to meet both conditions to branch. Similarly, in line 420, using the AND statement again, we made sure that the response had to be either "Y" or "N". If you are very perceptive, you may have asked yourself about some fishy format in the program. There are conditional IF/THEN lines that simply say THEN 100 and stuff like that. What's going on? Shouldn't there be a GOTO statement there? Again, we have slipped in another feature of COMMODORE-128 BASIC. When using IF/THEN statements, it is possible to drop the GOTO on a branch and simply put in the line number. However, note that we have used GOTO statements elsewhere in the program where no conditional is used within the same line or within a single set of colons. Until you become more familiar with programming you might want to keep your GOTO statements after IF/THEN statements, but they are not required. Another question you may have had involves the AND statement in line 420. In normal English if we say something is not "Y" or "N" sometimes we mean that it must be one or the other, exclusively. However, in programming, if we use OR, we are telling the program to branch if either condition is met. Thus, if we wrote line 420 as,

```
420 IF AN$ < > "Y" OR AN$ < > "N" THEN
PRINT "ANSWER 'Y' OR 'N' PLEASE " : GOTO 400
```

the program would have branched if AN$ was not equal to EITHER "Y" or "N". Thus, for example, if we responded with a "Y", that "Y" would have NOT been equal to "N" and so the program would have branched to "ANSWER 'Y' OR 'N' PLEASE" - not what we intended. To check this, change the AND to an OR in line 420 and RUN the program. Now, let's use the OR and NOT statements in a program:

```
10 SCNCLR
20 READ A
30 READ B
40 READ C
50 DATA 10,20,30
60 IF A+B = C OR A<B OR A-B = C THEN 100
70 END
100 REM **************
110 REM IF ONE IS TRUE
120 REM **************
130 SCNCLR : PRINT "ONE OF 'EM MUST BE TRUE"
```

Looking at line 60 we can see that A - B does not equal C; however, A + B does equal C and A is less than B. Using the OR statement, only one statement has to be true to branch. Now, let's try the following program:

```
10 SCNCLR
20 READ A : READ B : READ C
30 DATA 10,20,30
40 Z = A - B = C
50 IF NOT Z THEN 100
60 END
100 REM **********
110 REM NOT BRANCH
120 REM **********
130 PRINT "THAT'S RIGHT! A-B=C ISN'T RIGHT!"
```

As can be seen from the example, it is possible to use the "negation" of a formula to calculate a branch condition. In most cases, you will use < > (not equal) or the positive case, but at other times it will be simpler to employ NOT.

Subroutines

Often in programming there is some operation you will want your computer to perform at several different places in the program. Now, you can repeat the instructions again and again or use GOTO's all over the place to return to your original spot after branching to the operation. On the other hand, you can set up "subroutines" and jump to them using GOSUB and get back to your starting point using the RETURN statement. Up to a point the GOSUB statement works pretty much like the GOTO statement since it sends your program bouncing off to a line out of sequence. Also, the RETURN statement is something like GOTO since it also sends your program to an out-of-sequence line. However, the GOSUB/RETURN pair is unique in what it does. Let's take a look at a simple example to see how it works:

```
10 SCNCLR
20 A$ = "HELLO" : GOSUB 100
30 A$ = "HOW ARE YOU TODAY?" : GOSUB 100
40 A$ = "I'M FINE" : GOSUB 100
 50 END
100 REM ****************
110 REM PRINT SUBROUTINE
120 REM ****************
130 PRINT A$
140 RETURN
```

Our example shows that a GOSUB statement works exactly like a statement on the line itself except that it is executed elsewhere in the program. The RETURN statement brings it back to the next statement after the GOSUB statement. Using the GOSUB/RETURN pair it is much easier to weave in an out of a program than using GOTO since the RETURN automatically takes you back to the jump-off point. To better illustrate the usefulness of GOSUB, let's change line 100 to something more elaborate. Try the following. (Note: We will be getting ahead of ourselves a bit with this example, but the following is meant to illustrate something very useful in GOSUB's.)

```
130 L = LEN (A$)/2 : PRINT TAB(20 - L) ; A$
```

Now when you RUN the program, all of your strings are centered. As you can see, a single routine handled all of the centering, and instead of having to rewrite the routine every time you want a string centered all you had to do was to use a GOSUB to line 100.

=Neatness Counts=

We really have not discussed the structure of programs too much up to this point. In part, this is because we have not really had the need to do so. However, as our instruction set grows, so too does the possibility for errors, and by now if you haven't made an error, you haven't been keying in these programs! One way to minimize errors, especially using GOSUB's, is to organize them into coherent "blocks." Basically, a "block" is a subroutine within a range of lines. For example, you might block your subroutines by 100's or 1000's, depending on how long the subroutines are. Thus, you might have subroutines beginning at lines 500, 600 and 700. It doesn't matter if the subroutine is 1 line or 10 lines, as long as it is confined to the block, it is easier to debug, easier for others and you to understand what is happening in the program, and in general a good programming practice. We've used the stars (asterisks) and REM statements to highlight our blocks.

Computed GOTO and GOSUB

Now we're going to get a little fancier, but in the long run, it will result in clearer and simpler programming. As we have seen, we can GOTO or GOSUB on a "conditional" (e.g. IF A = 1 THEN GOTO 200). The easier way to make a conditional jump is to use "computed" branches using the ON statement. For example,

```
10 SCNCLR
20 INPUT "ENTER A NUMBER FROM 1 TO 5 " ; A
30 IF A < 1 OR A > 5 THEN 20 : REM TRAP
40 ON A GOSUB 100,200,300,400,500 : REM
COMPUTED GOSUB
50 PRINT "DO YOU WISH TO CONTINUE? (Y/N)" ;
60 GETKEY AN$
70 IF AN$ < > "Y" THEN END
80 GOTO 10
100 REM ***************
110 REM SUBROUTINE CITY
120 REM ***************
130 PRINT "ONE" : PRINT : RETURN
```

```
200 PRINT "TWO" : PRINT : RETURN
300 PRINT "THREE" : PRINT : RETURN
400 PRINT "FOUR"  : PRINT : RETURN
500 PRINT "FIVE"  : PRINT : RETURN
```

The format for a computed GOSUB/GOTO is to enter a variable following the ON command. The program will then jump the number of "commas" to the appropriate line number. If a 1 is entered, it takes the first line number, a 2, the second, and so forth. It's a lot easier than entering,

```
70 IF A = 1 THEN GOSUB 100
80 IF A = 2 THEN GOSUB 200
etc.
```

However, it is necessary to use relatively small numbers in the "ON" variable since there is a limited number of subroutines. If your program is computing larger numbers, all you have to do is to convert the larger numbers into smaller ones by changing the variables. For example:

```
10 SCNCLR
20 INPUT "ENTER ANY NUMBER--> "; A
30 IF A < 100 THEN B = 1
40 IF A >= 100 AND A < 200 THEN B = 2
50 IF A >= 200 THEN B = 3
60 ON B GOSUB 100, 200, 300
65 REM LINE 60 COMPUTED GOSUB ON 'B'VARIABLE
70 PRINT "DO YOU WISH TO CONTINUE?(Y/N)";
80 GET AN$:IF AN$<>"Y" THEN END:ELSE GOTO 10
100 REM ***********
110 REM SUBROUTINES
120 REM ***********
130 PRINT "LESS THAN 100" : RETURN
200 PRINT "MORE THAN 100 BUT LESS THAN 200 "
: RETURN
300 PRINT "MORE THAN 200" : RETURN
```

RUN the program and enter any number you want. Since the program is branching on the variable B, and not on A (the INPUT variable), you will not get an error since the greatest value of B can only be 3. Now let's get back to relationals and see how they can be used with computed GOSUBS. Remember, in using relationals, the only numbers we get are 0's and 1's for false and true respectively. However, we can use these 0's and 1's just like regular numbers. Try the following:

```
10 SCNCLR
20 X = 1 : Y = 2 : Z = 3
30 A = X < Z
40 B = Y > Z
50 C = Z > X
60 PRINT "A + A =" ; A + A
70 PRINT : PRINT "A + B =" ; A + B
80 PRINT : PRINT "A + B + C =" ; A + B + C
90 END
```

Now before you RUN the program, see if you can determine what will be printed by lines 60, 70 and 80. Once you have made a determination, RUN the program and see what happens. Go ahead and do it. How'd you do? Let's go over it step by step.

STEP 1. Since X is less than Z, A will be "true" with a value of one (-1). Therefore A + A (-1 + -1) will equal -2.

STEP2. Since Y is not less than Z , (Y = 2 and Z = 3, remember) B will be "false" with a value of 0. Therefore, A + B (-1 + 0) will total -1.

STEP3. Since Z is greater than X, C will be "true" with a value of -1. Therefore A + B + C (-1 + 0 + -1) will equal -2.

If you got it right, congratulations! If not, go over it again. Remember, very simple things are happening, and so don't look for a complicated explanation! Now that we see how we can get numbers by manipulating relationals, let's use them in computed GOSUB's. The following program shows how:

```
10 SCNCLR
20 INPUT "HOW BIG WAS THE CROWD?"; HC
30 R = 1 + (HC >= 500) + (HC >= 1000)
40 IF R = 0 THEN R = 2
50 IF R = -1 THEN R = 3
60 ON R GOSUB 100,200,300
70 PRINT : INPUT "DO YOU WISH TO CONTINUE?
(Y/N) "; AN$
80 IF AN$ < > "Y" THEN END : ELSE GOTO 10
100 REM **********
110 REM SUBROUTINES
```

```
120 REM ***********
130 SCNCLR:PRINT "THE CROWD WAS NOT VERY BIG
- LESS THAN 500" : RETURN
200 SCNCLR : PRINT "THE CROWD WAS A PRETTY
GOOD SIZE - BETWEEN 500 AND 1000.":RETURN
300 SCNCLR : PRINT "THE CROWD WAS VERY BIG
- 1000 OR OVER! " : RETURN
```

This program is hinged on line 30's formula or algorithm. Let's see how it works:

1. There are 3 conditions:
- **a.** HC is less than 500
- **b.** HC is 500 or more but less than 1000
- **c.** HC is 1000 or greater.

2. If the first condition exists both HC >= 500 and HC >= 1000 would be false. Thus 1 + 0 + 0 = 1. Therefore R = 1.

3. If HC is >= 500 but less than 1000 then HC >= 500 would be true but HC >= 1000 would be false. Thus we would have 1 + (-1) + 0 = 0 . Convert the value of R to 2.

4. Finally if HC is both >= 500 and >=1000 then our formula would result in 1 + (-1) + (-1) = -1. Convert the value of R to 3.

==GETTING IT RIGHT==

At this point let's take a little rest and reflection. In programming, there is no such thing as THE RIGHT WAY and THE WRONG WAY. Certain programs are more efficient, faster or take less code and memory than others, but the computer makes no moral judgments. If a program does what you want it to do, no matter how slowly it does it or how long it took you to write it, it is "right." In the above example we used an algorithm with relationals to do something we could have done with more code. Don't expect to use such formulas right off the bat unless you have a strong background in math. If you're not used to using algorithms, don't expect to understand their full potential right away. The one we used is relatively simple, and you will find far more elaborate ones as you begin looking at more programs. The main point is to keep plugging ahead. With practice, you will learn all kinds of little shortcuts and formulas, but if you get *stuck along the way, just keep on going. Remember, as long as you can get your program running the way you want it to, you're doing the "right" thing.*

Strings and Relationals

Before we leave our discussion of computed GOTO's and GOSUB's with relationals, let's take a look at how relationals handle strings. Try the following :

```
A$ = "A" : B$ = "B" : PRINT B$ > A$
<RETURN>
```

Surprised? In addition to comparing numeric variables, relationals can compare alphabetic string variables with "A" being the lowest and "Z" the highest. (Actually, any string variables can be compared, but we will just look at the alphabetic ones here.) So if we ask is B$ greater than A$, we get a "-1" (true) since B$ was a B and A$ was an A. Now you might be wondering what on earth you could possibly want to do with this knowledge. Well, in sorting strings (like putting names in alphabetical order) such an operation is crucial. Later on we will show you a routine for sorting strings, but for now let's make a simple string sorter for sorting two strings.

```
10 SCNCLR
20 INPUT "WORD #1 --> " ; A$
30 INPUT "WORD #2 --> " ; B$
40 PRINT : PRINT : PRINT
50 IF A$ < B$ THEN PRINT A$ : PRINT B$
60 IF A$ > B$ THEN PRINT B$ : PRINT A$
```

Just what you needed! A program that will put two words in alphabetical order! ARRAYS The best way to think about arrays is as a kind of variable. As we have seen, we can name variables A, D$, KK%, X1 and so forth. An array uses a single name with a number to differentiate different variables. Consider the following two lists, one using regular string variables and the other using a string array:

STRING VARIABLE	**STRING ARRAY**
P$ = "PIG"	AM$(1) = "PIG"
C$ = "CHICKEN"	AM$(2) = "CHICKEN"
D$ = "DOG"	AM$(3) = "DOG"

Now if we PRINT H$ we'd get HORSE and if we PRINT AM$(4) we'd also get HORSE. Likewise, we could use arrays for numeric variables such as:

```
A(1)  = 1
A(2)  = 2
A(3)  = 3
A(4)  = 4 etc.
```

Again, you may well ask, "So what? Why not just use regular numeric or string variables instead of arrays?" Well, for one thing, it can be a lot easier to keep track of what you're doing in a program using arrays, and for another, it can save a lot of time. Consider the following program for INPUTing a list of 10 names using a string array.

```
10 SCNCLR
20 FOR X = 1 TO 10
30 PRINT "NAME #"; X ; : INPUT NA$(X)
40 NEXT X
50 FOR X = 1 TO 10 : PRINT NA$(X)
60 NEXT X
```

Now, write a program that does the same thing using non-array variables. It would take a lot more code to do so, but go ahead and try it. Use the variables N0$ through N9$ for the names just to see what it would take. If you re-wrote the program, you saw how much time using arrays saved, but before going on let's take a closer look at how the program worked with the FOR/NEXT loop and array variable:

1. The FOR/NEXT loop generated the numbers sequentially so that the array would be the following:

```
FOR X = 1 TO 10

   NA$(1)   <--First time through loop
   NA$(2)  <--Second time through loop
   NA$(3)  <--Third time through loop
   NA$(4)  etc.
   NA$(5)
   NA$(6)
   NA$(7)
   NA$(8)
   NA$(9)
   NA$(10)
 NEXT X
```

2. Each string INPUT by the user was stored in a sequentially numbered array variable.

3. Output, using the PRINT statement, was generated by the FOR/NEXT loop sequentially supplying numbers to be entered into array variables. Now to get used to the idea that an array variable is a variable, enter the following:

```
A(10) = 432 : PRINT A(10) <RETURN>
XYZ(9) = 2.432 : PRINT XYZ(9) <RETURN>
R2D2$(1) = "BEEP!" + CHR$(7) : PRINT R2D2$(1)
<RETURN>
J%(5) = 321 : PRINT J%(5) <RETURN>
```

OK, maybe it didn't take all that to convince you that an array is a variable with a number in parentheses after it, but it's easy to forget and think of arrays as something more exotic than they are.

THE DIMension of an ARRAY

If you've been very observant, you may have noticed we haven't gone over the number 10 in our array examples. The reason behind that is because once our array is larger than 10 we have to use the DIM (dimension) statement to reserve space for our array. (Actually 11 array elements are automatically dimensioned - 0 to 10.) The following is an example of the format for DIMensioning an array.

```
10 SCNCLR
20 DIM AB(150) : REM DIMENSION OF ARRAY
VARIABLE 'AB'
30 FOR X = 1 TO 150
40   AB(X) = X
50 NEXT X
60 FOR X = 1 TO 150
70   PRINT AB(X),
80 NEXT X
```

RUN the program as it is written. It should work fine. Now delete line 20 by simply entering 20. (Remember how we learned to delete single line numbers by entering that number?) Now RUN the program, and you will get an error for not DIMing the ARRAY. (?BAD SUBSCRIPT ERROR IN 40 - that's because there was no DIM statement in Line 20). So, whenever your arrays are going to have more than 11 values from 0 to 10, be sure to DIM them.

=Better Safe Than Sorry Dept.=

Many programmers always DIM arrays, regardless of the number in the array. It is perfectly all right to do so, and statements such as DIM X$(3) or DIM N% (5) are valid. Often when copying programs from books or magazines you may run across these lower level DIM statements because the programmer thinks it's a good idea to DIM all arrays as part of programming style and clarity. Furthermore, you can save memory space by using the minimal amount of DIMension space, and if the program is large enough, it may be necessary to DIM and array at less than 11. Finally, some versions of BASIC require all arrays to be DIMensioned.

Multi-DIMensional Arrays

So far, all we have examined are single dimension arrays. However, it is possible to have arrays with two or more dimensions. Let's begin with two-dimensional arrays, and examine how to use arrays with more than a single dimension. The best way to think of a 2-dimensional array is as a matrix. For example if our array ranged from 1 to 3 on two dimensions the entire set would include: A(1,1) A(1,2) A(1,3) A(2,1) A(2,2) A(2,3) A(3,1) A(3,2) and A(3,3). By laying it out on a matrix, we can think of the first number as a row and the second as a column. This makes it much clearer:

	COLUMN #1	COLUMN #2	COLUMN #3
ROW #1	A(1,1)	A(1,2)	A(1,3)
ROW #2	A(2,1)	A(2,2)	A(2,3)
ROW #3	A(3,1)	A(3,2)	A(3,3)

Again, it is important to remember that each element in the array is simply a type of variable. To drum that into your head do the following:

```
XV$(3,1)  =  "I'M  LIKE  A  VARIABLE"  :  PRINT
           XV$(3,1) <RETURN>
JK%(2,2) = 21 : PRINT JK% <RETURN>
MM (1,1) = 3.212 : PRINT MM(1,1) <RETURN>
```

OK, so you were reminded a bit much, but in order to use arrays to their fullest advantage in programs, they must be envisioned as an orderly set of variables and not something else. Now, let's use a 2-dimension array in a program. Our program will be to line up people in a 12 member marching band.

```
10 SCNCLR
20 DIM BA$(4,4) : REM MAKE 4 'ROWS' AND 4
'COLUMNS'
30 FOR I = 1 TO 4 : REM ROWS
40   FOR J = 1 TO 4 : REM COLUMNS
50     READ BA$(I,J)
60   NEXT J
70 NEXT I
80 DATA MARY, TOM, SUE, PETE, JACK, NANCY,
BETTY, BILL
90 DATA RALPH, PAT, DARLENE, FRANK, HORACE,
DAVID, KARL, ERIC
100 REM ************
110 REM OUTPUT BLOCK
120 REM ************
130 FOR I = 1 TO 4 : REM ROWS
140   FOR J = 1 TO 4 : REM COLUMNS
150     PRINT BA$(I,J) , : REM COMMA WILL
 FORMAT OUTPUT 4 ACROSS
160   NEXT J
170 NEXT I
```

When you RUN this program, all of your band members will be lined up. However, you could have done the same thing with a single dimension array since all that "lines them up" is the use of the comma to format the PRINT statement in line 150. So, what's the big deal about a 2-dimension array? Well, to see, let's add some lines to our program:

```
180 PRINT :PRINT "HIT ANY KEY TO CONTINUE ";
190 GETKEY AN$
200  SCNCLR : PRINT "WHAT ROW & COLUMN WOULD
YOU LIKE TO SEE? "
```

```
210 INPUT "ROW #-> ";R
220 INPUT "COL #-> ";C
230 PRINT : PRINT BA$(R,C); " IS IN ROW "; R;
" COLUMN "; C
240 PRINT  : PRINT "MORE?(Y/N) ";
250 GETKEY M$
260 IF M$ = "Y" THEN 200
```

Now you can locate the value or contents on a specific array on two dimensions. In our example, if you know the row number and column number, you can find the band member in that position. The use of 2-dimensional arrays in problems dealing with matrixes is an important addition to your programming commands. It is also possible to have several more dimensions in an array variable. As you add more and more dimensions, you have to be careful not to confuse the different aspects of a single array. Sometimes, when a multi-dimensional array becomes difficult to manage (or use), it is better to break it down into several 1- or 2-dimensional arrays. But just for fun, let's see what we might want to do with a 3 dimensional array with the following program : (By the way, this problem is based on an actual application!)

```
10 SCNCLR
20 PRINT "WINECELLAR ORGANIZER "
30 PRINT : PRINT "HOW MANY RACKS,ROWS,
COLUMNS?"
40 INPUT "(ENTER EACH SEPARATED BY A
COMMA)";RK,R,C
50  DIM WI$(RK,R,C)
60  INPUT "HOW MAY BOTTLES TO STORE? ";N%
70  PRINT : FOR I = 1 TO N%
80  INPUT "RACK #-> ";RA
90  INPUT "ROW  #-> ";RO
100  INPUT "COL  #-> ";CO
110 INPUT "NAME OF WINE : ";WN$
120 WI$(RA,RO,CO) = WN$
130  NEXT I
200 REM ********************************
210 REM ROUTINE FOR CHECKING CONTENTS OF
WINE CELLAR
220 REM ********************************
230 SCNCLR : INPUT "WHICH RACK # WOULD YOU
LIKE TO CHECK? ";RR
240  FOR I = 1 TO R
```

```
250  FOR J = 1 TO C
260 IFWI$(RR,I,J)=""THEN WI$(RR,I,J)="EMPTY"
270 PRINT "RACK #";RR;" ROW #";I;" COLUMN
#";J;" CONTAINS ";WI$(RR,I,J)
280    NEXT J
290  NEXT I
300  END
```

Now that was a pretty long program, but go over it carefully to make sure you understand what it is doing. Again, let me remind you that all the 3-dimensional array is a variable with a lot of numbers in parentheses. Also, note on line 40 how we INPUT several values with a single INPUT statement. We used the format,

```
INPUT A, B, C
```

and as long as the operator (program user) is told to enter the appropriate number of responses and separate each with a comma, every thing will work fine. Also, it would be a good idea to save this program on a disk as an example of a multi-dimensional array.

Summary

We covered a good deal in this chapter, and if you understood everything, excellent! If you did not, don't worry, for with practice, it will all become very clear. Whatever your understanding of the material, though, experiment with all the statements. Be **BOLD** and daring with your computer's commands, and as long as you have a disk or cassette on which you can practice your skills, the worst that can happen is that you will erase a few programs! We learned that your COMMODORE-128 computer can compute! Using the IF/THEN/ELSE statements and relationals we can give the computer the power of "decision making." Using subroutines it is possible to branch at decision points to anywhere we want in our program. Computed GOTOs and GOSUBs allow the execution to move appropriately with a minimal amount of programming. Finally, we examined array variables. Arrays allow us to enter values into sequentially arranged variables (or elements). Using FOR/NEXT and DO/WHILE/UNTIL loops it is possible to quickly program multiple variables up to the limits of our DIMensions. Not only do arrays assist us in keeping variables orderly, they save a good deal of work as well. In the next chapter, we will begin working with commands that help arrange everything for us. As our programs become more and more sophisticated, we will need to keep better track of what we're doing. By organizing our programs into small, manageable chunks, we can create clear useful programs.

Program Organization

Introduction

Unless we organize, as we accumulate more and more information, work, or just about anything else, things get confusing. Good organization allows us to do more and to handle more complex and larger problems. These principles hold with programming. As we learn more statements, we can do more things, but the more we do, the more likely we are to get tangled up and lost. One of the areas that is likely to be the first to suffer from "overflow" is that of formatting output. Variables get mixed up, arrays are misnumbered and the screen is a mess. In order to handle this kind of problem, we will deal extensively with text and string formatting. Not only will we be able to put things where we want them, but we will do it with style! The second major area of disorganization is I/O (INPUT/OUTPUT). Some of the problem has to do with formatting, but even more elementary is the problem of organizing the input and output so that data is properly analyzed. Data has to be connected to the proper variables and be subject to the correct computations. Thus, in addition to examining string formatting, we will also look at organizing data manipulation.

Formatting Text

In Chapter 1 we said that the COMMODORE-128 keyboard works in many ways like a typewriter. One feature of a typewriter is its ability to set "tabs" so that the user can automatically place text a given number of spaces from the left margin. With your COMMODORE-128, you can TAB and SPC. Additonally, there are many statements you make with your keys. For example, if you enter PRINT {CLR/HOME} [press the CLR/HOME key and SHIFT key- *a little heart appears*], it works just like SCNCLR. Also, you can use the PRINT statement with the arrow keys and have the cursor moved around the screen. All statements using a keypress with a PRINT statement are in brackets {}, but the screen shows an inverse character. Statements using single keys must be placed in quote marks in the same way as are strings. Let's look at what each of these statements means:

Statement	**Meaning**
SCNCLR	Clears screen and homes cursor
TAB (N)	Used within PRINT statement to place next character N spaces from left margin
SPC (N)	Used within PRINT statement, creates specified number of spaces. (SPC starts printing non-space 1 space after N).
{CLR/HOME}	Works just like SCNCLR. Press SHIFT key and CLR/HOME keys. *Inverse heart on screen.*
{HOME}	Places cursor in upper left hand corner of screen. Use the CLR/HOME key without pressing SHIFT key. *Inverse 'S' on screen.*
{ARROW}	Moves cursor one space in direction arrow points. ↑ *Inverse ball* ← *Inverse vertical line* ↓ *Inverse 'Q'* → *Inverse right bracket*

Now, to better see how these statements format text output, let's USE THEM!

```
10 SCNCLR : PRINT : PRINT
20 PRINT TAB (20);"TAB TO HERE"
30 PRINT SPC(20);"SPC TO HERE"
40 PRINT "{HOME}";"UP HERE!" :REM PRESS
   THE CLR/HOME KEY WITHOUT THE SHIFT KEY
   - YOU'LL GET AN INVERSE "S"
50 FOR I = 1 TO 20 : PRINT : NEXT : PRINT
   "DOWN HERE"
```

When you RUN this program, note that when you used the {HOME} key/statement, it did not clear the screen. Rather, it placed the cursor at the top of the screen, leaving what was printed in lines 20 and 30 on the screen. Also, we were able to produce a vertical tab by using empty PRINT statements in line 50 to take the text down to vertical position 20 on the screen. We could have used the PRINT "{DOWN ARROW}" combination, but that would have taken extra key strokes. Again, the other text on the screen was not erased.

Now let's have a little fun with our statements. Here's a little program that will give you an idea of how to place text within your program.

```
10 SCNCLR : FOR I = 1 TO 4 : PRINT : NEXT
20 INPUT "ENTER MESSAGE--> "; MS$
30 PRINT : INPUT"HORIZONTAL PLACEMENT (1-40)>
    "; H
40 PRINT : INPUT "VERTICAL PLACEMENT (1-25)>
    "; V
50 SCNCLR
60 FOR VER = 1 TO V : PRINT :NEXT VER : PRINT
    TAB(H); MS$
70 PRINT : PRINT "HIT ANY KEY TO CONTINUE OR
   'Q' TO QUIT ";
80 GETKEY A$
90 IF A$ < > "Q" THEN 10 : ELSE END
```

As you can see, variables can be used with formatting statements. Thus, TAB (H), is read in the same way as TAB(10) or TAB(15) or any other number between 1 and 40. Using the above program, what do you think would happen if you entered "THIS IS A LONG STRING", a HORIZONTAL placement of 39 and a VERTICAL placement of 25? Since the maximum TAB is 40 and the maximum vertical placement is is 25, the string (MS$) will go over the boundaries. Go ahead and try it to see what happens. In fact, it would be a good idea to test the limits of TAB and vertical placement with this program to get a clear understanding of their parameters.

To see how to use the cursor control keys from within a program, we'll start off with a simple demonstration that will *dramatically* show you the importance of formatting statements using the arrow keys. In the following program, we will first RUN it with a semi-colon after the PRINT "{RIGHT ARROW}", and then RUN it a second time without the semi-colon.

```
10 SCNCLR
20 FOR X=1 TO 10
30 PRINT "{RIGHT ARROW}";
40 NEXT X
50 PRINT "X"
```

When you RUN the program, the 'X' is at the top of the screen in Column 11. Remove the semi-colon from line 30 and RUN it again. On the second time, the 'X' is in Row 11. What happened is that when the program was first RUN, the cursor was moved to the right 10 times with NO CARRIAGE RETURN. However, when the semi-colon was removed, each time the program went through the loop, it moved the cursor one space to the right AND then issued a CARRIAGE RETURN. If we substitute a character for the right arrow key, the we can better see this. Go ahead and place the character 'V' where the right arrow symbol [*inverse bracket*] is and RUN the program with and without the semi-colon. Where the letter 'V' is on your screen is where the cursor would move using the right arrow key within a PRINT statement.

Using PRINT USING and PUDEF

One of the most useful formatting tools in COMMODORE 128 BASIC is PRINT USING. Why they came up with this awkward name is beyond me, but I'm sure glad they have it for formatting text. The BASIC 7.0 version of PRINT USING is a little different other versions you may have used on other computers, but it is similar enough so that you should have no trouble in adjusting to it.

By now you have probably noticed that there are gaps between numbers and strings using the PRINT statement. Also, you may have noticed that when you have numbers with trailing zeroes after decimal points, the zeroes are dropped. For example, enter:

```
    PRINT 20.30 <Return>
```

Your results were

```
    20.3
```

For the most part that's fine, but with our CHECKBOOK program, it looks sloppy to have a balance of $ 456.7. The dollar sign is off and there's no zero after the 7. What's a programmer to do? For the first problem of having gaps between your dollar sign and amount, we can use the format,

```
    PRINT USING "#$###";123
```

The pound (#) signs refer to the number of digits to be printed out. Try the following program to see what happens when there are more digits than spaces for them:

```
10 SCNCLR
20 FOR X = 1 TO 150 STEP 50
30 PRINT USING "$#";X
40 NEXT X
```

The results show:

```
$1
$*
$*
```

Now, using your EDITOR, change line 30 to,
30 PRINT USING "$###";X
This time you got,

```
$  1
$ 51
$101
```

If you add more pound (#) signs, you will just get more spaces to the left of the dollar sign. Change line 30 by adding about five more pound (#) signs to see what happens.

Now let's solve the pesky problem of the dropped zeros. All you have to do is to include a decimal point (.) among the the pound signs (#) where you want your decimal points. Watch this when you RUN it:

```
10 SCNCLR
20 FOR X = 10 TO 250 STEP 20
30   PRINT USING "##.##"; X/5
40 NEXT X
```

You got all your trailing zeroes, and everything was lined up nicely.

Next, let's combine our knowledge and use both dollar signs and trailing zeroes. Also, in order to get the dollar sign adjacent to the left-most number, we will place a pound sign (#) in front of the dollar sign.

```
PRINT USING "#$##.##"; N
```

and try the following program: (**Note:** We have added some key formatting. See what it does on your screen. Use the non-shifted CLR/HOME for {HOME}.)

```
10 SCNCLR : PRINT
20 INPUT "HOW MUCH $ WOULD YOU LIKE ";N
30 PRINT "{HOME}" :REM INVERSE 'S' ON SCREEN
40 FOR X=1 TO 5 :PRINT"{DOWN ARROW}";:NEXT X
50 PRINT USING "#$#########.##"; N
60 PRINT "{HOME}"
70 IF N=0 THEN END : ELSE GOTO 20
```

When asked how much you would like, do not put any cents in to see what happens. (What's a few cents when dealing with millions?) Line 70 expects a 0 (zero) to end the program. However, if you enter a zero, and there are still values at the INPUT position on the screen, you will not exit the program. You have to enter a zero and then space over the unwanted numbers to exit. To fix that, we will use the cursor keys to move to the end of the INPUT and blank over it with 10 spaces between quote marks. Insert the following three lines:

```
62 FOR X=1 TO 27 :PRINT"{RIGHT ARROW}";
:NEXT X
64 PRINT "          " ; : REM 10 SPACES
BETWEEN QUOTES
```

```
    66 PRINT "{HOME}"
```

Your format is nice and clean when you RUN the program.

Now that you have an idea how to use PRINT USING with dollars and cents, let's take a look at what else you can do with this statement. Enter the examples to get used to each format.

PRINT USING CHART
(Suitable for Framing)
=====================

# One digit position for each pound sign (#).
EXAMPLE:

```
    PRINT USING "###";432
```

##.## Places decimal point in position relative to pound signs and number of pound signs.
EXAMPLE:

```
    PRINT USING "##.###"; 34.56
```

$ Places left justified dollar sign before right justified number.
EXAMPLE:

```
    PRINT USING "$###.##"; 23.45
```

#$ Places dollar sign adjacent to left-most number with right justified output of numbers.
EXAMPLE:

```
    PRINT USING "#$###.##";1.43
```

↑↑↑↑ Placed at the end of a PRINT USING format, it results in an exponential output.
EXAMPLE:

```
    PRINT USING "####↑↑↑↑";8502
```

, (comma) Places comma at indicated position in numeric output.
EXAMPLE:

```
    PRINT USING "#,###,###"; 1234567
```

+ OR - Places plus or minus in front or back of number.
EXAMPLE:

```
    PRINT USING "+####";5599
```

```
PRINT USING "#$####.##-";-898.88
```

= Centers strings based on the number of spaces indicated by pound signs (#).
EXAMPLE:

```
    10 SCNCLR
    20 FOR X= 1 TO 9
    30 INPUT "WHAT'S YOUR STRING";S$(X)
    40 NEXT X
    100 REM ***************
    110 REM CENTERED OUTPUT
    120 REM ***************
    130 FOR X=1 TO 9
    140 PRINT USING "##########=";S$(X)
    150 NEXT X
```

> Right justifies string in field.
EXAMPLE:

```
    10 SCNCLR
    20 FOR X= 1 TO 9
    30 INPUT "WHAT'S YOUR STRING";S$(X)
    40 NEXT X
    100 REM **********************
    110 REM RIGHT JUSTIFIED OUTPUT
    120 REM **********************
    130 FOR X=1 TO 9
    140 PRINT USING "##########>";S$(X)
    150 NEXT X
```

Other keys. Any other key placed in an end position will attach itself to the end of a number.
EXAMPLE:

```
    PRINT USING "##%";45
```

PUDEF - Roll Your Own Format

The default symbols in PRINT USING are set in four positions. They are:

<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>
Space	,	.	$

To redefine these symbols, use the PUDEF (for Print Using DEFinition) command. BE CAREFUL with it, though. Even your faithful *SYSTEM GUIDE* goofed up the example on page 79. When redefining the characters, you must redefine every single space or it will result in a blank. For example, let's say you wanted British pounds (£) instead of dollar signs ($), but you still wanted the commas and periods left the same. You would do the following:

```
PUDEF " ,.£"
```

That would swap the British pound sign for the dollar sign and keep the comma and period. If you did what your *SYSTEM GUIDE* suggested, you'd have blanks where the comma and period went! (We're still trying to figure out what else you can do with PUDEF. If you have some good ideas, write and let us know.) Try out the following program with PUDEF.

```
5 SCNCLR
10 PUDEF " ,.£"
20 PRINT "THE ROLLS ROYCE WILL COST YOU ";
30 PRINT USING "#$##,###.##";85641.55
40 PUDEF " ,.$"
```

Notice we returned PUDEF to the default conditions at the end of the program. Whatever you define PUDEF to be will stay put until you re-define it or turn off your computer. Therefore, it is a good habit to return it to the default conditions whenever it is used. If you constantantly have to redefine it, write a one line program to kick in the re-definition and leave it along.

PRINT USING Formats defined in Strings

A handy way to get the PRINT USING formats set up is to put them into string variables. For example, for formatting dollars and cents and percentages, you might want to do the following:

```
10 DLLAR$= "#$#####.##" : PERCENT$= "##%"
20 PRINT USING DLLAR$; 1234.56
30 PRINT USING PERCENT$;55
```

In big programs where various formats are used, you can define your PRINT USING formats at the beginning of your program, making it a lot easier to use the different outputs where required. Also, be sure to used a variable name for your string that will be easily remembered.

There are a lot of things you can do in formatting your output with PRINT USING. In some cases it will wholly replace PRINT, and in others it will not. The only way to find out is to use it. Experimentation is the heart of programming, especially with problems involving the format of your output.

Unravelling Strings

Our discussion of strings up to this point has involved "whole" strings. That is, whatever we define a string to be, no matter how long or short, can be considered a "whole" string. For example, if we define R$ as "WALK" then we can consider "WALK" to be the whole of R$. Likewise, if we defined R$ as "A VERY LONG AND WORDY MESSAGE" then, "A VERY LONG AND WORDY MESSAGE" would be the whole string of R$. There will be occasions, however, when we want to use only part of a string or tie several strings together. (When we get into data base programs, we will find this to be very important.) Also, there are applications where we will need to know the length of strings, find the numeric values of strings, and even change strings into numeric variables and back again.

=These Will Be Useful!=

I hate to admit it, but when I first learned about all of the statements we are about to discuss, I thought, "Boy, what a waste of time!" It was enough to get the simple material straight, but why in the world would anyone want to chop up strings and put them back together again? If you want only a certain segment of a string, why not simply define it in terms of that segment? And if you want a longer string, then just define it to be longer! Those were my thoughts on the matter of string formatting. However, I have now come to the point that I find it very difficult to even conceive of programming without these powerful statements. So, trust me! String formatting statements are terrific little devices to have, and if you do not see their applicability right away, you will as you begin writing more programs.

String Formatting

We will divide our discussion of string formatting into four parts: 1) Calculating the length of a string; 2) Locating parts of strings; 3) Changing strings to numeric variables and back again; and 4) Tying strings together (concatenation).

Calculating the LENgth of Strings

Sometimes it is necessary to calculate the length of a string for formatting output. Happily, your COMMODORE-128 is very good at telling you the length of a particular string. By the statement, PRINT LEN (A$) you will be given the number of characters, including spaces, your string has. Try the following little program to see how this works:

```
10 SCNCLR
20 INPUT "NAME OF STRING-> "; A$
30 PRINT A$; " HAS "; LEN(A$); " CHARACTERS"
40 PRINT  : PRINT " MORE?(Y/N) ";
50 GETKEY AN$
60 IF AN$ = "Y" THEN 20
```

Now to see a more practical application, we will look at a modified version of the centering routine we used in the last chapter.

```
10 SCNCLR
20 PRINT "ENTER A STRING LESS THAN 40
CHARACTERS" : INPUT"-> "; S$
30 SCNCLR
40 L = 20 - LEN(S$)/2 : PRINT TAB(L); S$
 50  FOR J = 1 TO 20: PRINT : NEXT J
60  PRINT "HIT ANY KEY TO CONTINUE OR 'Q' TO
QUIT ";
70 GETKEY A$
80 IF A$ < > "Q" THEN SCNCLR : GOTO 10
90 END
```

Now that we can see how to compute the LENgth of a string and then use that LENgth to compute our tabbing, let's see how we can control the input with the LEN statement. Suppose you want to write a program that will print out mailing labels, but your labels will only hold 30 characters. You want to make sure all of your entries are 30 or fewer characters long, including spaces. To do this we will write a program that checks the LENgth of a string before it is accepted.

```
10 SCNCLR
20 PRINT "ENTER A NAME LESS THAN 30
CHARACTERS INCLUDING SPACES"
30 INPUT "DO NOT USE COMMAS -> "; NA$
40 IF LEN (NA$) > 30 THEN GOTO 100 : REM
TRAP
50 PRINT : PRINT NA$
60 PRINT : PRINT "ANOTHER NAME?(Y/N) ";
70 GETKEY AN$
80 IF AN$ < > "Y" THEN END
90 GOTO 10
100 REM ****
110 REM TRAP
120 REM ****
130 SCNCLR : PRINT "PLEASE USE 30 CHARACTERS
OR LESS "
140 PRINT : GOTO 20
```

Now the first thing you should do is to break the rule!!! Go ahead and enter a string of more than 30 characters to see what happens. (If your computer gets snotty with you, you can always re-program it. It helps to remind it of that fact periodically.) If the program was entered properly, it is impossible to enter a string of more than 30 characters. From the above examples, you can begin to see how the LEN statement can be useful in several ways. There are many other ways that such statements can be employed to reduce programming time, clarify output, and compute information. The key to understanding its usefulness is to experiment with it and see how other programmers use the same statement.

Finding the MIDdle$, LEFT$, and RIGHT$ parts of a string

Suppose you want to use a single string variable to describe three different conditions, such as "POOR FAIR GOOD", but you want to use only part of that string to describe an outcome. Using MID$, LEFT$ and RIGHT$, it is possible to PRINT only that part of the string you want. For example, the following program lets you use a single string to describe three different conditions:

```
10 SCNCLR
20 X$ ="POOR FAIR GOOD"
30 PRINT "HOW DO YOU FEEL TODAY? (<P>OOR,
<F>AIR OR <G>OOD)";
40 GETKEY F$
50 IF F$ = "P" THEN PRINT LEFT$(X$,4)
60 IF F$ = "F" THEN PRINT MID$(X$,6,4)
70 IF F$ = "G" THEN PRINT RIGHT$(X$,4)
80  PRINT : PRINT : PRINT "ANOTHER GO?(Y/N)
";
90 GETKEY AN$
100 IF AN$ = "Y" THEN 10
```

Let's face it, it would have been easier to simply branch to a PRINT 'GOOD' 'FAIR' or 'POOR' and no less efficient. But, no matter, it was for purposes of illustration and not optimizing program organization. Let's see what the new statements do.

<u>STATEMENT</u>	<u>MEANING</u>
MID$(A$,N,L)	Finds the portion of A$ beginning at Nth character L characters long.
LEFT$(A$,L)	Finds the portion A$, L characters long starting at the LEFT side of the string.
RIGHT$(A$,L)	Finds the portion of A$, L characters long starting at the RIGHT side of the string.

To give you some immediate experience with these statements, try the following:

```
W$ = "WHAT A MESS" : PRINT LEFT$(W$,4) <RETURN>
G$ = "BURLESQUE" : PRINT MID$(G$,4,3) <RETURN>
X$ = "A PLACE IN SPACE" : PRINT RIGHT$(X$,5) :
PRINT RIGHT$(X$,3) <RETURN>
```

Another trick with partial strings is to assign parts of one string to another string. For example:

```
10 SCNCLR
20 BIG$ = "LONG LONG AGO AND FAR FAR AWAY"
30 LITTLE$ = MID$(BIG$,11,3)
40 AWY$ = RIGHT$(BIG$,4)
50 LG$ = LEFT$(BIG$,4)
```

```
60 PRINT : PRINT : PRINT AWY$;" "
;LG$;"";LITTLE$
70 REM BEFORE YOU RUN IT, SEE IF YOU CAN
GUESS THE MESSAGE.
```

For an interesting effect, try the following little program:

```
10 SCNCLR :  FOR I = 1 TO 10 : PRINT : NEXT
20 INPUT "YOUR NAME--> "; NA$
30 FOR I = LEN(NA$) TO 1 STEP -1 : PRINT
MID$(NA$,I,1); : NEXT I
40 SLEEP 2
45 REM ** LINE 50 USES THE NON-SHIFTED
CLR/HOME KEY **
46 REM ** NOTE HOW IT FUNCTIONS TO PLACE THE
CURSOR VERTICALLY **
47 REM ** IN CONJUNCTION WITH THE LOOP **
50 PRINT "{HOME}" : FOR V = 1 TO 11 : PRINT
: NEXT V
55 REM ** IN LINE 60 'K LOOP' SLOWS
56 REM IT DOWN FOR SLOW MOTION EFFECT **
60  FOR I = 1 TO LEN(NA$)  : PRINT MID$(NA$,
I,1); :FOR K = 1 TO 50 : NEXT K : NEXT I
70 FOR VT = 1 TO 5 : PRINT : NEXT VT :
PRINT TAB (5); "WANNA DO IT AGAIN?(Y/N) ";
80  GETKEY AN$
90  IF AN$ = "Y" THEN 10
```

Now you have probably been wondering ever since you got your computer how to make it print your name backwards. Well, now you know! (If your name is BOB you probably didn't notice it was printed backwards - try ROBERT.) Actually, the above exercise did a couple of things besides goofing off. First, it is a demonstration of how loops and partial strings (or substrings) can be used together for formatting output. Second, we showed how output could be slowed down for either an interesting effect or simply to give the user time to see what's happening. Since we're on the topic of speed, let's learn how to use your COMMODORE-128's clock. Remember we pointed out that TI$ was a "reserved variable," and now we will see why. Try the following in the Immediate Mode:

```
TI$ = "101030" <RETURN>
```

Now wait a few seconds and enter,

```
PRINT TI$ <RETURN>
```

The value of TI$ changed from 101030 to something else! If you waited for just a few seconds, 101030 changed to 101050 or somewhere in that range. To see what is happening, let's break it down in to hours, minutes and seconds.

10 10 30 = 10 hours 10 minutes 30 seconds.

We'd say that time is 10:10 and 30 seconds on a normal clock. Well, that's exactly what TI$ does. It ticks off the seconds, then minutes and finally hours. To better see this, let's make a little clock program.

```
10 SCNCLR : PRINT "COMMODORE-128 CLOCK"
20 FOR I = 1 TO 4 : PRINT : NEXT
30 PRINT"ENTER TIME(00 HRS 00 MINS 00 SECS)"
40 INPUT TI$
50 SCNCLR
60 PRINT "{HOME}":FOR I=1 TO 10:PRINT: NEXT
70 PRINT "COMMODORE TIME-> ";TI$ : GOTO 60
```

When you run this program, be sure to enter all 6 digits for hours, minutes and seconds. For example, if the time you want to enter is 8:14, enter 081400, not just 814. Besides using TI$ for a clock to display time on your screen, you can also use it for a timer in your programs. By first setting a value for TI$ and then checking it in your program, you can have timing for responses. The following is a simple math game that adds the element of time:

```
10 SCNCLR : FOR I = 1 TO 5 : PRINT : NEXT :
TI$ = "000000"
20 INPUT "ENTER 1ST NUMBER->"; A
30 INPUT "ENTER 2ND NUMBER->"; B
40 PRINT : PRINT "WHAT IS"; A ; "+"; B;
50 INPUT C
60 IF A + B < > C THEN 200
70 IF TI$ > "000010" THEN GOTO 100
80 PRINT : PRINT "THAT'S RIGHT!!!!"
90 SLEEP 2 : GOTO 10
100 REM *************
110 REM TOO MUCH TIME
120 REM *************
```

```
130 SCNCLR : PRINT : PRINT : "YOU RAN OUT OF
 TIME!"
140 SLEEP 2 : GOTO 10
200 REM ************
210 REM WRONG ANSWER
220 REM ************
230 PRINT "THAT'S NOT QUITE RIGHT"
240 INPUT "PRESS RETURN TO CONTINUE";CR
250 GOTO 10
```

Examine the program carefully. Note how the time is checked in line 70 and how it is reset to "000000" each time the process is restarted.

String Searching

Some programs require finding one string inside another string. (Sure it sounds weird, but there are a lot of really neat things you can do with it. Honest.) Using the INSTR statement, along with the substring statements (MID$ and those guys), it is possible to find parts of a string and then do something useful with them. We'll start off by seeing how INSTR works, and then we'll do something practical with it. To begin with, it returns the beginning position in a string of the search string. For example, let's see where CUP is in HICCUP.

```
10 A$="HICCUP"
20 PRINT INSTR(A$,"CUP")
```

When you RUN the program, you get a '4' indicating that the word "CUP" begins in the forth letter of the word "HICCUP." Since it is unlikely you will need our example application any time soon, let's see how it might be used in a practical program.

Suppose you have a bunch of strings that are arranged with last name first and first name last. For instance, Joe Blow is in a string:

    Blow Joe

It is not unusual to arrange strings that way for purposes of alphabetical sorting. However, when you want to create lists or mailing labels, it just doesn't look right having people's names backwards. We'll create a little program that will fix things so that first names come first and last names last and then go through it and explain how it works.

```
10 SCNCLR
20 INPUT "LAST NAME AND FIRST NAME";NA$
30 SP$=" " : REM A BLANK SPACE
40 L=INSTR(NA$,SP$)
50 NF$=MID$(NA$,L+1)
60 NL$=LEFT$(NA$,L-1)
70 PRINT NF$;SP$;NL$
```

When you RUN the program, be sure to put a space between the last and first name when prompted. Stepping through the program, we find:

Step 1. We will look for the space between the last and first names with SP$ which has been defined as a blank space. (Line 30)

Step 2. Using INSTR, we store the starting position of the space in the variable L. (Line 40)

Step 3. The first name is everything to the right of the space; so using MID$, we define NF$ as everything from the right of the space to the end of the string. *Remember*, if we do not put a second parameter value in for MID$, it defaults to everything from the first parameter (starting position) to the end of the string. (Line 50)

Step 4. Conversely, everything to the left of the space is the last name, so we load that into the string variable, NL$ using LEFT$. (Line 60)

Step 5. All that we have to do now is to rearrange things in the order we want and PRINT them out. (Line 70)

=A Very Short Sermon=

By breaking down a problem into simple little tasks, it is a lot easier to write programs.

Converting Between String and Numeric Variables

Strings to Numbers. Now we're going to learn about changing strings to numbers and numbers to strings. If you're like me, when I first found out about these statements, I thought they were pretty useless. After all, if you want a string use a string variable, and if you want a number use a numeric variable. Simple enough, but again, once you understand their value, you wonder how you could do without them. To get started, let's RUN the following program:

```
10 SCNCLR
20 FOR I = 1 TO 5 : READ NA$(I)  : NEXT I
30 FOR I = 1 TO 5
40   X(I)  = VAL(RIGHT$(NA$(I),1))
50 NEXT I
60 FOR I = 1 TO 5 : PRINT "OVERTIME PAY= $";
 X(I) * (1.5 * 7)  : NEXT I
70 DATA SMITH 7, JONES 8, MCKNAP 6, JOHNSON
 2, KELLY 3
```

Using DATA that were originally in a string format, we were able to change a portion of that string array to a numeric array. By making such a conversion, we were able to use our mathematical operations on line 60 to figure out the overtime pay for someone receiving time and a half at seven dollars ($7) an hour. Well, that's pretty interesting, but we don't have a list of who got what and the total overtime paid! Why don't you try it yourself. Change the program so that everyone's name appears with the amount of overtime they received and a total overtime paid. (Hint: You are looking for the substring LEFT$ (NA$(I), LEN (NA$(I)-2)) since you want to drop the number and space after each name.) When you get it, write me a letter to show me how you figured it out.

It always helps to do a few immediate exercises with a new statement to get the right feel; so try these:

```
A$ = "123" : PRINT VAL(A$) + 11 <RETURN>
Q$ = "99.5" : PRINT VAL(Q$) * 7 <RETURN>
SALE$ = "44.95" : PRINT "ON SALE AT HALF
PRICE ->$"; VAL(SALE$) / 2 <RETURN>
```

```
DO$ = "$103.88" : DN$ = "$18.34" : PRINT VAL
(RIGHT$(DO$,6)) + VAL (RIGHT$(DN$,5))
<RETURN>
```

NOTE: Since you may want to SAVE the above examples on tape or disk, all you have to do is to add a line number and SAVE them as little programs.

Numbers to Strings. All right, let's now go the other way. We saw why we might want to change strings to numbers, but we may also want to change numbers to strings. To make the conversion we use the STR$ statement. For example, look at the following program:

```
10 SCNCLR
20 PRINT "ENTER A NUMBER WITH 5 DIGITS "
30 INPUT " AFTER THE DECIMAL POINT ";A
40 A$ = STR$(A)
50 PRINT : PRINT LEFT$ (A$,4)
```

As you can see you have truncated the number to 3 characters including the decimal point. (Change LEFT$ to RIGHT$ in line 40 and you will get the rightmost 4 <not 3> characters of the string. (Only five people in the universe know why it does this, and they aren't talking. However, I'll bet it has something to do with first character in numbers being an *invisible* sign of plus or minus.) Now, let's do some in the immediate mode to get some practice.

```
A = 5.00 : A$ = STR$(A) : PRINT A$ <RETURN>
 V = 2345 : V$ = STR$(V) : PRINT V$
 <RETURN>
BUCKS = 22.36 : BUCKS$ = STR$(BUCKS) : PRINT
 LEFT$(BUCKS$,2) <RETURN>
```

Combining Strings with Concatenation

We have seen how we can take a portion of a string and PRINT it to the screen. Now, we will tie strings together. This is called CONCATENATION and is accomplished by using the "+" sign with strings. For example:

```
10 SCNCLR
20 INPUT "YOUR FIRST NAME -> "; NF$
30 INPUT "YOUR LAST NAME  -> "; NL$
40 NA$ = NF$ + NL$
```

```
50 PRINT NA$
```

A little messy, huh? However, you can see how NF$ and NL$ were tied together into a single larger string. Now, change line 40 to read

```
40 NA$ = NF$ + " " + NL$
```

This time when you RUN the program, your name will turn out fine. Not only did we concatenate string variables, we also concatenated strings themselves. For example, it is perfectly all right to do the following:

```
PRINT "ONE" + "ONE" <RETURN>
```

Now there isn't much you can do with ONEONE, but we can see the principle of operation with concatenating strings. To see something a little more practical and a nifty trick to boot, try the following program.

```
10 SCNCLR
20 FOR X=1 TO 40
30 LINE$=LINE$ + "-"
40 NEXT X
50 PRINT : PRINT
60 PRINT LINE$
```

Setting Up Data Entry

Now that we have a firm grip on numerous statements, it is time we begin thinking seriously about organizing our programs. The first thing we must do is to arrange our data entry in a manner that we ourselves and others can understand. This involves blocking elements of our program and deciding what variables and arrays we will be using. Also, when we enter data, we want to make sure that we are entering the correct type of data; so we have to set "traps" so that any input that is over a certain length or amount can be checked against our parameters. Let's look at a way to make our strings a certain length (no shorter or longer than a length we want). We've already discussed how to keep strings to a maximum length, so let's see how to keep them to a minimum as well. This latter process is referred to as "padding."

```
10 SCNCLR
```

```
20 FOR I = 1 TO 8 : PRINT : NEXT I
30 INPUT "YOUR COMPANY-->"; CM$
40 IF LEN(CM$) <= 10 THEN 70
50 IF LEN(CM$) > 10 THEN PRINT "10 OR FEWER
 CHARACTERS PLEASE" : REM TRAP FOR TOO LONG
A NAME
55 REM PRESS THE CONTROL AND 9 KEYS
 SIMULTANEOUSLY {CTRL-9} IN LINE 60
60  PRINT :PRINT "{CTRL-9}HIT ANY KEY TO
 CONTINUE-> "; :
70 GETKEY A$
80 GOTO 10
90 IF LEN(CM$) < 10 THEN CM$ = CM$ + "X" :
 GOTO 90 : REM PADDING
100 SCNCLR : FOR I = 1 TO 8 : PRINT : NEXT
110 PRINT "THE COMPUTER HAS DECIDED THAT " ;
120 PRINT CM$; " SHOULD GIVE YOU A RAISE!"
```

Now if YOUR COMPANY <CM$> is less than 10 characters, you will see some "X's" stuck on the end. These were put there to show you how padding works. Now change the "X" to " " <a space> in line 90 and see what happens. Go ahead. The second time you ran the program, if your company's name was less than 10 characters, there are a lot of blank spaces after the company name. To remove the spaces, we would enter:

```
95 IF MID$(CM$,LEN(CM$),1) = " " THEN CM$ =
 LEFT$(CM$,(LEN(CM$)-1): GOTO 95
```

Setting Up Data Manipulation

Once you have organized your input, the next major step is performing computations with your data. There are essentially two kinds of data manipulation you will deal with:

1. **Numeric** - Manipulating numeric data with mathematical operations.
2. **String** - Manipulating strings with concatenation and substring statements.

Most of the string manipulations are for setting up input or output, and so we will concentrate on manipulating numeric data. We will use a simple example that keeps track of three manipulations: (1) additions (2) subtractions and (3) running balance. This will be our check book program we started earlier.

```
10 SCNCLR : DLLAR$="#$#####.##"
20 REM *************************
30 REM BEGIN INPUT & HEADER BLOCK
40 REM *************************
50 CB$="=COMPUTER CHECK-BOOK="
60 L=20-LEN(CB$)/2
70 PRINT TAB(L);CB$
80 FOR I=1 TO 4 : PRINT : NEXT
90 INPUT "ENTER BALANCE->";BA
100 PRINT : PRINT : PRINT"1. ENTER DEPOSITS"
110 PRINT: PRINT "2. DEDUCT CHECKS"
120 PRINT : PRINT "3. EXIT"
130 FOR X=1 TO 7 : PRINT : NEXT X
140 PRINT "CHOOSE BY NUMBER"; : INPUT A
150 ON A GOTO 210,410,610
160 GOTO 130 : REM TRAP
200 REM ********************
210 REM DATA MANIPULATION #1
220 REM ********************
230 SCNCLR  :FOR I=1 TO 6  :PRINT : NEXT
240 INPUT "ENTER AMOUNT OF DEPOSIT";DP
250 BA=BA+DP:REM RUNNING BALANCE
260 PRINT:PRINT:PRINT"YOU NOW HAVE ";
270 PRINT USING DLLAR$;BA
280 PRINT:INPUT "MORE DEPOSITS(Y/N)";AN$
290 IF AN$="Y" THEN 230
300 PRINT:INPUT "WOULD YOU LIKE TO DEDUCT
 CHECKS(Y/N)";AN$
310 IF AN$="N" THEN 610
320 IF AN$="Y" THEN 410
330 SCNCLR : GOTO 300 : REM TRAP
400 REM ******************
410 REM DATA MANIPULATION 2
420 REM ******************
430 SCNCLR : FOR I=1TO6 : PRINT:NEXT
440 INPUT "ENTER AMOUNT OF CHECK ";CK
450 BA=BA-CK : REM RUNNING BALANCE
460 PRINT : PRINT"YOU NOW HAVE ";
470 PRINT USING DLLAR$;BA
480 PRINT : PRINT
490 INPUT "MORE CHECKS(Y/N)-'Q' TO QUIT";AN$
500 IF AN$="Y" THEN 430
510 IF AN$="Q" THEN 610
520 PRINT:INPUT"ANY DEPOSITS(Y/N)";AD$
```

```
530 IF AD$="Y" THEN 210
540 GOTO 480 : REM TRAP
600 REM ****************
610 REM TERMINATION BLOCK
620 REM ****************
630 SCNCLR : FOR I=1 TO 400  :PRINT"$"; :
 NEXT
640 PRINT"YOU NOW HAVE A BALANCE OF ";
650 PRINT USING DLLAR$;BA
```

This program is designed to provide a simple illustration of how to block data manipulation. However, there are some problems with it in the output. We are not getting the 0's on the end of our balance! This is an "output" problem we will discuss in the following section, but before we continue, make sure you understand how we blocked the data manipulation. We used only three variables:

BA = BALANCE
CK = CHECK
DP = DEPOSIT

When we subtracted a check, we simply subtracted CK from BA, and when we entered a deposit, we added DP to BA. In this way we were able to keep a running balance and at the very end BA was the total of all deposits and checks. By keeping it simple and in blocks we were able to jump around and still keep everything straight.

Scroll Control

One of the big problems in output occurs when you have long lists that will scroll right off the screen. For example, the output of the following program will kick the output right out the top of the screen:

```
10 SCNCLR
20 FOR X = 1 TO 100 : PRINT X : NEXT
```

Instead of numbers, suppose you have a list of names you have sorted or some other output you wanted to see before they zipped off the top of the screen. Depending on the desired output, screen format and so forth there are several different ways to control the scroll. Consider the following:

```
10  SCNCLR
```

```
20   FOR X=1 TO 100
30   IF C=23 THEN GOSUB 100
40   C=C+1
50   PRINT X : NEXT X
60   GOSUB 100
70   END
100 REM *******
110 REM HOLD IT
120 REM *******
130  PRINT : PRINT "HIT ANY KEY TO CONTINUE";
140  GETKEY A$
150   SCNCLR : RETURN
```

POS(0)

While we are on the topic of locating the cursor position, let's take a look at POS(0) as well. The POS(0) variable locates the horizontal position, and it allows you to control side to side scrolling. For example, the following program enters a PRINT statement to give a linefeed when a certain horizontal position is exceeded.

```
10 SCNCLR
20 FOR X=1 TO 40
30 Y=POS(0)
40 IF Y>30 THEN PRINT
50 PRINT "HERE";
60 NEXT X
```

Run the program, and all the "HERE's" are arranged in a block. Now delete line 40 in the program and RUN it again. The second time, the arrangement is not blocked. In larger programs when you do not want to have to determine where to locate the cursor every time there is a new screen output, you can store the values of POS(0) in variables, and then use those variables to move the cursor back to the desired position. The key to understanding how to use these two variables is to experiment with them in formatting output.

REMEMBER!! You, not the computer, are in CONTROL! You can have your output any way you want it. To use more of the screen, you could have the output tabbed to another column after the vertical screen is filled. For example:

```
10 SCNCLR
20 FOR X=1 TO 36
30 IF X>18 THEN GOSUB 100
40 PRINT X : NEXT X
50 END
100 REM *******
110 REM ARRANGE
120 REM *******
130 IF X=19 THEN PRINT "{HOME}"; : REM
 UNSHIFTED CLR/HOME
140 PRINT TAB(20);
150 RETURN
```

Another trick is to make "calculated columns" that come up simultaneously. For example, the following program lines up output in three equal columns. If the number is not equally divisible by three, then it tags on the extra values in the last column. Notice how we created a MOD (modulo=division remainder) function to determine whether or not the number of columns would be even. See if you can change the program to line up the numbers evenly with the extra values being placed in the first two columns.

```
10  SCNCLR
20  INPUT "ANY NUMBER";N
30  Y=INT(N/3) : MOD=N-(3*Y)
40  FOR X=1 TO Y
50  PRINT X,X+Y,X+(2*Y)
60  NEXT
70  IF MOD > 0 THEN GOSUB 100
80 END
100 FOR K=X TO X+(MOD-1)
110 PRINT,,K+(2*Y):NEXT K
120 RETURN
```

You get the idea. Format your ouput in a manner that best uses the screen and your needs and get that scroll under control!

Yes-The Commodore 128 *Does Do* Windows

A final formatting command we will discuss is WINDOW. This command allows you to partition your screen into separate little screens or 'windows.' You may want your input in one window and output in another, or you may want different colors for different windows. The window command can do a lot to make your programs look professional. To get started, we'll partition your 40 column screen into four little screens. For each window, you include the opposite corners for the windows in terms of the X (horizontal)/Y (vertical) positions. The full screen on 40 and 80 columns can be imagined as :

```
WINDOW 0,0,39,24 (40 columns)
WINDOW 0,0,79,24 (80 columns)
```

Every other window falls within those maximum parameters. Now, let's see how we can program four windows.

```
10 SCNCLR
20 FOR X=1 TO 4
30 READ T1,T2,B1,B2
40 WINDOW T1,T2,B1,B2
50 SCNCLR : PRINT X
60 NEXT X
100 REM ***********
110 REM WINDOW DATA
120 REM ***********
130 DATA 0,0,19,12
140 DATA 20,0,39,12
150 DATA 0,13,19,24
160 DATA 20,13,39,24
```

Each time through the loop, the program opens another window PRINTing the loop value in the window. Notice when you RUN the program how SCNCLR in line 50 clears only the current window and not the entire screen. If you LIST the program, you will find it is crowded into the lower right hand corner of your screen. Hit the RUN/STOP and RESTORE keys simultaneously to get back your full screen or key in `WINDOW 0,0,39,24` <RETURN>.

By adding a '1' to the end of the WINDOW parameter list, each time a window is 'opened', it is cleared. That is easier than putting in SCNCLR every time you open a WINDOW. The following program shows you how to do that and how to partition the screen into an "input" and "output" window. We'll use the opposite corners of the screen and only place the "1" at the end of the 'INPUT WINDOW' so that you can see the difference between how it is cleared and the 'OUTPUT WINDOW' is not.

```
10 SCNCLR
20 GOSUB 100
30 INPUT "WHAT'S UP";WU$
40 GOSUB 200
50 PRINT WU$
60 GOSUB 100
70 PRINT "ANOTHER(Y/N)?"
80 GETKEY A$
90 IF A$ <> "Y" THEN END : ELSE GOTO 30
100 REM ************
110 REM INPUT WINDOW
120 REM ************
130 WINDOW 0,0,19,12,1
140 RETURN
200 REM *************
210 REM OUTPUT WINDOW
220 REM *************
230 WINDOW 20,13,39,24
240 RETURN
```

Summary

The formatting of programs makes the difference between a useful and not-so-useful application of your computer. The extent to which your program is well organized and clear, the better the chances are for simple yet effective programming. Formatting is more than an exercise in making your input/output fancy or interesting. It is a matter of communication between your COMMODORE 128 and you! After all, if you can't make heads or tails of what your computer has computed, the best calculations in the world are of absolutely no use.

In the same way it is important to have your computer tell you what you want, it is also important to write your programs so that you and others can understand what is happening. By using "blocks" it is easier to organize and later understand exactly what each part of your program does. Obviously, it is possible to write programs sequentially so that each command and subroutine is in an ascending order of line numbers, but to do so would mean that you would have to repeat simple and/or complex operations which could be better handled as subroutines. Also, it would be considerably more difficult to find bugs and make the appropriate changes. In other words, by using a structured approach to programming, you make it simpler, not more difficult.

Finally, you should begin to see why there are commands for substrings and all the fuss about TABs,curosr control keys, POS(0), WINDOW and PRINT USING. These are handy tools for organizing the various parts in a manner which gives you complete control over your computer's output. What may at first seem like a petty, even silly command in COMMODORE 128 BASIC 7.0, upon a useful application, can be appreciated as an excellent tool. Therefore, as we delve deeper into your computer, look at the variety of commands as mechanisms of more efficient and ultimately simpler control and not a complex "gobbleygook" of "computerese" for geniuses. After all, if you've got this far, you should realize that what you thought would be impossible at the outset is not only relatively easy, it's a lot of fun too.

Inside The Mind of Your Commodore 128

Introduction

The topics of this chapter are more "code like" and contain the kinds of commands that look frightening. At least that's how I interpreted them when I first saw them. Many of the functions can be done with commands we already know, but many cannot. Still others, as we will see, can be accomplished better using these new commands. Like so much else you have seen in this book, what at first may appear to be "impossible" is really quite simple once you get the idea. More importantly, by playing with the commands, you can quickly learn their use.

The first thing we will learn about is the ASCII code. ASCII (pronounced ASS-KEY) stands for the AMERICAN STANDARD CODE for INFORMATION INTERCHANGE. Essentially, this is a set of numbers that have been standardized to mean certain characters. In COMMODORE-128 BASIC the CHR$ (character string) command ties into ASCII and can be used to directly output ASCII. As we will see, the CHR$ command is very useful for outputting special characters.

The next commands have to do with directly accessing locations in your computer's memory. The first, POKE, puts values into memory and the second, PEEK, looks into memory addresses and returns the values there. We will examine several different uses of these two commands. These commands are essential for producing certain types of graphics and sound.

The ASCII Code and CHR$ Functions

In a couple of places we have used control characters in programs, such as CONTROL-9. In the program all we saw was something like the following:

```
    PRINT "{INVERSE R}": REM CONTROL-9
```

What that means is that we enter the CONTROL-9 between the quote marks, but an inverse "R" is there. Unfortunately, we cannot see the CONTROL-9 when we list our program to printer or screen; so we have to use a REM statement to let us know what's there or remember that an inverse "R" is really a CONTROL-9. Another way to access any characters we want, including control characters, is to use CHR$ commands and the ASCII code. In APPENDIX A there is a complete listing of ASCII that you will want to examine. Whenever we want to access a character, all we have to do is to enter the CHR$ and the decimal value of the character we want. For example enter the following:

```
    PRINT CHR$(65) <RETURN>
```

You got an "A." That's simple enough and not too interesting. On the other hand, try the following little program, and I'll bet you couldn't do it without using the CHR$ function:

```
    10 PRINT CHR$(147) : REM USES ASCII FOR
SCNCLR
    20 QU$ = CHR$(34) : REM USES ASCII VALUE FOR
QUOTE MARKS
    30 FOR I = 1 TO 20 : PRINT : NEXT  : PRINT
```

```
CHR$(18);"HIT ANY KEY TO CONTINUE OR ";
40 PRINT QU$ ; "Q" ; QU$ ; " TO QUIT ";
50 GETKEY AN$
60 IF AN$ = "Q" THEN END
70 GOTO 10
```

RUN the program and look carefully. Note the quotes around the Q. If we tried to PRINT a quote mark, the computer would think it got a command to begin printing a string. However, bY defining QU$ as CHR$(34) we were able to slip in the quote marks and not confuse the output! (Just for fun, see if you can do that without using the CHR$ command.) Also, did you notice how we began the program? Instead of using the SCNCLR, we used CHR$(147). We did not have to put in the quote (") marks around CHR$(147) as we did with SCNCLR. Likewise, we used CHR$(18) instead of a CONTROL-9 to set the inverse mode. To see what different characters you have available, RUN the following program:

```
10 PRINT CHR$(147)
20 FOR X = 32 TO 127 : PRINT CHR$(X); : NEXT
30 FOR X = 158 TO 191 : PRINT CHR$(X) ; :
 NEXT
```

Voila! There you have all of your symbols. Before we go on, though, let's see some other symbols simply by pressing two keys. Hold down the COMMODORE key (in the lower left hand corner of your keyboard) and press the SHIFT key. The first set of letters were printed in lower case, and the symbols, beginning with the "spade" changed to upper case letters. Thus, depending whether or not the lower case letters are "on" or "off," CHR$'s will output different symbols.
 Now, to watch funny things happen to your screen RUN the following program.

```
10 PRINT CHR$(147)
20 FOR X = 0 TO 31
30 PRINT CHR$(X) ;
40 PRINT "WHAT IS THIS";
50 NEXT X
```

Not much happened since in that range of ASCII (from 0 to

31) you ran through the control characters. Your characters turned colors and got shifted into lower case, and everything else was invisible. Just press CONTROL-2 to get your white characters back and COMMODORE-KEY/SHIFT to pop your characters back to upper case. To get used to your increased power over your computer, try the following little programs:

```
10 PRINT CHR$(147)
20 LB$ = CHR$ (54) : RB$ = CHR$(52)
30 CO$ = "COMMODORE" + CHR$(45) + LB$ + RB$
40 L = 20 - LEN (CO$)/2 : PRINT SPC(L); CO$
50 FOR I = 1 TO 20 : PRINT CHR$(32) : NEXT

10 PRINT CHR$(147)
20 PRINT CHR$(18); CHR$(28);
30 FOR X = 1 TO 35 : PRINT CHR$(32) ; : NEXT
40 PRINT CHR$(5)
50 REM BEFORE YOU RUN THIS, SEE
60 REM IF YOU CAN FIGURE OUT WHAT WILL
 HAPPEN
```

On the last program, you will get an idea of the use of CHR$ commands with graphics. The red bar was created using CHR$(32), a space, after the color red had been set with CHR$(18) <CONTROL-9> and CHR$(28) <CONTROL-3>. In the next chapter on graphics, we will use the CHR$ command a good deal in creating pictures, charts and graphs. (By the way, to reset everything to normal, use the RUN/STOP and RESTORE keys.) The following program is a handy little device for printing out all of the CHR$ values to screen. Save it to tape or disk to use as a handy reference guide to look up CHR$ values and symbols.

CHR$ MAP

```
10  PRINT CHR$(147)
20  GOSUB 300
30  FOR I = 33 TO 99
40  IF I = 34 THEN 400
50  PRINT I; ".  ="; CHR$(I),
60  NEXT
70  PRINT : PRINT "HIT ANY KEY TO CONTINUE";
80  GETKEY A$
90  PRINT CHR$ (147)
```

```
100 GOSUB 300
110 FOR I = 100 TO 127: PRINT I;".  =" ;
 CHR$ (I), : NEXT
120 FOR I = 161 TO 191 : PRINT I; ". =";
 CHR$(I), : NEXT
130  PRINT : PRINT : END
300  FOR I = 1 TO 4 : PRINT " CHR$ / S", :
 NEXT
310  RETURN
400  PRINT I; ".  ="; "''", : REM THERE ARE
2 SHIFT 7'S BETWEEN THE QUOTE MARKS
410  GOTO 60
```

The program, CHR$ MAP, can be used as a handy reference for you to look up the CHR$ values of different symbols. You may have noticed that the program branches to a subroutine at line 400 if I = 34. The reason for that is because once a quotation mark - CHR$(34) - is encountered, inverse brackets will be printed in the rest of the output. To avoid that, we made a "phony quote mark" using two apostrophes (SHIFT 7). This left a gap between 34 and 35, but it looks a lot better than all those inverse brackets! Also, we left out CHR$ values that would either lock up the display, clear the screen, change the colors or somehow mess up the output. See if you can made a program that will include useful CHR$ values (such as CONTROL-9 and colors) but not destroy the output.

Programming With ESC - CHR$(27)

A final area where you can have some fun with CHR$ is to use the ESC sequences from your editor. Since ESC is a special key that cannot be part of a program line, the only way to use it is from the immediate mode. However, with CHR$(27), it's possible to incorporate ESC into your program. The following program shows how to use the ESC-V and ESC-W sequences to scroll up and down your screen.

```
10 PRINTCHR$(147)
20 PRINT"HI THERE PROGRAMMER"
30 FOR X=1 TO 23:PRINT CHR$(27)+"W";:NEXT
40 FOR X=1 TO 24:PRINT CHR$(27)+"V";:NEXT
```

Experiment with CHR$(27) to see what else you can do. Try out the various editor sequences to find what is possible.

POKES and PEEKS
Looking inside your Commodore-128's Memory.

At first you won't have too many uses for POKES and PEEKS, but as you begin exploring the full range of your computer's capacity, they will be used more and more. Basically, a POKE command places a value into a given memory location and a PEEK command returns the value stored in that location. For example, try the following:

```
    POKE 2048, 255 : PRINT PEEK (2048) <RETURN>
```

You should have gotten "255" since the POKE command entered that value into location 2048 and PRINT PEEK (2048) printed out the value of that address. That's relatively simple, but more is going on than storage of numbers. The key importance of POKE and PEEK involves what occurs in a given memory location when a given value is entered. In some locations nothing other than the storage of the number will occur, as in our example above. However, with other memory locations, very precise events occur. What we will do in the remainder of this section is to examine some of the more useful locations for POKEing and PEEKing in your COMMODORE-128. We will not be getting into the more complex elements of POKEs and PEEKs, however.

=A TALE OF TWO NUMBER SYSTEMS=

When using POKEs and PEEKs, we use decimal numbers for accessing locations. However, much of what is written about special locations in your PROGRAMMER'S REFERENCE GUIDE available for your COMMODORE-128 is written in HEXADECIMAL, generally referred to as HEX. Since we've used decimal notations for counting all our lives, it seems to be a "natural" way of doing things. However, decimal is simply a "base 10" method of counting and we could use a base of anything we wanted. For reasons I won't get into here, "base 16", called HEXADECIMAL is an easier way to think about using a computer's memory, and that's why so much of the notation we see is in HEX. HEX is counted in the same way as decimal except it is done in groups of 16, and it uses alphanumeric characters instead of just numeric ones. You can usually tell whether a number is HEX since they are typically preceded by a dollar-sign (e.g.

$45 is not the same as decimal 45), and often there are alphabetic characters mixed in with numbers. (e.g. FC58, AAB, 12C). The following is a list of decimal and hexadecimal numbers.

Decimal	Hexadecimal
0	$0
1	$1
2	$2
3	$3
4	$4
5	$5
6	$6
7	$7
8	$8
9	$9
10	$A
11	$B
12	$C
13	$D
14	$E
15	$F
16	$10

As you can see, instead of starting with double digit numbers at 10, hexadecimal begins double digits at decimal 16 with a $10. In the major memory locations of interest in your COMMODORE-128 PROGRAMMER'S REFERENCE GUIDE, both the decimal and hexadecimal numbers are given.

Hex and Decimal Conversion

Your Commodore has some nice built-in statements for converting between hex and decimal numbers. HEX$ converts decimal values into hexadecimal, and DEC converts hex into decimal. Try the following from the immediate mode:

```
PRINT HEX$(254) <RETURN> Converts to hex.
PRINT DEC ("ABC") <RETURN> Converts to decimal.
```

Now that you know ABC is *really* 2748, don't forget it! For the time being, you probably will not be using hex and decimal conversion, but it is important to know a little about it. The following is a handy utility for your program library that will quickly convert between the two number systems for you.

Hexconvert

```
10 SCNCLR : RESTORE
20 FOR X=1 TO 5 : PRINT : NEXT X
30 FOR X=1 TO 2 : READ R$
40 PRINT X;". ";R$ : NEXT
50 DATA  DECIMAL TO HEX,HEX TO DECIMAL
60 PRINT:PRINT "CHOOSE ONE"
70 GETKEY A
80 IF A=2 THEN 200
90 IF A<>1 THEN 60
100 REM **************
110 REM DECIMAL TO HEX
120 REM **************
130 SCNCLR
140 INPUT "DECIMAL VALUE"; D
150 PRINT "HEX VALUE= $";HEX$(D)
160 GOTO 260
200 REM **************
210 REM HEX TO DECIMAL
220 REM **************
230 SCNCLR
240 INPUT "HEX VALUE";H$
250 PRINT "DECIMAL VALUE=";DEC(H$)
260 PRINT : PRINT "HIT ANY KEY"
270 GETKEY A$ : GOTO 10
280 GOTO 10
```

=A ROTTEN TRICK!!=

When you start POKEing and PEEKing into different locations of your COMMODORE-128, you will not always get what you expect. In the decimal addresses from 4864 through 5120, you will be pretty safe since this is free RAM However, other locations are the "homes" of special routines that will react directly to anything POKEd into them. For example, if you POKE 768,255 : PRINT PEEK (768), your machine will lock up, and not even RUN/STOP and RESTORE will unlock it. You have to turn your computer off to free it up. Now if you slipped that into one of your programs and gave it to a friend, it would lock up his machine, and that would be a Rotten Trick! Of course, you wouldn't ever do anything like that. Would you?

Now let's take a look at some places to POKE. We will begin with your text screen.

POKEing the Text Screen

Another use of POKEs is to enter a character to a location on your 40 column text screen. Each character has a different value between 0 and 255. Your screen can be envisioned as a set of addresses on a 40 by 25 grid beginning with decimal location 1024 ($400) and ending at 2023 ($7E7). That gives you exactly 1000 locations on your screen where you can place text. The addresses are contiguous, and by using FOR-NEXT loops, it is a simple matter to enter sequential lines of text. Or, using POKEs, you can put text anywhere on the screen you want. To get an idea of what you will see, try the following POKEs:

```
PRINT CHR$(147) :POKE 1190, 1 : POKE 1191,
 129 <RETURN>
PRINT CHR$(147) : FOR I = 1880 TO 1890 :
 POKE I, 81 : NEXT <RETURN>
PRINT CHR$(147) : FOR I = 1240 TO (1240 +
255) : POKE I, I - 1240 : NEXT <RETURN>
```

The first line showed different addresses for normal and inverse "A" located at adjacent addresses. The next exercise used a sequence of addresses from 1880 to 1890 and put in a white ball at each location. Finally, the third exercise used adjacent memory locations to insert a sequence of ASCII characters.

Next, let's take a look at the entire screen area using POKEs and the letter "A." You will remember that an 'A' is CHR$(65), but when we want to POKE an 'A', we use '1'. That's because screen memory and ASCII are arranged differently. The screen memory alphabet begins with 1 and ends with 26, which makes it very easy to remember. Also, we will see how to use the easily remembered beginning screen address of hex $400 in a loop, and a mystery command, SYS.

```
10 SYS DEC("C000")
20 B$="400" : E$="7E7"
30 FOR X=DEC(B$) TO DEC(E$)
40 POKE X,1
50 NEXT X
```

(Further on in this chapter, we will reveal the mystery of SYS....)

The next program will introduce you to the concept of an "offset" in programming. Basically, an offset is a number that will add or subtract a specified value. There are two different offsets in the program to note. The first is "127" used in determining the maximum address for the loops beginning in lines 20 and 40. Since we want to POKE in 128 characters (from 0 to 127), we set our first offset to 127 and then terminate our screen location at the offset plus our beginning location. Since we begin at 0 (1024-I), we will end at 127 since that is our offset. Secondly, we use an offset of 128 in line 50 to get the inverse characters we generated in our first set. That is because any character we POKE in from 0 to 127 has the inverse same character at a value of the first character plus 128. Thus, for any character we want to display in inverse, we simply add 128 to the original POKE value.

```
10 PRINT CHR$(147)
20 FOR I = 1024 TO (1024+127)
30 POKE I,(I-1024) : NEXT
40 FOR I = 1424 TO (1424+127)
50 L = I-1424 : POKE I,L+128 : NEXT
```

```
60 FOR I = 1 TO 15 : PRINT : NEXT
```

You might wonder why line 60 was included. Take it out and see what happens. The reason for that is because the cursor follows the line numbers in the program and not the screen locations being POKEd. Therefore, you can POKE in a screen character from the HOME position on the screen, and even though the location will output a character to the bottom of the screen, the cursor will remain near the top. Try it and see.

In order to easily see what characters are produced with different values we POKE into screen locations, the following program allows you to INPUT a value and then displays the character on the screen for you. Of particular interest in this program are lines 50 and 60. Line 50 prints out a message and ends it with a blank instead of a semi-colon. However, when the program is RUN the character output is right next to the end of the string we entered in line 50. The reason for that is we POKEed the output in a screen address right next to the end of our string. We would have placed a semi-colon, comma or blank at the end of line 50 and the output would have been in the same place. Try it and see.

```
10 PRINT CHR$(147)
20 PRINT CHR$ (19) : PRINT: PRINT :  INPUT
 "ENTER A NUMBER FROM 0 TO 255-> "; X
30 IF X > 255 THEN 20
40 PRINT CHR$(19) : FOR I = 1 TO 11 : PRINT
: NEXT
50 PRINT "THE CHARACTER FOR"; X ; "IS "
60 POKE (1504 + 25), X
70 PRINT : PRINT : CHR$(18); "HIT ANY KEY TO
 CONTINUE OR 'Q' TO QUIT";
80 GETKEY HK$
90 IF HK$ < > "Q" THEN 10
```

The Color Screen

In addition to being a text screen memory map, there is a color screen as well. You might think of the color screen overlayed on the text screen with a different beginning address. It begins at 55296 ($D800) and ends at 56295 ($DBE7). By POKEing one of those locations, it is possible to change the color of the character. The concept is the same as for

POKEing characters except colors are entered. This next program runs through the 16 colors for you:

```
10 SCNCLR
20 FOR Y=0 TO 15
30 S=1024 : C =55298
40 POKE S+Y,35 : POKE C+Y,Y
50 NEXT Y
```

Each little pound sign is a different color. For changing just the color of certain characters without changing all of them, these POKEs can come in handy. Also, using an inverse space (160), it is possible to create low resolution lines. The following program draws a red line across the top of the screen.

```
10 SCNCLR
20 FOR X=0 TO 39
30 POKE DEC("D800") + X,2
40 POKE DEC("400")+X,160
50 NEXT X
```

For a more interesting effect, the following program runs the gamut of colors and characters.

```
10 SCNCLR
20 BS=DEC("400") : BC=DEC("D800")
30 FOR X=0 TO 999
40 C=C+1 : IF C > 255 THEN C=1
50 POKE X+BS,C : POKE X+BC,C
60 NEXT
```

Accessing Machine Language Subroutines

The SYS command can be a useful tool in speeding up your programs. A SYS command "runs" a machine-level subroutine in your computer's ROMs or in memory. In the COMMODORE 128 PROGRAMMER'S REFERENCE GUIDE there is a listing of your computer's ROM. In order to give you a simple and quick reference, we'll put some useful SYS locations in a little chart for you. You can access these subroutines using either hex or decimal values. If you use hex numbers, use the following format:

SYS DEC("Hex numbers" *or* string$)

Remember, all addresses that are POKEd, PEEKed or SYSed must be in **decimal.**

=CHART IT!=

In addition to having labels stuck all over my computer, I have a number of charts. (For labels, I use the tape kind made with a label gun.) The nice thing about a chart is that it has everything from a single category together in one place. You should make or buy or somehow get your hands on charts that will summarize SYS's, POKEs, and other handy locations and addresses. Also, in several computer magazines, you can find charts. Make copies of the charts and using rubber cement, paste them to cardboard and keep them handy. RUN Magazine's special Annual Edition has a great chart in it.

Some SYSes for your SYS collection.

`SYS 49152 ($C000)` Works like SCNCLR
`SYS 49153 ($C001)` Don't do this with young children in the room.
`SYS 65357 ($FF4D)` Reconfigure to C-64. (Quicker than GO64.)

Try some of the above SYS's in your programs to see their effect. Here is a program to play with SYS. (BE SURE TO DSAVE THE FOLLOWING PROGRAM BEFORE YOU RUN IT!!)

```
10 SYS 49152
20 INPUT "ENTER THE PASSWORD";PW$
30 IF PW$ <> "SHAZAAM" THEN GOTO 100
40 PRINT "YOU GOT IN THE PROGRAM"
50 END
100 REM *******
110 REM GET 'EM
120 REM *******
130 FOR X=0 TO 999
140 POKE DEC("400") + X,R
150 POKE DEC("D800")+X,R
160 R=R+1 : IF R > 255 THEN R=0
170 NEXT X
180 SYS 49153
```

That'll fix whomever tries to get into your programs.

Summary

This chapter has taken us beyond the range of more elementary programming, but we only tested the waters. The reason for including this chapter in a beginner's book is to give you greater flexibility and useful applications. You're not expected to use CHR$, POKE, PEEK and SYS to any great extent to begin with, but they are important to know a little about to use on a limited basis. Even if you did not understand everything we covered, you have the foundations for expanding your knowledge later.

In the most simple, fundamental sense, CHR$ and ASCII are "direct" codes to the mind of your Commodore 128. Using keys and alphanumeric characters is more sensible most of the time, but sometimes it is actually easier to know what is going on in a program using CHR$ functions. Control-key functions are not as clear as CHR$ functions in a program, for example. Similarly, it is possible to access certain characters, such as quote marks, only with the CHR$ function.

POKEs and PEEKs are not as vital on the Commodore 128 as they were on the Commodore 64. For graphics and sound on the Commodore 64, just about everything had to be PEEKed and POKEd, but as we will see in the next few chapters, new BASIC 7.0 statements make it easy create sound and graphics without a lot of PEEKs and POKEs. However, there will be some situations where you may find them useful. Later, when you get into more advanced programming, you will use PEEKs and POKEs, along with SYS a great deal. For the time being, just think of them as tools for entering into the realm of machine language and sticking things on the screen.

Sound and Music

The New Wave

For the oldtimers among you, the powerful new sound and music commands on the Commodore 128 will make the creation of sound and music programs much easier. Therefore, if you are moving up from a Commodore 64, **forget** everything you know about POKEing and PEEKing in sound and music. We'll be going through the various new BASIC 7.0 statements and commands for generating music and sound one at a time. We'll then see about creating something fun and useful with these new keywords.

Six, Sound and SID

There are **six** statements to be used in **sound**-making programs that access the Sound Interface Device **(SID)** on your computer. The six statements include:

```
SOUND    ENVELOPE
VOL      TEMPO
PLAY     FILTER
```

If you're not a music buff or a sound expert, a lot of the features available on the Commodore 128 might be over your head. Don't be too concerned about the minute detail of the sound generating capabilities of your machine. We're going to concentrate on creating a program that will make it easy for you to make sounds and experiment with sound so that your computer will tell you what's going on. You *System Guide* has a lot of the technical details you will need. We're going to concentrate on the programming details to set up a logical-experimental environment where your computer will tell you what's going on.

SOUND

The SOUND statement has eight parameters, each separated by a comma. The following is a shorthand lits of those parameters;

SOUND,

1. Voice (1-3)
2. Frequency (0-65535)
3. Duration (1=1/60 Second)
4. Direction (0-2)
5. Minimum Frequency (0-65535)
6. Step Value (0-32767)
7. Wave Form (0-3)
8. Pulse Width (0-4095)

To get a quick orientation, we'll just use the first three parameters. Also, before we continue, you will need a statement to turn up the sound on your TV. The VOL (for volume) statement has a range from 0-15, with the lower range being silence. Be sure to include it in your SOUND programs or the sounds you'll hear are the sounds of silence. Enter the following Sound Tester program to get a feel of what each parameter does:

Sound Tester

```
10 SCNCLR
20 VOL 15
30 INPUT "VOICE (1-3) ";V%
40 INPUT "FREQUENCY (0-65535) ";F
50 INPUT "DURATION (1-1/60TH SEC)";DUR%
60 PRINT
70 PRINT "YOUR SOUND STATEMENT IS" : PRINT
```

```
80 PRINT CHR$(18) "SOUND" V% "," F "," DUR%
90 REM IN LINE 80 CHR$(18) MAKES IT INVERSE
100 SOUND V%,F,DUR%
110 PRINT : PRINT "ANOTHER(Y/N)";
120 GETKEY A$
130 IF A$="Y" THEN 10 : ELSE END
```

As you experiment with that program, you will find that the lower the frequency value, the lower the sound tone. Thus, higher notes are made with higher frequencies and *vice versa.* Likewise, the higher the duration value, the longer the sound was sustained. By printing the final SOUND statement to the screen, you can write down those you may want to incorporate in a program. For example, a certain sound may be appropriate to prompt the user to make an input choice in a program or indicate a mistake has been made. The following little program shows how this can be done:

```
10 SCNCLR
20 INPUT "ENTER A VALUE OF LESS THAN 100";N
30 IF N > 99 THEN GOSUB 100
40 IF N < 100 THEN PRINT "YOUR NUMBER IS";N
 : ELSE GOTO 20
50 END
100 REM ************
110 REM SOUND PROMPT
120 REM ************
130 VOL 15
140 SOUND 3,1234,12
150 RETURN
```

Enter a value of greater than 100 to see what happens. If you want a softer or harsher reminded, change the values.

The remaining five parameters for SOUND are more advanced, and while they significantly increase the flexibility of the sounds you can create, we will spend most of this chapter with other statements that can be used for creating musical sounds. Therefore, we'll just do a few more things with SOUND before getting into the powerful musical statements in BASIC 7.0.

Sweep. After the duration parameter, there are three sweep parameters;

1. Direction of sweep: Values 0-2.
0= upward sweep;
1=downward sweep
3= osillating sweep

2. Minimum frequency or beginning of sweep (0-65535)

3. Steps of sweep through frequency. (0-32767)

To isolate the effect of the sweep parameters, we'll hold the voice, frequency and duration parameters constant. This next program will do that for you:

```
10 SCNCLR
20 VOL 15
30 INPUT "ENTER SWEEP DIRECTION (0-2)"; SD%
40 INPUT "ENTER MINIMUM SWEEP (0-65535)";SM
50 INPUT "ENTER SWEEP STEP (0-32767)";SS
60 PRINT
70 PRINT CHR$(18) "SOUND 1,3000,30,"; SD%
"," SM "," SS
80 SOUND 1,3000,30,SD%,SM,SS
```

If you play with that enough, you will get a good idea of the incredible versatility you have with SOUND on the Commodore 128. With practice, you can create any sound you want for everything from musical instruments to rocket ships.

Wave Control. For the final two parameters, we'll just add a couple of lines to our last program. Holding voice,frequency and duration constant, we'll vary sweep and the wave values. The four waveforms are:

1. Triangle (0)
2. Sawtooth (1)
3. Variable Pulse (2)
4. White Noise (3)

The pulse width, the last parameter, varies the width of the pulse with variable pulse waveforms. Make the following changes and additions to the last program to test these parameters.

```
52 INPUT "WAVEFORM (0-3)";W%
54 INPUT "PULSE WIDTH (0-4095)";WW%
70 PRINT CHR$(18) "SOUND 1,3000,30,"; SD%
"," SM "," SS "," W% "," WW%
80 SOUND 1,3000,30,SD%,SM,SS,W%,WW%
```

Even if you don't know a lot about sound, the programs will let you experiment until you learn something! Make notes of the sounds you think may be useful in programs or ones that just sound weird. For the more serious and advanced applications, you will have to venture beyond the scope of this book. Look at some books on music theory, music and voice synthesizing and similar topics. Your Commodore 128 has some of the best sound-making capabilities available in a small computer.

Music to Compute By

We're going to make some music programs, and keep them simple. However, you will be able to make very complex music with the Commodore 128. The ENVELOPE statement has ten built-in instruments, but we access them by the PLAY statement. To get going we'll look at the different instruments and how to PLAY different musical notes.

The Play Statement

The PLAY statment has five parameters that can be remembered by the word VO-TUX (pronounced vo-tux) which stands for:

Voice	1-3 (1)
Octave	0-6 (4)
T-insTrument	0-9 (0)
U-loUdness	0-15 (9)
X-filter	0-1 (0)

We've fixed up VO-TUX a little since the TUX letters didn't make any sense, but we couldn't do much about making "filter" into anything with an X.

Making musical notes is done by simply placing the note in quotes after a PLAY statement. From the immediate mode enter the following:

```
    PLAY "C D E F G A B" <Return>
```

That will run throught the notes for you with all of VO-TUX set at the default parameters. Now, let's see about changing instruments. Just select one of the instruments below and place the letter "T" and the instrument number within the quotation marks after the PLAY statement.

Instrument	**T-value**
Piano	0
Accordion	1
Calliope	2
Drum	3
Flute	4
Guitar	5
Harpsicord	6
Organ	7
Trumpet	8
Xylophone	9

Try the following:

```
    PLAY "T8 C D E F G A B" <Return>
```

That will take you through the scales on a trumpet. Experiment with the other instruments as well. Key in the following program to have a handy example of each instrument and see how to make your computer do most of the work for you:

```
    10 FOR E=0 TO 9
    20 READ IN$(E) : NEXT
    30 FOR X=0 TO 9
    40 SCNCLR : PRINT X; IN$(X)
    50 I$ = "T" + STR$(X)
    60 MUSIC$= I$ + "C D E F G A B"
    70 PLAY MUSIC$
    80 NEXT
    100 REM ***********
    110 REM INSTRUMENTS
    120 REM ***********
```

```
130 DATA PIANO,ACCORDION,CALLIOPE,DRUM
140 DATA FLUTE,GUITAR,HARPSICORD
150 DATA ORGAN,TRUMPET,XYLOPHONE
```

To change the way the instrument sounds, you change the values in the ENVELOPE. The ENVELOPE parameters are:

> Instrument, Attack, Decay, Sustain, Release, Waveform, Width

Your *System Guide* shows the default ENVELOPE for each instrument, but by altering the values of ENVELOPE, when your instrument is PLAYed, it will sound different. For example, try the following to test the different piano sounds you can get by changing the decay default value of 9.

```
10 SCNCLR
20 INPUT "DECAY (0-15)";D%
30 ENVELOPE 0,5,D%,0,0,2,1536
40 PLAY "T0 C D E F G A B"
50 PRINT : PRINT "ANOTHER(Y/N)"
60 GETKEY A$
70 IF A$ "Y" THEN 20
```

Experiment with the different instruments and parameters of ENVELOPE by changing lines 20-40. You can even design your own instrument.

If you want to combine instruments, it is necessary to coordinate the elements. Using different voices, it is possible to simulate the sound of orchestrated music. For example, to play a 'C' on both the trumpet and harpsichord, you could define one as voice one and the other as voice 2 and play them together. Enter the following and you will hear a single note made of the two instruments playing together:

```
PLAY "V1 T8 C V2 T6 C" <Return>
```

Now that's a lot of junk to play one crummy note! Let's see how we can use some programming tricks to make music for us.

First of all, we can make strings of our notes and then PLAY the string. For example, the following little program will play the scales:

```
10 S$ = "C D E F G A B"
20 PLAY S$
```

Using concatenation and imagination, you can tie together not only notes but instruments as well. The following program, for instance, uses all three voices and three instruments to play the scales, but it takes a lot less programming than writing in a new voice and instruments before each note.

```
10 V1$ = "V1T0" : REM VOICE 1/PIANO
20 V2$ = "V2T2" : REM VOICE 2/CALLIOPE
30 V3$ = "V3T8" : REM VOICE 3/TRUMPET
40 FOR X=1 TO 7 : READ N$ : REM GET THE
 NOTES
50 TRIO$=V1$+N$+V2$+N$+V3$+N$
60 PLAY TRIO$
70 NEXT
80 DATA C,D,E,F,G,A,B
```

Mix the instruments around to see what makes the most interesting trio.

Commodore Composing

With just a few more parameters, we'll be able to create Commodore 128 music from sheet music. We know the notes, and we can create flats with $'sand sharps with #'s. Note duration is indicated with the first letter of the note ,except for eight notes which uses I instead of E.

Whole
Half
Quarter
eIghth
Sixteenth

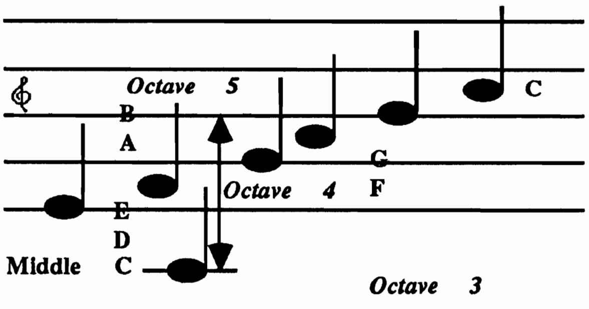
Notes can be held an extra half-beat by a (.) dot. Thus a dotted half note takes the same time as a half and quarter note combined. Here are two simple rules to remember in lining up your notes:

1. Dots (.) go *before* the beat lengths (e.g. - .Q)
2. Sharps and Flats go *before* the notes (e.g. - $ A)

Those rules are important, for in sheet music listings, they re're exactly the opposite. That is, first the note appears and the the dot or flat/sharp.

In addition to the note duration, there's an R for Rest and an M for wait until end of Measure. The R is handled just as the other notes. Thus a QR would rest the beat of a quarter note and a WR would rest the length of a whole note.

Octave. The octave defaults to 4 with the first note in an octave being 'C' (not 'A' as in the alphabet.) On your piano or keyboard, Octave 4 C is known as "middle C." You must pay very close attention to the octave when converting music from sheet to computer, for if you forget to change the octave to the correct level, things get messed up. Look at the following diagram. It will show how Octaves 3-5 are arranged. Middle C is in Octave 4, and the next seven notes *below* middle C are in Octave 3. On the other end of the scale, Octave 5 begins with C in the treble scale.



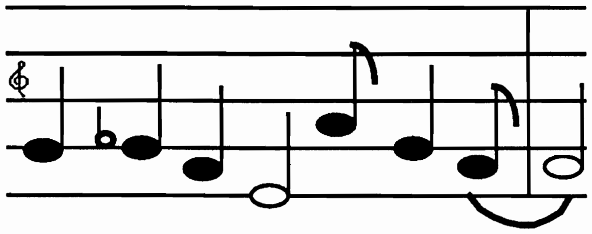
Tempo. About the only other matter we need to cover before writing a song is tempo. The default value for TEMPO is 8 with a range from 1-255. Higher values speed up the beat and lower values slow it down. Once TEMPO is set, it will remain at that speed until RUN/STOP and RESTORE are pressed or another TEMPO statement is issued. Run the following program a few times to get a sense of TEMPO:

```
10 INPUT "TEMPO (1-255)";T%
20 TEMPO T%
30 PLAY "C D E F G A B"
40 PRINT "ANOTHER (Y/N)?"
50 GETKEY A$
60 IF A$ = "Y" THEN 10
```

Song Strings

Finally, we can start making computer music from sheet music. We will use the most simple single-instrument example to get you going, and then you're on your own. We'll break the song down into so that it is easier to debug.

First, translate the notes from sheet to paper in a form that can be used by your computer. The default octave is 4, but no matter what octave your tune begins in, start by indicating the octave. Our song will begin with octave 4, but we'll key it in anyway. We'll use excerpts from "The Birth of the Blues" to show how this is done.



```
O4
QG
Q$G
QF
HE
IA
QG
IHF
```

Starting with the octave, list all of the notes with the correct duration and any flats or sharps. When two notes are connected, just place both of the beat durations next to one another, and then indicate the note. This was done with the last two notes (IHF). Notice how all of this was placed in B1$ in line 140.

```
10 SCNCLR
20 PRINT "BE SURE TO TURN UP THE SOUND"
30 PRINT : PRINT "HIT ANY KEY TO BEGIN"
40 GETKEY A$
100 REM ***********
110 REM ENTER NOTES
120 REM ************
130 B1$="O4 QG Q$G QF HE IA QG IHF"
140 B2$="IB QA IHG O5 IC O4 QB IHA"
150 B3$="O5 ID QC O4 IHB HQB IB IIB O5 IC QD
 HQC"
160 BLUES$=B1$ + B2$ + B3$
170 TEMPO 14
180 PLAY BLUES$
```

You can do the rest of the song yourself. Don't let BLUES$ get much longer. It'd be a good idea to limit the length of a concatenated string to be PLAYed to about 100 characters. Use shorter strings to build a larger one so that it is easy to isolate the notes in case something does not sound right. Also, try altering the TEMPO to see if it sounds right.

Summary and Review

Making sound and music is a lot of fun, and your Commodore 128 makes it very easy to do. Using the powerful words in BASIC 7.0, we began by examining the many different parameters that go to make up any sound. The SOUND statement gathers in the various parameters to give your computer a wide range of sound producing capabilities.

Since it is very difficult to calculate every possible note and instrument, you built-in BASIC 7.0 sound statements have special words and strings to create music. Using a series of special string variables, the PLAY command is allows the user to easily compose music. Using one's own creations or sheet music, the Commodore 128 can replicate individual instruments or whole orchestras. By using the programming tricks you've learned so far, computer coposing is fun, educational and useful.

Using Graphics

Introduction

One of the nicest features of the COMMODORE-128 is its graphics capability. Basically, there are three kinds of graphics: (1) Screen Graphics, (2) Bit Map and (3) Sprite Graphics. Screen graphics are something like text except that we use a lot more color and figures instead of letters and numbers. The way the graphics are used, however, we can access both graphics and text simultaneously. This feature is especially useful for labeling our graphics, such as charts or figures we may wish to create. As a matter of fact, if you have pressed the "Commodore key" and/o shift key and one of the other keys simultaneously, you may already have accessed some of your computer's keyboard graphics capabilities.

Bit mapped graphics take advantage of BASIC 7.0's more powerful set of statements. These statements allow you to draw on your screen. With words like DRAW,CIRCLE and BOX, you can create all kinds of shapes. Furthermore, these words can be integrated with programs to make graphic charts, animation and other interesting and useful graphics.

Sprite graphics are easy with BASIC 7.0 's sprite statements. Using a combination of bit mapped graphics and sprite statements, commands and functions, you can program these colorful and animated graphic characters. Once you become adept at using sprite graphics, there are limitless possibilities for the creation of colorful animated characters.

Screen Graphics

Screen graphics are very simple to use, since you can enter figures directly from the keyboard. To create a single figure all you have to is to PRINT that figure in the same way you would a letter or number. For example, if you,

```
    PRINT "{SHIFT-Z}"
```

you will get a diamond figure. However, to create more interesting graphics, you will want to enter commands from the Program Mode. One way this can be done is to write a series of PRINT statements, entering the drawing as you go along. For example, let's make a graphic playing card. We'll keep it simple and program a two of spades. (It would be a good idea to SAVE this program to disk or tape, as well as the others in this section. SAVE them under different file names since, even though some will have the identical results, they are programmed differently.)

```
    10 SCNCLR
    20 PRINT "{SHIFT-U} {7 SHIFT-D's} {SHIFT-I}"
    30 PRINT "{SHIFT-G} 2 {SHIFT-A} 5 SPACES
     {SHIFT-H}"
    40 PRINT "{SHIFT-G} 7 SPACES {SHIFT-H}"
    50 PRINT "{SHIFT-G} 3 SPACES {SHIFT-A} 3
     SPACES {SHIFT-H}"
    60 PRINT "{SHIFT-G} 7 SPACES {SHIFT-H}"
    70 PRINT "{SHIFT-G} 3 SPACES {SHIFT-A} 3
     SPACES {SHIFT-H}"
    80 PRINT "{SHIFT-G} 7 SPACES {SHIFT-H}"
    90 PRINT "{SHIFT-G} 5 SPACES 2  {SHIFT-A}
     {SHIFT-H}"
    100 PRINT "{SHIFT-J} {7 SHIFT-F's} {SHIFT-
    K}"
```

When you are finished writing the program you should be able to see a "Two of Spades" on your screen - even before you RUN the program. When you do RUN it, the screen will clear and a "Two of Spades" will appear in the upper left hand corner of your TV/monitor. In the same way, you can draw anything else you want with the different shapes and characters on your keyboard. REMEMBER, to get the figure on the right side of the key, use the SHIFT key, and to print the characters on the left side of the key, use the COMMODORE key. Let's take another look at our "Two of Spades" and see if we can improve the program. First, note that lines 40, 60 and 80 are identical as are lines 50 and 70. Instead of having to re-write those lines, let's use our GOSUB commands, treating the repeated lines as subroutines. Using your editor, change line 40 to line 200 and line 50 to 300. Now, add a colon and RETURN after lines 200 and 300. Now, change lines 40, 60 and 80 to read GOSUB 200, and lines 50 and 70 to read GOSUB 300. Add line 110 END. The program should now look as follows:

```
10 SCNCLR
20 PRINT "{SHIFT-U} {7 SHIFT-D's} {SHIFT-I}"
30 PRINT "{SHIFT-G} 2 {SHIFT-A} 5 SPACES
 {SHIFT-H}"
40 GOSUB 200
50 GOSUB 300
60 GOSUB 200
70 GOSUB 300
80 GOSUB 200
90 PRINT "{SHIFT-G} 5 SPACES 2  {SHIFT-A}
 {SHIFT-H}"
100 PRINT "{SHIFT-J} (7 SHIFT-F's) {SHIFT-
K}"
110 END
200 REM **********
210 REM CARD SIDES
220 REM **********
230 PRINT "{SHIFT-G} 7 SPACES {SHIFT-H}"
240 RETURN
300 REM ******
310 REM SPADES
320 REM ******
330 PRINT "{SHIFT-G} 3 SPACES {SHIFT-A} 3
 SPACES {SHIFT-H}"
340 RETURN
```

Now that didn't save a lot of programming time, but if you begin to think of screen graphics as you would any other program, you will want to look for shortcuts to save both memory space and minimize programming redundancy. Now, to see how easy it is to change the two of spades to a three of hearts, using your editor, change the 2 of spades in lines 30 and 90 to a three of hearts - 3 (SHIFT-S), and the spade in line 300 to a heart. Now change line 60 from GOSUB 200 to GOSUB 300. This time when you RUN the program, you have an entirely different card and all you had to do was to make a few changes. Try out different suits, and see if you can make an entire deck.

=EDIT IT!!=

If you did not use your editor to change the above lines, you are working too hard! All that is required when you edit a line is to enter the changes and hit RETURN. To change a line to a different line number, simply enter the new line number over the old line number. For example, to change line 40 in our original "Two of Spades" to line 200, simply use the cursor key to walk up to line 40, place the cursor over the "4", enter "200" and press RETURN. When you LIST the program, line 40 will still be there in its original form, but there will now be a line 200 identical to line 40.

Coloring Your Graphics

If all of the graphics we did were in the two shades of blue on your screen, it would be pretty dull. However, if you do not have a color TV or monitor, the colors will appear as different shades of black and white or green (if you have a green screen monitor). The different color patterns will create different density in the lines and figures you create. If you have something other than a color TV or monitor, it is best to experiment with white {using CHR$(5) or CTRL-2} until you get used to the commands. Later when you get used to the line patterns created on a non-color screen, you can mix them for different effects. Assuming you have a color screen, it might be necessary to adjust your TV/monitor to get the proper colors. One way we can do that is to make a color test chart program. The following program uses only half of your COMMODORE-128's range of colors, but that is because we can only access half using the keyboard or CHR$ commands.

We'll get to the second half of your colors in a bit, but for now, we'll make our color chart so that you can adjust your TV set or monitor.

```
10 SCNCLR
20 FOR X=1 TO 8
30 READ C(X) : NEXT X
40 DATA 5,28,30,31,144,156,158,159
50 FOR L=1 TO 40 : L$=L$+" " : NEXT L
60 FOR BAR=1 TO 8
70 PRINT CHR$(C(BAR))
80 PRINT CHR$(18)L$
90 NEXT B
100 C$="CHR$ COLOR CHART"
110 PRINT SPC(20-LEN(C$)/2)C$
```

Run the program and adjust your set. Once that's done, we can begin doing more with different colors.

=Back to Normal=

Since we will be changing the screen to all kinds of colors, remember, all you have to do to get it back to normal is to press the RUN/STOP and RESTORE keys simultaneously.

Let's go back to our "Two of Spades" program. Since spades are black, let's turn our card from light blue to black. To do that, LOAD your "Two of Spades" program into memory and add the following line:

```
15 PRINT CHR$(18); CHR$(144)
```

That was easy. Do the same with the "Three of Hearts" program, but instead of using CHR$(144) use CHR$(28) for red. Play with the different colors for a while to see what you ge. The following chart shows the color and its associated CHR$ value.

Color	CHR$ Value
White	5
Red	28
Green	30
Blue	31
Black	144

Purple	156
Yellow	158
Cyan	159

Now let's make a simple bar graph using a combination of screen graphics and a little text at the bottom of the screen.

```
10 SCNCLR
20 INPUT "TITLE OF PLOT"; T$
30 PRINT : PRINT : INPUT "HOW MANY PLOTS (1-
7)"; P% : IF P% > 7 THEN 10
40 FOR C = 1 TO P% : READ C(C) : NEXT C
50 FOR I = 1 TO P%
60 PRINT "VALUE OF PLOT#" ; I ; : INPUT "(1-
20)"; P(I) : IF P(I) > 20 THEN 60
70 NEXT I
100 REM ******
110 REM OUTPUT
120 REM ******
130 SCNCLR : S=4
140 FOR I = 1 TO P%
150 PRINT CHR$(19)
160 FOR V = 0 TO (20 - P(I)) : PRINT : NEXT
 V
170 FOR PT = 1 TO P(I)
180 PRINT CHR$(18) ; CHR$(C(I)); SPC(S) ;
 CHR$(32) ; CHR$(32) : NEXT PT
190 PRINT CHR$(5) ; TAB(S) ; I : S=S+4
200 NEXT I : PRINT CHR$(28); : FOR LN = 1 TO
 39 : PRINT CHR$(100); : NEXT LN
210 L = 20 - LEN(T$)/2 : PRINT CHR$(5);
SPC(L); T$
220 GETKEY A$
300 REM *********
310 REM COLOR DATA
320 REM **********
330 REM CHR$ COLOR CODES EXCEPT FOR BLACK
340 DATA 5, 28, 30, 31, 156, 158, 159
```

RUN the program and see how nicely you can present data graphically. The program is severely limited in that it only does a maximum of 7 plots and values from 0 to 20. It is simple to change the number of plots above 7. All you have to do is to change the trap value to a higher number, change the number of colors, change the offset (S variable) and make the bars narrower by using 1 CHR$(32) in line 180.

Changing the values to above 20 requires more sophisticated manipulations, however. This is because, 20 represents the maximum length of a vertical plot and still puts in the material at the bottom of the screen. Using a 2 bar plot, we will examine how to enter any range of numbers we want.

```
10  SCNCLR
20 PRINT : PRINT : INPUT "MAX VALUE->";MV
30  N = 1:NN = MV : REM FOR MORE PRECISE
 CALCULATIONS LET N = .1
40  IF NN > 20 THEN N = N + 1:NN = MV / N:
 GOTO 40
50  FOR I = 1 TO 2
60  PRINT "PLOT VALUE"; I; : INPUT PV(I)
70 PV(I) =  INT (PV(I) / N)
80  NEXT I
90  REM *** END INPUT BLOCK ***
100 SCNCLR; : FOR PL = 1 TO 2
110 C$ = CHR$(32) + CHR$(32) + CHR$(32) +
 CHR$(32) : REM MAKING BARS 4 SPACES WIDE
120  PRINT CHR$(19): FOR V=0 TO (20-PV(PL))
: PRINT: NEXT
130  FOR PT = 1 TO PV(PL) : PRINT CHR$(18);
 CHR$(28); SPC(PL * 10); C$ : NEXT PT
140  NEXT PL
150 FOR LN = 1 TO 39 : PRINT CHR$(30);
 CHR$(100); : NEXT
160 PRINT CHR$ (5)
170 PRINT SPC(9); "PLOT 1"; SPC(5); "PLOT
 2";
180 GETKEY A$
```

In order to understand what happened, we will go over the significant lines and explain each.

1. In line 30 the variable NN was defined to equal the maximum value (MV) entered in line 20.

2. In line 40, the crucial line for creating a proportional scale, NN is compared with 20 to find if the maximum value is greater than 20. If it is greater, then the counter variable N is incremented by 1 and NN is re-defined to be the value of MV divided by N and looped back to the beginning of the line for another comparison. As soon as the value of N increases to a point where the maximum value, MV, divided by N is not greater than 20, the loop exits. Whatever the value of N is at that time will be used in the rest of the program to divide any value entered.

For example:
The value of MV is established to be 100. Since 100 is greater than 20, 1 is added to N and 100 is divided by 2 resulting in the value of NN equaling 50. Since 50 is still larger than 20, N is incremented to 3. When MV is divided by 3, the result is 33.33. Again it is larger than 20; so there is another loop. When N is equal to 4, MV divided by N equals 20. This time, when the comparison to 20 is made, it is found that NN is not larger than 20 and so the line is exited and the value of N is established at 4. No matter what value is entered, as long as it does not exceed the maximum value, there will be no errors since all plot values PV (1), etc., will be divided by 4. Since 100 is the maximum value to be entered, 20 is the maximum value that will be charted.

3. Two values for PV (I) are entered in line 60, and in line 70, PV(I) is divided by N. The INT command is introduced to provide an integer (whole) number for charting.

4. In line 110, C$ is defined as the concatenation of 4 spaces, CHR$(32). This is to make our graph bars 4 spaces wide.
5. Lines 120 through 140 chart are plots, very much like was done in our first chart program.

=Precise But Slow=

We incremented N by 1 each time we passed through our test loop in line 40. If we wanted to get a finer value, we could have incremented N by .1 or .01 or even .00001! This would give us a nearer minimum value by which to divide PV(I) and still keep it proportional. However, it would take longer for the loop to find the minimum value of N. Change the program to see the different results in the charts. The smaller the increment, the closer to the top of the chart the maximum value will appear, but the longer the program will take to execute.

COLOR. To make life a lot simpler, BASIC 7.0 has a statement COLOR that colors things for you. With a modification of the program that drew color bars for you, we'll look at all 16 colors using the COLOR statement. First, thought, let's look at the parameters of color:

COLOR What gets colored, Color itself

There are seven color sources (what gets colored) and 16 colors. First let's take a look at some examples in the immediate mode and then look at all the colors in our bars:

Colored	**Number**
Background	0
Foreground	1
Multicolor 1	2
Multicolor 2	3
40-col border	4
Character	5
80-col bkgnd	6

This next program lets you step through the sources and see what they change. Each time the program pauses, hit any key.

```
10 SCNCLR
20 FOR X=0 TO 6
30 COLOR X,X+4
40 PRINT "THIS IS SOURCE";X
50 GETKEY A$
60 NEXT X
100 REM **************
110 REM BACK TO NORMAL
120 REM **************
130 COLOR 5,2 : REM CHARACTER
140 COLOR 0,1 : REM BACKGROUND
150 COLOR 4,14 : REM BORDER
```

Now let's take a look at all 16 colors for your Commodore 128. Depending on whether you used 40 or 80 columns, your results will be different. (We have been sticking with 40 columns since more users have TV sets, and color monitor users can get 40 columns as well.) The following shows the numbers associated with different colors:

Color(40)	Number	Color(80)
Black	1	Black
White	2	White

Red	3	Dark Red
Cyan	4	Lt. Cyan
Purple	5	Lt. Purple
Green	6	Dark Green
Blue	7	Dark Blue
Yellow	8	Lt. Yellow
Orange	9	Dark Purple
Brown	10	Dark Yellow
Lt. Red	11	Lt. Red
Dark Gray	12	Dark Cyan
Med Gray	13	Med. Gray
Lt. Green	14	Lt. Green
Lt. Blue	15	Lt. Blue
Lt. Gray	16	Lt. Gray

Using your editor, modify the CHR$ Color Chart program using COLOR so it looks like the following:

```
10 SCNCLR
20 FOR L=1 TO 40 : L$=L$+" " : NEXT L
30 FOR BAR=1 TO 16
40 COLOR 5,BAR
50 PRINT CHR$(18)L$
60 NEXT BAR
70 C$="COLOR CHART 16"
80 PRINT SPC(20-LEN(C$)/2)C$
```

Animation

We have spent a good deal of time working on charts in screen graphics, but it is important to see the practical applications of such graphics. Often users simply see screen graphics as something to draw mosaic pictures on and nothing else but, as we have seen, it is possible to make very good practical use of them as well. Now let's have a little fun with animation before going on to POKEing graphics in the screen. Animation in screen graphics can be used in games and for special effects. However, we will only touch upon some elementary examples to provide you with the concepts of how animation works. Basically, by placing a figure on the screen, covering it up and then putting it in a new position, you can create the illusion of moving figures. It works in exactly the same way as animated cartoons. A series of frames are flashed on the screen sequentially. Even though each individual frame has a stationary figure, by rapidly flashing a series of such frames, the figures appear to move. Your computer does the same thing. For example, the following little program appears to bounce a ball in the upper left hand corner:

```
10 SCNCLR
20 PRINT "{SHIFT-Q} " : REM SPACE BETWEEN
 SHIFT-Q AND SECOND QUOTATION MARK
30 FOR I = 1 TO 100 : NEXT
40 PRINT CHR$ (19) : PRINT " '{SHIFT-Q}" :REM
SPACE BETWEEN FIRST QUOTATION MARK
AND SHIFT-Q
50 FOR I = 1 TO 100 : NEXT
60 PRINT CHR$(19) : GOTO 20
```

What appeared to be a moving "ball", was actually a figure being placed on the screen, erased, and then replaced in a different location. Now, let's do the same thing on the vertical axis and use cursor movement within our program. Also, just for fun, let's add some sound and special effects.

```
10 SCNCLR : REM ** BEGIN ANIMATION BLOCK **
20 FOR I = 1 TO 12
30 PRINT TAB(20); "{SHIFT-Q} {UP-CURSOR}" :
REM A WHITE AND INVERSE BALL WILL APPEAR ON
 YOUR SCREEN
40 FOR J = 1 TO 50 : NEXT J : REM DELAY LOOP
 TO SLOW MOVEMENT
50 PRINT TAB(20); "{SPACE}"  : REM PUTS
 SPACE WHERE BALL WAS
60 PRINT : REM FORCE DISPLAY DOWN ONE LINE
70 NEXT I
80 GOSUB 200
90 PRINT TAB(20); "*" :
REM *** END ANIMATION BLOCK ***
100 END
200 REM *************
210 REM SOUND EFFECTS
220 REM *************
230 SOUND 3,90,10,0,2000,3000,3,3
240 RETURN
```

By experimenting with different algorithms, you can create a wide range of effects. If you have played arcade games with movement and sound, you now have an idea of how they were created. Now, go ahead and start working on that SUPER SPACE BLASTER ALIEN EATER game.

POKEing In Color

This section shows you how to POKE in individual color and characters onto your screen. You can skip this section if you want since you can do everything using graphic statements; however, it will help you learn a lot about your 40 column screen memory and color maps.

First, we will see how to change the background color and border of our screen using POKE instead of the COLOR statement, and then we will examine the screen and color memory maps in the COMMODORE-128 to put anything anywhere in any color on our screen. To begin, let's go back to our "Two of Spades" program. Now, we already noted that the card should be black instead of light blue, but every card player also knows that "green felt" is the correct background for the "table." To change the background and border colors, we use the following POKES:

`POKE 53281, (0-15)`	Background Color
`POKE 53280, (0-15)`	Border Color

Load your "Two of Spades" into memory, press CTRL-1 to change the drawing color to black, and now POKE 53281,5. When you RUN the program now, you will have a black card on a green background. Since the borders of card tables are made of wood, let's change the border to brown with a POKE 53280,9. There's our black two of spades on what looks more like a card table! (To get everything back to normal, remember just to press RUN/STOP and RESTORE.) It is quite simple to change the colors of the background and borders. The following are the color codes for the 16 colors you can POKE in (Notice they are one off from the COLOR codes).

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

To get used to what's available, the following program gives you a quick trip through the various background and border colors.

```
    10 SCNCLR
    20 BG = 53281 : BD = 53280
    30 INPUT "BACKGROUND COLOR "; B1 : IF B1 >
    15 THEN 30
    40 INPUT "BORDER COLOR"; B2 : IF B2 >
    15 THEN 40
    50 POKE BG,B1 : POKE BD,B2
    60 GOTO 10
```

When you RUN the program, experiment with different text colors as well by pressing CTRL and keys 1 through 8. You will find that certain text colors are more or less clear with certain background colors. (White on white is very difficult to read!) You may have noticed that we were able to change the border and background colors to 16 different hues, but we still can only get 8 colors for our keyboard characters. To access all colors for our keyboard characters, we will have to understand the COMMODORE-128's screen and color memory maps. As you know, your screen is a 40 by 25 matrix. Each element of the matrix is represented by an address in your computer's memory. Your screen's memory map begins at 1024 and ends at 2023, giving you 1000 locations to put something on the screen. By POKEing these locations with different values, you are able to place characters on the screen anywhere you want. Each character has a code, very much like CHR$ codes, except the code numbers are different, and rather than PRINTing them, you POKE them. For example, the code for the letter "A" is "1." If you POKE a location between 1024 and 2023 with "1" an "A" will appear there. For example, clear your screen and POKE 1475,1. In the middle of your screen, a white "A" appears. But note the location of your cursor. It is still near the top of your screen. That is because you did not PRINT the letter at the location of the cursor, but instead you accessed a memory location. To watch these memory locations fill up with "A's" enter the following from the Immediate Mode:

```
    FOR I = 1024 TO 2023 : POKE I, 1 : NEXT
```

Now to see the different codes, key in the following little program that will give you a run-through of the codes in the middle of your screen:

```
10 SCNCLR
20 FOR I = 0 TO 127
30 POKE 1484, I
40 FOR J = 1 TO 200 : NEXT J : NEXT I
```

Now let's take an animated tour or our screen. We'll start with location 1024 and travel with an arrow (code 62) to location 2023.

```
10 SCNCLR
20 BG = 1024 : ES = 2023
30 FOR I = BG TO ES : POKE I, 62 : FOR J = 1
 TO 5 : NEXT J : POKE I, 96
40 FOR K = 1 TO 5 : NEXT K : NEXT I
```

To create animation on our screen, we first POKEd an arrow, gave it a short delay, then POKEd a space (code 96) in the same location, gave it a short delay, and then went on to the next memory location. Using memory screen locations, we have far more power over characters and animation, for we can go from any point to another without having to worry where the cursor is. Let's go back to our bouncing ball, but this time do it with POKEs.

```
10 SCNCLR
20 FOR I = (1024 + 20) TO 1984 STEP 40
30 POKE I,81 : FOR D = 1 TO 10 : NEXT D
40 POKE I,96 : FOR X = 1 TO 10 : NEXT X :
NEXT I
50 FOR I = (1984 + 20) TO 1024 STEP -40
60 POKE I,81 : FOR D = 1 TO 10 : NEXT D
70 POKE I,96 : FOR X = 1 TO 10 : NEXT X :
 NEXT I
80 GOTO 20
```

Note that we started with the top left corner and added an offset of "20" to put the ball in the middle of the screen. Then we reversed the process in line 50 . If you look on your screen memory map in Appendix C, you will see that 1984 is the bottom left corner of your screen. Now let's look at the color memory map. It begins at location 55296 and ends at location 56295. Again, it is 1000 locations, and think of it as an overlay on your screen map. The upper left hand corner of your screen map is 1024 and on your color map it is 55296. First, we will POKE in a character on your screen map, and then a color for that character on your color map.

```
POKE 1024,81 <RETURN>
POKE 55296,8 <RETURN>
```

First the ball appeared, and then it was colored orange, a color you did not have for your characters before now. Now at this juncture, you may be asking yourself,"How in the world am I expected to figure out one of a thousand screen locations, then one of 127 character codes and then superimpose a thousand different color map locations with one of 16 colors on top of the screen map and get it in the correct place?" Actually, it is not as difficult as it sounds, and like everything else having to do with such calculations, let your computer do the work! The following is a step-by-step outline of how to set up a program to do your calculations using variables.

BS = 1024 <-Beginning location of your screen map.
BC = 55296 <-Beginning location of your color map.
CS = XXXX <-Current location (with XXXX being a number from 1024 to 2023) of your character on the screen.
OF = CS - BS <-Your offset based on the difference between your current location and the starting location on the screen map.
CC = BC + OF <-Color map location to place color.
C1 = XX <-Character code for screen character with XX being a value from 0 to 127.
C2 = XX <-Color code with XX being a value from 0 to 15.

Essentially, the way to determine the mutual location for the screen and color map is to have your computer count the number of locations between the beginning of the map and the current location. Since both maps use sequential addresses, the same offset can be used for both maps. The following program uses the above variables and allows you to place characters anywhere you want them.

```
10 SCNCLR
20 BS = 1024 : BC = 55296
30 INPUT "SCREEN LOCATION (1024-2023)" ;CS
40 INPUT "CHARACTER CODE (0-127)"; C1
50 OF = CS - BS : CC = BC + OF
60 INPUT "COLOR CODE (0-15)"; C2
70 POKE CS,C1 : POKE CC,C2
```

```
80 GET A$ : IF A$ = "" THEN 80
90 GOTO 10
```

Play with the program until you get used to the idea of what codes give you different characters and colors in various locations. Once you're finished, try the following program to give you a "Beaded Curtain" and show another way to create effects with color using programmed POKEs.

```
10 SCNCLR
20 BS = 1024 : ES = 2023 : REM BEGINNING AND
 ENDING ADDRESSES OF SCREEN MAP
30 BC = 55296 : EC = 56295
40 FOR I = BS TO ES : POKE I, 81 : NEXT I
50 FOR C = BC TO EC STEP 2 : POKE C,8 :
NEXT C
60 FOR NC = (BC + 1) TO EC STEP 2 : POKE
 NC,4 : NEXT
70 FOR C = BC TO EC STEP 2 : POKE C,5 :
NEXT C
80 FOR NC = (BC + 1) TO EC STEP 2 : POKE
NC,6 : NEXT
90 GETKEY S$
100 SCNCLR
```

Suppose you don't want to have to look up every character you key in. Let's say you want to write your name or a chart heading or anything else simply by using an INPUT statement and the keyboard. Well, you can use the screen map and POKE in characters. To do this, we will have to learn a new command, ASC. The ASC command converts the firsT character of a string to a CHR$ code. For example, the ASCII code for an "A" is 65. If you keyed in

```
PRINT ASC("A")
```

You would get,

```
65
```

Now since the CHR$ values won't do you any good for POKEs, we will have to convert the CHR$ value to a POKE value. If you look at your screen display codes and ASCII and CHR$ codes, you will see that the ASCII alphabet begins at 65 and the screen codes at 1. To convert one to the other, we simply add or subtract 64, depending on which way we want to convert. Since we want to convert ASCII into screen codes, we will subtract 64 from whatever ASC value we determine. For example, key in the following:

```
10 SCNCLR
20 INPUT "ENTER LETTER FROM A-Z: "; A$
30 A = ASC(A$) : A1 = A-64
40 POKE 1024,A1
50 GOTO 20
```

As you saw, every time you keyed in a letter, that letter would appear in white in the upper left hand corner of your screen (location 1024). Now that we know how to get a single letter in a single location, let's see if we can get entire strings on the screen. To do this, we must do the following:

1. Define or INPUT our string

2. Break up our string into individual letters so that we can get the ASCII values for each character. (The ASC command only reads the leftmost character of a string.)

3. Convert the ASCII values into screen display codes.

4. POKE in the codes.

Using our offset of 64, this means that we will have to use our MID$ command to examine each character. However, when we come to a space (ASCII value 32), we will run into trouble since 32 - 64 is a negative number. To fix that, we'll set a trap for spaces and define them with the correct POKE value - which also just happens to be 32! To keep things simple, we'll use the upper left hand corner of our screen to print our strings. (Line 140 acts like a GOTO 10.)

```
10 SCNCLR
20 PRINT : PRINT : INPUT "YOUR NAME:"; NA$
30 BS = 1024 : BC = 55296
40 DIM A (LEN(NA$)) , A1(LEN(NA$)) : REM
DIMENSION OUR ARRAYS IN CASE THE NAME IS
 LONGER THAN 11
50 FOR I = 1 TO LEN (NA$)
60 A(I) = ASC(MID$(NA$,I,1)) : A1(I) =
A(I) - 64 : REM CONVERT FROM ASCII TO
SCREEN CODE
```

```
65 IF ASC(MID$(NA$,I,1)) = 32 THEN A1(I) = 32
67 REM  IF ASC(MID$(NA$,I,1)) = 46 THEN
A1(I) = 46
70 NEXT I
80 FOR P = 1 TO LEN(NA$) : POKE BS + P,
 A1(P) : NEXT P : REM POKE IN THE CODE
100 REM *** CHART COLOR ***
110 INPUT "COLOR CODE (0-15)"; C2
120 FOR C = 1 TO LEN(NA$) : POKE BC +
 C,C2 : NEXT C
130 GETKEY W$
140 RUN
```

Now if you entered a period (.), you got an ILLEGAL QUANTITY ERROR IN 80. This is because the period, like the space, has an ASCII value of less than 65. If you want to fix the program to accept periods, take a look at line 65 where we trapped spaces. The same was done for periods in line 67. Just remove the REM statement from line 67. That will fix it for you.

Bit Mapped Graphics

Bit mapped or high resolution graphics, allow you to draw graphics on the high resolution screen. The default screen you've been using is the 'text screen', and now you'll move to the 'bit mapped' or 'high resolution' graphic screen. Instead of working with character units, you will be working with little dots of light called "pixels." Your screen is 320 pixels wide and 200 pixels deep. Like your text screen is a 40 x 25 character matrix, your bit map screen is a 320 x 200 pixel matrix. This section examines the statements to use these graphics.

GRAPHIC. To get to your graphic screen, use the GRAPHIC statement. The two parameters for GRAPHIC are the mode and the clear option. Of the six modes, we'll be using Modes 1 and 2, the most. (1=standard bit map, 2=standard bit map with text at bottom.) The second parameter, the clear option is 0 or 1. A 1 clears the graphic screen and a 0 does not. To get started, we'll use Mode 2 so that we can see what we type at the bottom of the screen. Enter the following:

```
    GRAPHIC 2,1
```

You'll see the familiar READY. prompt and the cursor at the bottom of your screen, but try moving the cursor to the top of the screen. It disappears! That's because only the five bottom rows in a split screen display text. The rest of the screen is in the graphics mode.

DRAW. Now that you're in the bit mapped mode, let's take a look at some drawing and see what a pixel is. The DRAW statement allows you to place a pixel (a dot of light) anywhere on your screen except where your have text if you're in GRAPHIC 2. To get started, enter the following:

```
    GRAPHIC 2 <Return>
    DRAW 1,10,10 <Return>
```

You should see a white dot in the upper left corner of your screen. That is a single pixel. Now let's see what we've done with the DRAW parameters.

1= Bit map color source as foreground
10=The horizontal or 'x' position of pixel (0-319)
10=The vertical or 'y' position of pixel (0-199)

Try placing pixels in different screen positions until you can plot one anywhere you want on the screen. Once you can do that, try the following little program.

```
    10 GRAPHIC 2,1
    20  Y=100
    30 FOR X=1 TO 319
    40 DRAW 1,X,Y
    50 NEXT
```

Before you run the program, see if you can guess what it will do. Once you're finished, try this next one.

```
    10 GRAPHIC 2,1
    20 DRAW 1,0,100 TO 319,100
```

By placing the word TO after the X/Y coordinates and adding a second set of coordinates, you can use DRAW to draw a line. It's a lot faster too. See if you can draw a vertical line and a horizontal line with DRAW..TO.

BOX. Before you start drawing things with DRAW, we'd better take a look at a few of other words that will help out. First, there's BOX that makes (would you believe?) boxes and rectangles. It works just like DRAW, except you provide the opposite coordinates of the box or rectangle you wish to draw. Try this.

```
10 GRAPHIC 2,1
20 BOX 1,10,10,100,100
```

Practice make some different shapes on your own as you did with DRAW, and then try the following program:

```
10 GRAPHIC 1,1
20 FOR X=10 TO 100 STEP 5
30 BOX 1,10,10,X,X
40 NEXT X
```

Play with BOX and programs like the one above to see the designs you can make. Remember to use variable parameters for graphics just as you would use variables in any other kind of non-graphics program. Also, try changing colors in your BOX drawings as illustrated in the following:

```
10 REM****
20 REM BOX
30 REM****
40 GRAPHIC1,1
50 COLOR1,9 :REM CHANGE FOREGROUND
60 BOX1,20,30,70,130,,1
70 COLOR1,14:REM CHANGE FOREGROUND
80 BOX1,80,100,130,130,,1
90 COLOR1,5 :REM CHANGE FOREGROUND
100 BOX1,140,50,190,130,,1
110 COLOR1,8 :REM CHANGE FOREGROUND
120 BOX1,200,40,250,130,,1
```

CIRCLE. Like BOX draws boxes, CIRCLE draws circles. However, CIRCLE does a lot more. It can draw ovals (ellipses), arcs and even octagons and triangles. To get started, let's look at the minimal parameters of CIRCLE.

CIRCLE COLOR SOURCE, X,Y,RADIUS

The next little program draws a white circle centered at X =160 and Y =100 with a radius of 50. That puts it in the center of the screen. The 'circle' is a bit elliptical.

```
10 GRAPHIC 2,1
20 COLOR 1,2
30 CIRCLE 1,160,100,50
```

If you add another number after the radius, it will treat it as the Y radius. The first radius is treated as the X radius For example add 20 to the parameter list in the above program.

```
30 CIRCLE 1,160,100,50,20
```

The second time you run the program, you get an ellipse, flattened out by the small Y radius. Change the 20 to a 40 to get a truer looking circle compared with the default circle.

Now to see an arc, we add two more parameters.

CIRCLE CS, X,Y,RX,RY,Begin Arc,End Arc

The default beginning of an arc is 0 and the end 360. To visualize where the arc begins, let's make a half circle. Change line 30 to the following:

```
30 CIRCLE 1,160,100,50,40,0,180
```

When you RUN the program, you can see where the default beginning is. Change the zero (0) in line 30 to a 90 and see what happens. Instead of starting at the 0 position, it begins at 90 degrees. Change the starting and ending positions in line 30 and experiment until you can get any arc you want.

Now we want to look at the rotation of the circle. We'll need yet another parameter for CIRCLE following the beginning and ending arc parameters. While we're at it, we might as well look at the last parameter, the degrees of increment.

CIRCLE CS, X,Y,RX,RY,BA,EA,ROT,INC

First, to see ROTation, change line 30 to the following. (We're making a complete circle so BA=0 and EA=360).

30 CIRCLE 1,160,100,50,40,0,360,50

That will tilt our circle to the left. Change the 50 to a 150 and it will tilt to the right.

Finally, the increment of degrees can be changed. Each segment of the circle is two degrees. However, a smoother circle can be made by placing a one (1) in the last parameter position. To make a less smooth circle and polygons, change the last parameter to a higher number.

=Quick Change=

To get out of the bit map graphic mode and back in the text mode so that you can look at your listing, you're supposed to enter GRAPHIC 0. However, it's a lot quicker to make a ?SYNTAX ERROR and get dropped back into GRAPHIC 0 quickly. A really simple way to do that is to hit the 'equal' key (=) and the RETURN key. Since the two keys are right next to each other, it's a quick and dirty way to get back to GRAPHIC 0.

PAINT. Another graphic word for bit mapped graphics is PAINT. This word fills an area with color. All you do is specify the color source, and the X,Y coordinates in a surrounded area. For example, the following PAINTs a triangle for you.

```
10 REM******
20 REM PAINT
30 REM******
40 GRAPHIC1,1
50 DRAW 1,40,90 TO 270,90
60 DRAW 1,40,90 TO 40,140 TO 270,90
```

To paint with different colors within a given area, you can use the foreground color for one PAINT job and the background color for another. The following breakfast time example shows how. (**Note:** Try changing the parameters on the following program to get different shapes, colors and sizes.)

```
10 REM*******
20 REM CIRCLE
30 REM*******
40 GRAPHIC 1,1
50 COLOR 1,2 :REM WHITE
60 CIRCLE 1,160,120,150,60
70 PAINT 1,240,170
80 COLOR 0,8 :REM DRAWING WITH BACKGROUND
90 CIRCLE 0,160,100,60,20
100 PAINT 0,160,100
```

CHAR. The final graphic word we'll examine in this section is CHAR. This statement allows you to place characters on the bit mapped graphic screen. This is very useful for labelling your graphics. For example, add the following two lines to the end of the last program above:

```
120 REM PUT THE WORD 'EGG' AT THE TOP OF THE
 SCREEN
130 CHAR 0,18,3,"EGG",0
```

The first three parameters are the color source and X,Y coordinates of the beginning of the text. This is followed by the string you want printed and finally a 0 or 1 for inverse or regular printing.

That concludes the section on bit-mapped graphics. However, don't forget what you've learned so far because we'll be using a number of these statements in dealing with sprites and in the last chapter where we'll look at some more sophisticated tricks with this stuff.

Sprite Graphics

Sprites are little characters that you can easily animate on your screen. They are used to create arcade-like games and other applications where mobile 'translucent' characters are required. The ability to move over other text and graphics without disturbing them, give you a lot of flexibility. They've been used for everything from "menu pointers" to space ships. You'll really enjoy them on your Commodore 128.

If you've programmed sprites on the Commodore 64, you can do it the same way on the Commodore 128. However, BASIC 7.0 has some very powerful words that make sprite creation and manipulation much easier than the old way. In fact there are three ways to make sprites on the Commodore 128:

1. Use bit-mapped graphics
2. Use the built-in sprite editor
3. POKE them in (Commodore 64 method)

The third method makes little sense with the new BASIC 7.0 words, and so we won't cover it. Instead we'll concentrate on using the first two. Since we've just covered using bit-mapped graphics, let's make a sprite with them and then crank it up and move it around. We'll make a simple "Sprite Space Fighter" to buzz some planets we can create.

Step 1. Using bit mapped graphics, we'll make a simple little character. You are limited to a 24 x 21 pixel matrix (12 x 21 if multicolor sprites are used.) First, just draw the character you want to make into a sprite.

Step 2. Once you have drawn your character, you save it in a string with with SSHAPE. This word is something like BOX in that you specify the upper left and lower right hand corners of your sprite. For example, if your character is in the upper left corner between 0,0 and 23,20, you would specify,

```
SSHAPE S$,0,0,23,20
```

The variable S$ can be any string variable you choose. Let's get started with out program. Enter the following:

```
10 REM *************
20 REM CREATE SPRITE
30 REM *************
40 GRAPHIC 1,1
50 COLOR 1,2
60 DRAW 1,7,6 TO 7,15
70 DRAW 1,19,6 TO 19,15
80 DRAW 1,7,11 TO 19,11
90 CIRCLE 1,13,10,3,2
100 SSHAPE F$,1,1,24,21
```

At this point, you've created a sprite and gathered it into a string.

Step 3. Now, we save the sprite into a sprite number. This is so easy, we'll save two of them. Using the statement SPRSAV, you save the string into a sprite number. For example, SPRSAV F$,1 saves the sprite defined in F$ as Sprite 1. Look at the next section where F$ is saved as sprites 1 and 2.

```
200 REM ************
210 REM SAVE SPRITES
220 REM ************
230 SPRSAV F$,1
240 SPRSAV F$,2
250 GRAPHIC 1,1
```

Step 4. Next we'll turn on the sprites. This is easy but there are *eight* parameters involved in the statement SPRITE. For the most part, you'll just need the first three which are:

1. Sprite number [1-8]
2. On=1 Off=0
3. Foreground color [1-16]

Look at this next segment to see how our two sprite were turned on.

```
300 REM ***************
310 REM TURN ON SPRITES
320 REM ***************
330 SPRITE 1,1,14 :REM TURN ON SPRITE 1 IN
GREEN
340 SPRITE 2,1,8:REM TURN ON SPRITE 2  IN
YELLOW
```

If you RUN the program at this point, you can see the yellow sprite on your screen. (It's over the green one; so you can't see the green sprite.)

Step 5. All that's left to do is to move the sprites. BASIC 7.0's word, MOVSPR has three parameters to do that:

1. Sprite number [1-8]
2. Angle [0-360]
3. Speed #[0-15]

There's more to MOVSPR parameters than that, but it's enough to get started. Go ahead and blast off with this final segment.

```
400 REM ************
410 REM MOVE SPRITES
420 REM ************
430 MOVSPR 1,70 #4 :REM MOVE SPRITE 1, ANGLE
70, SPEED 4
440 MOVSPR 2,280 #6 :REM MOVE SPRITE 2,
 ANGLE 280, SPEED 6
```

For those of you used to laboriously POKEing in your sprite information, this method must seem incredibly simple. It is! Later, in discussing joystick control, we'll return to this program to examine MOVSPR some more, but for now, let's add one final segment to this program to provide the right setting.

```
500 REM ****************
510 REM ADD SOME PLANETS
520 REM ****************
530 COLOR 2,5
540 COLOR 3,8
550 GRAPHIC3,1
560 CIRCLE 2,120,40,15
570 PAINT 2,134,41
580 CIRCLE 3,10,10,50
590 PAINT 3,10,10
600 COLOR 3,7
610 CIRCLE 3,10,40,70,10,47,200
620 COLOR 3,3
630 CIRCLE 3,10,50,70,10,46,200
```

Now there you have the makings of an arcade game. If there were only some way to control the sprite's movement with a joystick....

JOY. For those of you who want to control your sprites or graphics with a joystick, BASIC 7.0 provides a handy function called JOY. The function JOY returns the value of joystick 1 or 2 with JOY(1) or JOY(2). Plug in your joystick in Port #1 and RUN the following little program:

```
10 JS=JOY(1)
20 PRINT JS;
30 GOTO 10
```

Move your joystick around to see what values are returned. Be sure to press your 'fire' button as well. All values will be between 0-8 until you press the 'fire' button. Then you'll get a 128.

Fortunately, the values JOY returns are in relationship to angles you can MOVSPR your sprites with. With a center or neutral value of 0, the value are arranged around the joystick clockwise from 1 to 8 beginning at the 12 o'clock position. Let's look at these positions in terms of the angles they represent:

Joystick Positions and Angles

```
              1=0°
      8=315°          2=45°
  7=270°        0          3=90°
      6=225°          4=135°
             5=180°
```

By aligning the angles of MOVSPR with the associated joystick values, we can control the direction of the sprites with the joystick. The following subroutine does that for you in your sprite program.

```
400 REM ****************
410 REM JOYSTICK CONTROL
420 REM ****************
430 JS=JOY(1)
440 IF JS>127 THEN GRAPHIC 0 : END
450 IF JS=1 THEN SA=0
460 IF JS=2 THEN SA=45
470 IF JS=3 THEN SA=90
480 IF JS=4 THEN SA=135
490 IF JS=5 THEN SA=180
500 IF JS=6 THEN SA=225
510 IF JS=7 THEN SA=270
520 IF JS=8 THEN SA=315
530 MOVSPR 1,SA  #4
535 MOVSPR 2,280 #4
540 GOTO 430
```

Notice how the value of the joystick was stored in the variable JS and how that value was transferred into the variable SA (for 'sprite angle') in lines 450-520.

Line 440 tests for the fire button being pressed. If it is, it ends the program and goes back to GRAPHIC 0.

Using the Sprite Editor: SPRDEF

Your built-in sprite editor is very simple to use for creating very fine sprites. In fact, if you build a sprite and run it with the method we've used, you can make detailed changes to them from the editor. To get going, just enter SPRDEF and when the editing window appears type in '1'. Since cyan is a good color to work with, press CONTROL-4. If there's a sprite in memory, it will appear in your window. If there is one there or some garbage, press CLR/HOME and you'll have a clean editing window. Your *System Guide* shows you clearly how to edit sprites, and so we will concentrate on making programs to save them and load them back into memory for use. We'll make a multicolored sprite to use as a double example to 1) use as a sprite which we can work with in the sprite editor and 2) a multicolored sprite.

Multicolor Sprite Creation. First, to set multicolor 1 and 2, use the SPRCOLOR statement. For example, to set light red and green use the following:

```
SPRCOLOR 11,6
```

Multicolor 1 is the first parameter and multicolor 2, the second. These color *cannot* be controlled once you are in the editor. Look up the colors for red and blue and set them for the example we will use.

To make a multicolored sprite get into the editor with the SPRDEF command. (Just type SPRDEF and hit RETURN.) Choose '1' (for sprite #1) and clear the sprite screen with CLR/HOME. Press 'M' so that you have a double cursor. Now you're set to make a multicolored sprite. The '2' key gives you the background color, the '3' key ,multicolor 1 and the '4' key, multicolor 2. Change colors by pressing CONTROL or COMMODORE keys 1-8. We'll make a 3 - colored flag; red, white and blue. Press the 2 key until the top third of the screen is white. The 3 and 4 keys were set with SPRCOLOR.

Press SHIFT-RETURN and then RETURN to exit the editor.

You've just completed a multicolored sprite. Next we'll crank it up, fly it around, save it, turn off the computer and then reload it and fly it some more. First to get it on the screen enter the following:

```
10 GRAPHIC 4,1
20 SPRITE 1,1,7,0,0,0,1
```

Check to make sure that works by RUNning it, and then type in the next line to fly it.

```
30 MOVSPR 1,180 #4
```

That's all there is to it! Now to write some programs to save and load them.

Saving and Loading SPRDEF Sprites

To save a sprite, key in the following program.

```
10 INPUT "SPRITE NAME";SN$
20 BSAVE (SN$),B0,P3584 TO P4096
```

The little program will save all eight sprites stored in addresses 3584 to 4096. The 514 bytes in the addresses between 3584 and 4096 represent eight 64 byte sprites. (Actually, the sprites only use 63 bytes in 3 x 21 byte matrices, but an unused byte is attached.) The beginning addresses for each sprite are as follows:

Dec	Hex	Sprite #
3584	$E00	1
3648	$E40	2
3712	$E80	3
3776	$EC0	4
3840	$F00	5
3904	$F40	6
3968	$F80	7
4032	$FC0	8

To save individual sprites use the sprite's starting address and add 63 to obtain the ending address. For example, to save sprite 2 only, you would enter,

```
BSAVE "SPRITENAME",B0,P3648 TO P3711
```

Usually, it's easier to save a block of sprites and then just use the ones you need when you load them back into memory for use.

To load a sprite back into memory after you've reset or turned off your computer, you need only to BLOAD the file. There is no need to specify addresses. The following little program will do it automatically for you:

```
10 INPUT "SPRITE NAME";SN$
20 BLOAD (SN$)
```

If you have a program that uses different groups of sprites, you can introduce the above routine to automatically load your sprites for you. The following program will either save your sprites created with SPRDEF or load your sprites and put them on "parade for you."

```
10 SCNCLR
20 PRINT "LOAD OR SAVE SPRITE (L/S)";LS$
30 GETKEY A$ : IF A$="L" THEN 110
40 IF A$= "Q" THEN END
50 IF A$<>"S" THEN 20
60 SCNCLR
70 INPUT "SPRITE NAME";SN$
80 BSAVE (SN$),B0,P3584 TO P4096
90 SCNCLR
100 GOTO 10
110 INPUT "SPRITE NAME";SN$
120 BLOAD (SN$)
130 FOR X=1 TO 8
140 SPRITE X,1,X+1
150 MOVSPR X,10*8 #X*+5
160 NEXT
```

Summary

This chapter has covered a lot of material on graphics, and we didn't even examine the full range of Commodore 128 graphic words. Beginning with keyboard graphics, we saw how to do everything from make graphs to create arcade style characters and movement with sprites. The powerful and colorful features of your Commodore 128, make it well suited for a first class graphics computer.

The most important thing you should have learned in this chapter is the interaction between programming and graphics.

All of the programming tricks you've learned up to this point using text can be applied equally well to graphics. Remember that text is simply one way for you to communicate with your computer. The text tells you what was calculated or stored in memory with words. Graphics does the same thing with other kinds of figures you draw on the screen. Every programming trick (and then some) are important to use to create graphics as well. In Chapter 11, there are some more advanced graphic applications which will demonstrate further how to program graphics.

All About Files

Introduction

In this chapter we are going to learn more about some advanced applications with the tape and disk system. We will be covering two types of files: (1) Sequential Files and (2) Relative files. Before beginning, I want to point out that the Commodore 1571 (or 1541) floppy disk system is a very sophisticated and smart device. For beginners, it can be difficult to understand some of its more advanced applications, and there is a very real risk of destroying programs and data on your disk. Therefore, in this section, we will take each step slowly, and even at the risk of redundancy, explain the various functions of commands dealing with your disk system. Also, we will not be dealing with the most advanced features of the disk operating system, for they are beyond the scope of this book. However, we will be going to a "middle" range of sophistication, and it is strongly advised for those of you with a disk system to use a blank formatted disk on which you have *not* accumulated programs. By doing so you will not inadvertently destroy valuable data and programs. (This comes from the voice of experience, having clobbered numerous disks myself!)

Data Files and Your Cassette (Disk users skip this section)

You can save any kind of data, numeric or string, to tape and then using a special set of commands we will learn, load that data directly into your program. The data are saved in sequential files, called 'data' or 'sequential' files. You can create a check book program that saves all of your check entries and balances to tape, make a mailing list that creates, saves and retrieves names, addresses and telephone numbers, or even a list of your favorite recipes. In Chapters 1 and 2 we discussed how to SAVE a program and retrieve it with LOAD on your COMMODORE-128 using the Commodore C2N Cassette Unit. Both of these commands are executed in the Immediate Mode. The commands we will now discuss are executed from the Program Mode, but they too function to load and save information to your tape. They simply do it in a different format. To begin, we will review the different commands for working with data files, and then we will work with some programs employing these commands:

OPEN, INPUT#, PRINT# and **CLOSE**

In order to prepare your cassette for reading or writing information from within a program, the tape file must first be "prepared" with an OPEN statement. The format is as follows:

```
OPEN N,1,(0,1 or 2),"FILE NAME"
```

1) First, "N" can be any integer from 1 to 255 to reference the file. For example, you might want to reference your file with number 21 (but any number between 1 and 255 would do just as well); so you would write:

```
OPEN 21,etc.
```

2) Second, since the device is the cassette recorder, the second number would be "1". Your cassette is always addressed as "1" and your disk as "8."

```
OPEN 21,1,etc.
```

3) Third, your file is prepared for reading with a 0, and writing with a 1 or a 2. If you want to write with an End-Of-File marker, use a "1", and for an End-Of-Tape marker use a "2."

`OPEN 21,1,0,`etc. <-Read a file.
`OPEN 21,1,1,`etc. <-Write a file with an End-Of-File marker.
`OPEN 21,1,2,`etc. <-Write a file with an End-Of-Tape marker.

4) Fourth, provide a reference name for your file. For example, let's say you want to save your check amounts you wrote, called "CHECKS". You would write

```
OPEN21,1,1,"CHECKS"
```

or

```
OPEN21,1,2,"CHECKS"
```

To read that data, you would write,

```
OPEN21,1,0,"CHECKS"
```

It may appear to be somewhat involved, but once you get used to it, it is very simple. At the same it, it is quite flexible as well, since it is possible to open a number of different files simply by giving them different names. But usually, you will want to CLOSE a file before OPENing another. To close a file, all you have to do is enter CLOSE and the file number. In our example, we would enter:

```
CLOSE21
```

So while there is a lot to remember in OPENing a file, there is not much when it comes to CLOSEing one. The next command involves writing data to tape. Using the PRINT# command we can do this. The format for PRINT# is

```
PRINT#,N,D
```

where "N" is the file number and D is the data. For instance, sticking with our example, to print a number or string to tape, we would enter:

```
PRINT#21,etc.
```

If our data were strings, we would enter:

```
PRINT#21,A$
```

or if numeric;

`PRINT#21,A` (or A% for integers)

It is important to remember that PRINT# is not the same as PRINT, and you cannot substitute a question mark (?) as ; you can when using PRINT. That is, if you entered ?#, you'd get an error when you ran the program even though when you LISTed it, it would appear as PRINT#. Just to show you, enter the following:

```
10 ?#1,5
20 PRINT#1,5
```

RUN the program, and you will get ?SYNTAX ERROR IN 10. Now, RUN 20, and you will get

```
?FILE NOT OPEN ERROR IN 20.
```

The format in line 20 is correct, but since the file is not open you get an error. Now LIST the program, and lines 10 and 20 look identical! This is one case where LIST will not help in debugging a program. You must remember to write out PRINT# whenever you use it instead of using the "?" shortcut.

In the same way that PRINT# "prints" data to your tape, INPUT# "inputs" information into your computer from the tape. It has the same format as PRINT# using the OPENed file's number and reads in numeric or string variables.

`INPUT#21,A` (or A% for integer numbers) <- Numbers
`INPUT#21,A$` <-Strings

Finally, we have the GET# statement that is formatted exactly like INPUT#, but like the GET command, it only gets 1 character at a time. However, it can read commas, colons and other characters that INPUT# cannot. It will not be used very much since most applications will want more than a single character, but when you want to read special characters not available with INPUT#, GET# will come in handy. At this point we have commands to open a file, read from or write to a file and close a file. However, before we continue, there is a special variable, ST, that we have to examine. The variable ST is reserved for checking your tape to see if it is finished entering data. If ST = 0, then more data is coming in. The End-Of-File or End-of-Tape marker has not yet been read. Therefore, we can loop back to read more data using ST within an IF/THEN statement. For example, the following format can be used

```
20 INPUT#21,A$
30 PRINT A$
40 IF ST = 0 THEN 20
```

Line 40 checks to see if there is more data, and if there is , it loops back to line 20 to get it. Now that we have seen all of the commands for reading and writing files from and to tape, let's take a look at an application. We might as well use a practical application; so we will make a list of our friends' phone numbers. Whenever we want to call a friend, all we have to do is read the list from tape. First, we must create a list to enter names and same them to tape. After we have done that, we will write a program to retrieve the names and numbers.

```
10 SCNCLR
15 REM **********
20 REM ENTER DATA
25 REM **********
30 INPUT "HOW MANY NAMES TO ENTER"; N%
40 PRINT : DIM NA$(N%), PH$(N%)
50 FOR I = 1 TO N%
60 PRINT "NAME#"; I ; : INPUT" (FIRST
 LAST)"; NA$(I)
70 INPUT "PHONE(XXX-XXXX)"; PH$(I)
80 NEXT I
100 REM ****************
110 REM SAVE DATA TO TAPE
120 REM ****************
130 OPEN1,1,1,"FRIENDS' PHONES"
140 FOR I = 1 TO N%
150 PRINT#1,NA$(I)
160 PRINT#1,PH$(I)
170 NEXT I
```

```
180 CLOSE1
```

To use this program, get a blank tape, and rewind your cassette. RUN the program, and you will be prompted to PRESS RECORD & PLAY ON TAPE when you have entered all the names and numbers you want. As soon as you press the play and record buttons, your screen will go blank and your tape recorder spindles will begin turning. When all the information is saved, the recorder will stop and the screen will reappear indicating that all your data has been saved. (Tape storage is relatively slow compared to disks, so to save time it is suggested to use just a few names (3 or 4) at first.) Now, let's see if everything worked out according to plan. To do that we need a program to read our data, and we will use INPUT# to read the names and numbers. Since both the names and phone numbers were saved as strings, we can do the whole thing with a single string variable we will call DA$ for "DATA STRING." (Remember to rewind your tape before RUNing this program!)

```
10 SCNCLR
20 OPEN1,1,0,"FRIENDS' PHONES"
30 INPUT#1,DA$
40 PRINT DA$
50 IF ST = 0 THEN GOTO 30
60 CLOSE1
```

When you RUN this program, you will be prompted to PRESS PLAY ON TAPE. When you do so, the screen will go blank, and after a bit your text screen will reappear will all the names and phone numbers you entered. At this point you may say, "Now just a minute here! I entered that data as two different string arrays, and this program read only a single string variable! What happened to the arrays and how was it possible to get all that information back with a single variable?" The answer to that question can be seen in how the data is stored and what our program did. While the file was OPENed, we INPUT# whatever data came along. As soon as it was in memory, we PRINTed it with our BASIC PRINT statement, not the PRINT# statement we use to print information to tape. The computer did not care whether the data entered into memory was a name or phone number, only a string, and as soon as that string was in memory it PRINTed to the screen. However, since the screen was blank, we could not see it being printed. In line 50 the program checked to see if there was more information in the file and if there was, it simply picked up and printed the next string, regardless of whether it was a name or phone number. To test this, simply enter PRINT DA$ from the immediate mode, and the last entry will be printed to the screen. Now, let's make our program a little fancier and more useful. If you use this program to store friends' phone numbers, the list will eventually cover more than a single screen. Therefore, you will only be able to see the last screenful of names and phone numbers. What we need is a program to search for and find a specific name and then close the file and print the name and number to the screen as soon as it has been located.

```
10 PRINT CHR$(147)
20 INPUT "NAME TO LOCATE";NA$
30 OPEN1,1,0, "FRIENDS' PHONES"
40 INPUT#1,DA$
50 IF DA$ = NA$ THEN GOTO 100
60 IF ST = 0 THEN GOTO 40
70 PRINT "NAME NOT FOUND"
80 CLOSE1
90 END
100 REM **********************
110 REM PRINT OUT NAME AND NUMBER
120 REM **********************
130 PRINT DA$ : REM PRINT THE NAME FOUND
140 INPUT#1,DA$: REM GET THE NUMBER
150 PRINT DA$ : REM PRINT THE NUMBER
160 CLOSE1
```

Now you have a handy program for storing names and numbers to tape and retrieving a single name and number you want to call. The next problem is updating your file without having to re-enter all of the names. That is, once you have made your phone list, you may want to add new names but you do not want to key in all the names you already have on your list. Can this be done? Yes, but we have to first read all the names into memory from tape and then write them back to tape. There are several ways this can be done, and our example is simply one way. We will do the following:

1. Load all the names and numbers on the tape into an array.
2. Input the new names and numbers on the end of the array.
3. Rewind the tape and resave the old and new data to tape.

```
10 SCNCLR
20 DIM NA$(30), PH$(30)  : REM DIM VALUE
   SHOULD BE NUMBER OF NAMES ON LIST PLUS
   THE NUMBER OF NAMES YOU WISH TO ADD
30 OPEN1,1,0,"FRIENDS' PHONES"
40 N = 0 : REM INITIALIZE COUNTER VARIABLE
50 INPUT#1,NA$(N)
60 INPUT#1,PH$(N)
70 N = N+1
80 IF ST = 0 THEN 50
90 CLOSE1
100 REM **************
110 REM ENTER NEW DATA
120 REM **************
130 INPUT "HOW MANY NEW NAMES";NN
140 FOR I = (N+1) TO (N+NN)
150 INPUT "NAME";NA$(I)
160 INPUT "PHONE";PH$(I)
170 NEXT I
200 REM************************
210 REM COMBINE OLD AND NEW DATA
    AND PUT IT ON TAPE
220 REM ***********************
230 NP = N + NN : REM COMBINED TOTAL
 OF ALL NAMES
240 OPEN 1,1,1,"FRIENDS' PHONES"
250 FOR I = 0 TO NP
260 PRINT#1,NA$(I)
270 PRINT#1,PH$(I)
280 NEXT I
290 CLOSE1
```

Make sure to rewind your tape as soon as all of the old data are loaded. In fact, it would probably be a good idea to insert a couple of lines to remind you. So add,

```
125 PRINT CHR$(147) :  PRINT "REWIND TAPE
 NOW!"
127  PRINT : INPUT "PRESS RETURN  TO
 CONTINUE";RT$
```

That ought to remind you.

Programming the Disk
(Cassettes Users skip the rest of this chapter)

The remainder of this chapter examines programs that do things with your disk system. First, you'll see some programs that automatically run programs for you. These are "Menu" programs. After that, we'll look at sequential and relative files; a real power in your Commodore 128.

You know how to RUN and DLOAD programs from your disk. You can do the same thing from another program. All it takes is for the name of the file to be loaded into a variable name using INPUT, and then the RUN or DLOAD commands followed by the variable name in parenthesis. For example, enter the following little program:

```
10 SCNCLR
20 DIRECTORY
30 INPUT "NAME OF FILE TO RUN "; NF$
40 RUN (NF$)
```

The program did three simple things:

1. Showed the disk contents.
2. Asked for a file name to be entered into the variable NF$
3. Using the RUN command, executed the filename in NF$.

The main thing to remember is to place the variable in parentheses was we did with (NF$).

The following program uses these simple elements, coupled with WINDOW to make a fancy menu program.

```
10 POKE DEC("D020"),0
20 SCNCLR
30 WINDOW 0,0,39,5
40 FOR X=1 TO 40
50 LN$=LN$ + " " : NEXT
60 LN$=CHR$(18) + LN$
70 PRINT LN$
80 PRINT : PRINT " MENU" : PRINT
```

```
90 PRINT LN$
100 REM ****************
110 REM DIRECTORY WINDOW
120 REM ****************
130 WINDOW 0,6,39,24
140 DIRECTORY
150 WINDOW 0,0,39,5
200 REM ***********
210 REM BACK TO TOP
220 REM ***********
230 PRINT CHR$(17)
240 INPUT "WHICH FILE ";NF$
250 WINDOW 0,0,39,24,1
260 RUN (NF$)
```

Sequential Files

If you do not have a disk system, you can skip this section and go on to the next chapter. However, if you are considering purchasing a disk drive for your COMMODORE-128, the following material will be of interest. In many respects storing data on disks is similar to storing it on tape except the storage and retrieval process is much quicker. In fact, all of our examples in the previous section can be operated with the disk system with only a few minor changes in the format. Therefore, to get started, we will see how we can store data to disks using a slightly different format than we did with tape. To do this we will examine the

DOPEN, DCLOSE, INPUT#, PRINT# and **GET#**

commands for disk.

DOPEN. To open a disk channel, we access the device number "8" instead of "1" as we did with the tape. Using DOPEN, though, it is unnecessary to specify which device number unless it is is a second drive. All we need to specify is the logical file number. We will use #9.

```
DOPEN#9
```

Next you need to specify a file name. Whatever you place in quotes will be written to disk as a sequential file.

```
DOPEN#9,"ADDRESS LIST"
```

You can optionally specify a ",P" within the quote marks to write a program file, but we won't be doing that. We need sequential data files. That's all you need preparing to read a file. However, to write to a file, you need one more parameter, 'W'. You do that by placing a ",W" after the file name *outside* the quotes like this,

```
DOPEN#9,"ADDRESS LIST",W
```

PRINT#. Once you have DOPENed a file, you want to put something into it. To do this, you use PRINT# followed by the logical file number. For example, you might want to print "Cinderella"; so you would enter,

```
PRINT#9,"Cinderella"
```

When you PRINT# to your disk, it is just like PRINTing to your screen. You just channel it to your disk instead of the screen. Your can PRINT# USING as well with parallel results on your disk as on your screen.

=All About Buffers=

For years I used buffers but never understood what they were. Once I realized how simple (and important) they were, I used them even more. Whenever something is in RAM in the form of a variable or array element, it is in a 'buffer.' A buffer is simply a temporary storage area used to organize input and output. This is especially important when you're working with your disk system. When you write to your disk, your first place stuff in a buffer, and then move from the buffer to your disk. The opposite is also true. When your read from your disk, you first load the disk information into a buffer. Buffers are really simple and useful. Take one to lunch someday.

Usually before you write something to disk, you place it in an array or variable buffer. Thus, you're more likely to see something like the following:

```
PRINT#9,G$(X)
```

INPUT# & GET#. To get everything back from the disk, you use INPUT# or GET#. INPUT# reads entire strings from the disk and GET# reads one byte at a time. Usually, you'll used INPUT# since your can organize more effectively with it. Often when you do not know what's in a file, or you want to reconstitute it, GET# is used. We'll write a simple file and then read it back with INPUT# and GET# to see the differences.

```
10 SCNCLR
20 INPUT "WHAT'S YOUR NAME ";NA$
30 DOPEN#9,"YOUR NAME"
40 PRINT#9,NA$
50 DCLOSE
```

You should have a small file called, "YOUR NAME" on your disk. (Better check your DIRECTORY.) In that file is whatever you entered when asked 'WHAT'S YOUR NAME?'. Now let's check it out;

```
10 SCNCLR
20 DOPEN#9,"YOUR NAME"
30 DO UNTIL ST : REM NOTE THIS LINE
40 INPUT#9,A$
50 PRINT A$
60 LOOP
70 DCLOSE
```

Line 40 put the contents of the file "YOUR NAME" in the variable A$ (a *very small* buffer) and then A$ was printed to the screen. Think of disk INPUT replacing keyboard INPUT when you use INPUT#.

Now look carefully at line 30. The variable ST is a special reserved variable. It is "activated" when your computer detects an "End of File" marker. When it is not zero, (IF ST <> 0), there's still more stuff for you in the file. Thus when the program hits line 60, it LOOPs back to line 40 to read your disk some more. The following program uses GET# picking up only a byte at a time. You'll see the ASCII values printed next to the letters.

```
10 SCNCLR
20 DOPEN#9,"YOUR NAME"
30 DO UNTIL ST
```

```
40 GET#9,A$
50 PRINT A$; ASC(A$)
60 LOOP
70 DCLOSE
```

Slight changes were made to lines 40 and 50, but the results are a lot different. Now you can better see what's going on. The values after your name are just the ASCII values, but at the bottom is a 13. You may remember that CHR$(13) is a CR or 'carriage return.' That is added to the end of a string you write to disk with PRINT#.

APPEND. Finally, you may want to add something on the end of a sequential file. BASIC 7.0 uses the APPEND command to do that. Instead of using DOPEN, you use APPEND. Since APPEND *only* writes to the disk, you need not specify 'W'; just the logical file number and file name. For example, if you had a file called "SCORE" and wanted to add some more to your data file, you would use APPEND like this,

```
50 APPEND#9,"SCORES"
```

Now to see how all of this goes together, we will create an ADDRESS LIST file with a program called ADDRESS BOOK. (*BE CAREFUL!* Do not use the name ADDRESS LIST as a file name for your program. DSAVE the program as ADDRESS BOOK. The program itself *write* a file called ADDRESS BOOK.) We'll examine the program in "bite sized" (or 'byte sized') chunks and point out what everything does. First, we'll create a menu and the segments to create and append our sequential file.

```
10 SCNCLR
20 RESTORE:CLR
30 FOR I=1 TO 14 : PRINT"*";:NEXT : PRINT
"ADDRESS BOOK"; : FOR I=1 TO 14 : PRINT"*";
:NEXT
40 PRINT : PRINT
50 FOR I=1 TO 5 : PRINT I".":PRINT:NEXT
60 PRINT CHR$(19); : PRINT : PRINT  :FOR I=1
 TO 5 : READ D$ : PRINT SPC(5);D$:PRINT:NEXT
70 DATA CREATE NEW FILE,ADD TO EXISTING FILE
80 DATA READ ALL FILES, READ SINGLE FILE,
 EXIT
```

```
90 PRINT:PRINT"CHOOSE BY NUMBER";
100 GET A:IF A<1 THEN 100
110 SCNCLR
120 ON A GOTO 200,300,400,600,750
200 REM ********************
210 REM *** CREATE A FILE ***
220 REM ********************
230 SCNCLR : PRINT : PRINT
240 INPUT"HOW MANY NAMES";N%
250 DOPEN#9,"ADDRESS LIST",W
260 DIM NA$(N%),AD$(N%),CT$(N%),
SA$(N%),ZI$(N%)
270 FOR I=1 TO N%:GOSUB 800:GOSUB 900: NEXT
280 DCLOSE
290 GOTO 10
300 REM ***************************
310 REM *** ADD TO EXISTING FILE ***
320 REM ***************************
330 SCNCLR :PRINT:PRINT
340 INPUT "NUMBER OF NAMES TO ADD";N%
350 APPEND#9,"ADDRESS LIST"
360 FOR I=1 TO N% : GOSUB 800 : NEXT
370 FOR I=1 TO N%:GOSUB 900 : NEXT
380 DCLOSE
390 GOTO 10
```

In the block beginning with line 200, your program first finds the number of names to be entered and dimensions string arrays in line 260. The array will be the buffer for input from the keyboard. (It is a temporary storage area for what you type in from the keyboard.) The append subroutine (300 block) is almost identical. However, lines 360 and 370 show you a slightly different way of buffering keyboard input to disk output. In lines 270, 360 and 370 there are GOSUBs two subroutines beginning in 800 and 900. Let's take a look at those two subroutines next.

```
800 REM ************************
810 REM *** INPUT SUBROUTINE ***
820 REM ************************
830 PRINT"NAME#";I;
840 INPUT NA$(I)
850 INPUT "ADDRESS";AD$(I)
860 INPUT"CITY";CT$(I)
870 INPUT"STATE";SA$(I)
```

```
880 INPUT"ZIP CODE";ZI$(I)
890 RETURN
900 REM *************************
910 REM *** PRINT# SUBROUTINE ***
920 REM *************************
930 PRINT#9,NA$(I)
940 PRINT#9,AD$(I)
950 PRINT#9,CT$(I)
960 PRINT#9,SA$(I)
970 PRINT#9,ZI$(I)
980 RETURN
```

The 800 block is the input routine for entering all the information into a buffer. Once it is in the buffer, you can write it to the disk with PRINT# routines in the 900 block. Since both the write and append operations use the same INPUT and PRINT# sequences, they were placed in subroutines. It saves a lot of programming time. It is very simple, and you should think of it in term of these simple parts.

Now let's get back to where we left off to look at the subroutines. We were about to examine the block that reads data stored to a disk. The first one reads the entire file and prints it all to the screen. The second block, reads a file until a certain name has been found, and then it writes it to the screen.

```
400 REM ***********************
410 REM *** READ SUBROUTINE ***
420 REM ***********************
430 DOPEN#9,"ADDRESS LIST"
440 DO
450 GOSUB 1000
460 GOSUB 1100
470 LOOP UNTIL ST
480 DCLOSE
490 PRINT:PRINT"HIT ANY KEY TO CONTINUE"
500 GETKEY A$
510 GOTO 10
600 REM *************************
610 REM *** SEARCH SUBROUTINE ***
620 REM *************************
630 SCNCLR
640 INPUT "NAME TO FIND";NS$
```

```
650 DOPEN#9,"ADDRESS LIST"
660 S=1
670 DO WHILE S<>0
680 GOSUB 1000
690 IF NA$=NS$ THEN GOSUB 1100 : S=0
700 LOOP
710 DCLOSE
720 PRINT: PRINT : PRINT : PRINT "HIT
 ANY KEY TO CONTINUE"
730 GETKEY A$
740 GOTO 10
750 END
```

The two different subroutines illustrate two different ways of using a DO/LOOP to check a "flag." A flag is anything that will indicate a given condition. The 400 block uses the reserved variable ST as a flag, and the 600 block uses a variable called 'S' (not reserved). The S variable indicates the name has been found. Both of these flags force a loop exit. The DO/UNTIL loop repeats *until* ST is not zero. The statement UNTIL ST means *until the variable ST is <u>not</u> zero*. (We could have put UNTIL ST <>0 as well.) The DO/WHILE loop repeats *while* S is not zero. Both block jump to the subroutines in blocks 1000 and 1100 for INPUT# from the disk and output to the screen with PRINT. Let's now look at them.

```
1000 REM *************************
1010 REM *** INPUT# SUBROUTINE ***
1020 REM *************************
1030 INPUT#9,NA$
1040 INPUT#9,AD$
1050 INPUT#9,CT$
1060 INPUT#9,SA$
1070 INPUT#9,ZI$
1080 RETURN
1100 REM ************************
1110 REM *** PRINT SUBROUTINE ***
1120 REM ************************
1130 PRINT NA$ : PRINT AD$ : PRINT CT$;",
 ";SA$;" ";ZI$
1140 RETURN
1150 END
```

Now that was a long program! When writing such a program, it is a good idea to save your file about every 10-15 lines so that if you accidentally lose it, you can retrieve most of your work. It is important to note several aspects of this program, including a new command, CLR. The CLR command clears all variables and arrays. That is important in this kind of program since you may want to do different things with it while it is in memory. For example, you may want to add to your address list and then locate a single name. By clearing the variables and arrays every time you go back to the menu, you will not have incorrect values. Another important aspect to note is how the program is blocked into subroutines. Not only does this make it easier to read, but you can save a good deal of programming time by such organization. For example, in both the "READ" and "SEARCH" subroutines, the "INPUT#" subroutine is used. Thus, instead of having to key in the INPUT# commands more than once, the program simply jumps to the single subroutine. In the next chapter we will add a subroutine to print out the names and addresses to a printer, and instead of re-writing the entire program, all it takes is adding on another subroutine! A final item you may have wondered about is using a string array for Zip Codes, ZI$(n). Why didn't we use a real variable? Well, a characteristic of the COMMODORE 128 we noted was its propensity to drop leading "0's" with real and integer variables and arrays. If your Zip Code is 07734, you wouldn't want your computer to say it was "7734." By using a string array, we retain the leading "0."

Relative Files

Rrelative files are like containers of equal sizes in which you store data. If you are familiar with containers in shipping that use standardized boxes, you will have some idea of how relative files work. Basically, you first decide how big a container you will need, based on the maximum size of the material you will be putting in the box. Since all we can put into a relative file is strings, the problem is greatly simplified. Each character in a string takes 1 byte, and there is a 1 byte "overhead" for each file. The overhead is the carriage return character automatically placed to mark the end of the record in 'free form' relative file that we will be using. Therefore, if your maximum length for a given string is 10, it will be necessary to allocate a total of 11 bytes : one for each of the ten characters and one for the overhead. (As you remember, a "byte" is a unit of measurement in the Commodore 128's memory.) All entries into a relative file must be in a string format,including numbers.

For the most part, the process of creating and reading relative files looks very much like sequential files, but there are very important differences. When you DOPEN a relative file, you must include the length of the file. First, as we did with sequential files, we DOPEN the file and place the name of the file in quotes. However, instead of writing the mode, we indicate the file number and the the (L)ength of our file. The following example shows the format for DOPENing a relative file:

```
DOPEN #1,"NAMEFI",L12
```

With this statement we can either write to the disk *or* read from the disk, depending on what else we place in the program. Unlike sequential files, we do not indicate whether the mode is for read or write when we DOPEN a file. Each record in the above example can be a maximum of eleven bytes. (Don't forget the overhead!)

RECORD#. BASIC 7.0 has an important statement, RECORD#, that makes writing relative files much easier than the old way. RECORD# is used to specify the file and record number of a given entry. For example, RECORD#1,3 specifies the next entry using PRINT# will be the third record in a file DOPENed as #1. This feature also allows you to read a specific record rather than having to search through the whole file as with sequential files. For example, using INPUT# or GET# you can specify a certain record to read. In conjunction with DOPEN the following shows the correct sequence and format for RECORD#.

```
10 DOPEN #1,"NAMEFI",L12
20 RECORD#1,5
```

In the above example, line 20 specifies the next read *or* write statement (INPUT# or PRINT#) will go into a file called NAMEFI as record number 5.

Since each record number must be specified before you can write to it, it would seem to be tedious indeed compared with sequential files. However, you can just as well use a variable in place of a single number and loop through several records. For example, the following little relative file will allow you to indicate the number of names you want to put in a relative file and writes the names as seperate records.

Simple Relative Files: Name Writer

```
10 SCNCLR
20 INPUT "HOW MANY NAMES TO ENTER";N%
30 DIM NA$(N%)
40 FOR X=1 TO N%
50 INPUT "NAME PLEASE ";NA$(X)
60 IF LEN(NA$(X)) > 19 THEN 50
70 NEXT X
80 REM ** END OF INPUT BUFFER **
90 REM
100 REM ******************
110 REM WRITE RELATIVE FILE
120 REM ******************
130 DOPEN#1,"NAMEFI",L20
140 FOR X=1 TO N%
150 RECORD#1,X
160 PRINT#1,NA$(X)
170 NEXT X
180 DCLOSE
```

If you have ever tangled with relative files on the Commodore 64, you will appreciate how much easier it is with BASIC 7.0. That little program takes care of everything you need to know to create a relative file. Further one you will learn how to add to relative files and change them. But first, let's read a relative file. We could read all the records, but let's first take advantage of the unique characteristic of relative files; namely, you can read a single file. The following program allows you to read a single record:

Read Single Relative File

```
10 SCNCLR
20 INPUT "WHICH RECORD TO READ";X
30 DOPEN #2,"NAMEFI",L20
40 RECORD#2,X
50 INPUT#2,D$
60 PRINT D$
```

```
70 DCLOSE
80 PRINT
90 PRINT "ANOTHER(Y/N)?"
100 GETKEY A$
110 IF A$="Y" THEN 20
```

Notice that the same set-up format used for writing records is used for reading them. The only thing different is the use of INPUT# in line 50 instead of PRINT#.

Now we're all set to read an entire file. You will remember that ST is a special variable to indicate the end of a file. If you use ST as a flag, your program will shut down at the end of a single record. Instead of ST, the string character 'π' (the pi symbol right next ot the RESTORE key on your keyboard) is a useful flag. A 'π' on your disk means nothing is there, and you can use the 'π' flag to tell your program to DCLOSE the file. Notice how the following read program does this.

Read All Relative Files

```
10 SCNCLR
20 X=1
30 DOPEN#2,"NAMEFI"
40 DO
50 RECORD#2,X
60 INPUT#2,D$
70 X=X+1
80 IF D$ <> "π" THEN PRINT D4
90 LOOP UNTIL D$="π"
100 DCLOSE
```

Before combining this knowlege into a general program, let's take a look at a way to read any file we want. The problem with a "read anything" program is that we also have to specify the file length. This is difficult to remember, for while we can retrieve the file's name from the directory, we cannot do the same with file's length.

Read Any Relative File

```
10 SCNCLR
20 DIRECTORY
30 INPUT "NAME OF FILE TO READ";NF$
```

```
40 INPUT "LENGTH";ML
50 L$ = ",L" + CHR$(ML)  : REM NOTE THIS
 LINE
60 X=1
70 DOPEN#2,(NF$)+L$ : REM NOTE THIS LINE
80 DO
90 RECORD#2,X
100 INPUT#2,D$
110 X=X+1
120 IF D$ <> "π" THEN PRINT D$
130 LOOP UNTIL D$="π"
140 DCLOSE
```

Lines 50 and 70 show the peculiar format to get the length of the file attached to the file name. Notice also that the variable containing the file name must be placed in parentheses.

To illustrate how to use relative files, we will make a program to store clubs and their membership. We will call the file we create CLUB. This program will create a relative file that keeps track of the membership of various clubs. All you need to enter is the club name and the number of members. Since we want only strings, the number of members will have to be changed to strings. Furthermore, since a single record can accept only a single PRINT#, we will concatenate the strings into a single string.

First we will plan the program. We will need to know the number of bytes we can use and choose variable/array names:

CLUB$( )	30 bytes (club name)
M$( )	4 bytes (membership number)
CR	1 byte
D$( )	CLUB$+M$( )
<u>M( )</u>	<u>Not placed in file</u>
Total	35 bytes

There are traps to keep the lengths short enough, and we've included routines to make sure each file has the same length. This is called padding. While it is unnecessary with relative files to pad strings, it is a good practice to do so. Since everything is going to be tied together in a single big string when you write a file, when you take out the contents of a relative file, standard sized parts help rearrange the information into blocked "fields."

Finally, note how we keep an update on the records in the file in the 'CHECK POINTER' block. This ensures that you will not write on top of existing records as you update your club list.

```
10 SCNCLR
20 DIRECTORY
30 INPUT "(N)EW OR (E)ISTING FILE";NE$
40 IF NE$="E" THEN GOSUB 500
50 SCNCLR
60 INPUT "HOW MANY ENTRIES";N%
70 DIM CLUB$(N%),M(N%),M$(N%),D$(N%)
80 FOR X=1 TO N%
90 INPUT "CLUB NAME";CLUB$(X)
100 IF LEN(CLUB$(X)) >30 THEN 90
110 IF LEN(CLUB$(X)) <30 THEN GOSUB 300
120 INPUT "HOW MANY MEMBERS ";M(X)
130 IF M(X) >9999 THEN 120
140 GOSUB 400
150 D$(X)= CLUB$(X)+M$(X)
160 NEXT X
170 DOPEN #2,"CLUB",L35
180 FOR X=1 TO N%
190 RECORD#2,X+PT
200 PRINT#2,D$(X)
210 NEXT
220 DCLOSE
230 END
300 REM ************
310 REM NAME PADDING
320 REM ************
330 IF LEN(CLUB$(X)) < 30 THEN CLUB$(X) =
CLUB$(X) + " " : GOTO 330
340 RETURN
400 REM **************
410 REM NUMBER PADDING
420 REM **************
430 M$= STR$(M(X))  : M$(X)=MID$(M$,2)
440 IF LEN(M$(X)) <4 THEN M$(X)="0" + M$(X)
 : GOTO 440
450 RETURN
500 REM *************
510 REM CHECK POINTER
520 REM *************
530 X=1
```

```
540 DOPEN #2,"CLUB",L35
550 RECORD#2,X
560 INPUT#2,P$
570 IF LEFT$(P$,1) <>"π" THEN X=X+1:GOTO550
580 DCLOSE
590 PT=X-1
600 PRINT "YOU HAVE";PT;"RECORDS"
610 PRINT :PRINT "(HIT ANY KEY)"
620 GETKEY A$
630 SCNCLR : RETURN
```

The conversion of the numeric variable into a string variable in line 430 is worth noting. Since there is an invisible sign in front of number, we want to strip it off. Thus, using MID$, only the string from the second character on is placed into M$(X). This saves a byte, and when the value is re-converted back into a number, the invisible sign will automatically be replaced.

There is a great deal more you can do with files, and this introductory look at them just scratches the surface. To a beginner it might look like a lot, but it is just the beginning. It is possible to make "Data Base" systems to search for individual records, changes individual records, sort records and do all kinds of other things with them.

Summary

In this chapter we learned how to save a lot of time by saving files to tape and disk. Data can be saved to your cassette tape for use later within a program. This is handy since it allows you to enter data at one time and then use it later without having to key in the data all over again. Of course this can be done within a single program using READ and DATA statements, but the user is stuck with that program for using the data. By storing it on tape, it is possible to use it in many different programs. This is especially handy with sprites you have created. Using a disk system, it is possible to store data in sequential files much like saving data to tape. However, disks access the data much faster than tapes, and it is possible to have a single program do several different things with data files on disks. The "File Master" program showed how a single program could be used to create, append, and read a single or multiple files. Care has to be taken to keep everything straight with such a program, but using sequential files increases the power of your computer a great deal. The practical applications of such programs are immense.

10

Working Your Printer

Introduction

By now you should be used to "outputting" information to your screen, cassette or disk. When you write in PRINT "HELLO" you "output" to your screen. When you SAVE, DSAVE or PRINT# something, you "output" to your tape or disk. In the same way that you access your screen, tape or disk, you can access your printer. It is simply another "output" target. However, you cannot DLOAD, INPUT or in some other way get anything from your printer as you can from your keyboard, tape or disk. (How are you going to get the ink off the paper and back into memory?) The procedures for getting material out to your printer and using your printer's special capabilities require certain procedures not yet discussed.

Therefore, while much of what we will examine in this chapter will not be new in terms of the language of commands, it will be new in terms of how to arrange those commands. Also, we will see how we can use the printer in ways that have been done poorly using the screen. For example, no matter how long a program listing is, it can be printed out to the printer, while long listings on the screen scrolled right off the top of the screen into Never-Never land. Likewise, in Chapter 9, we made a handy little program for storing friends' names,addresses and phone numbers. With a printer we can print-out our phone numbers or run off mailing labels with commands that output information to the printer.

There are a lot of printers on the market for computers. However, to keep things simple and to show the maximum use of your COMMODORE-128 with a printer, all examples will be with Commodore's VIC-1515 printer. This printer will provide all graphic and text features you will need, and it is easily interfaced to the COMMODORE-128 system. Besides, it is a very inexpensive printer. If you have another printer and an interface for the COMMODORE-128, then you will have to rely heavily on your printer's manual. Unfortunately, many printer manuals are not very good for beginners since they tend to use highly technical descriptions of how to interface and operate their printers. Therefore, pay special attention to the codes used to turn on or off special features of your printer. This is usually done with a CHR$ command from BASIC, and so, usually all you will have to do is to follow the instructions in this book using the appropriate code from your printer's manual.

=Before You Buy a Printer!!=

The most important aspect in purchasing a printer is making certain it will interface with your COMMODORE-128. Many times over-enthusiastic salespersons will tell buyers all the qualities of a printer and naively believe they can be used on any computer. This is simply not true! In order for a printer to work with a computer, it must have the proper interface, and the best printer in the world will not work with your COMMODORE-128 without such an interface. Therefore, when you buy a printer other than one made specifically for your COMMODORE-128, make sure to buy the proper interface for it. The only certain way to insure the printer works with a COMMODORE-128 is to have it demonstrated with your computer. The VIC-1515,VIC-1525 and MPS 1000 printers by COMMODORE will work with the COMMODORE-128, but otherwise you should have the printer's ability to work with your computer shown to you.

Printing Text on Your Printer

The first thing you will want to do with your printer is to print some text in "hardcopy." ("Hardcopy" is a really impressive term computer people use to talk about print-outs on paper. Use the term and your friends will be amazed.) Like your cassette tape and disk drive, it is necessary to first go through a number of steps to channel information to your printer. Let's review those steps first. OPEN. First, you OPEN a channel to your printer. On the VIC-1515, there is a switch in the back to make the device number 4 or 5. We will use the "4" in our examples, so check to make sure the switch is flipped to "4" before proceeding. (Remember on your disk drive the device number is "8.") The sequence for OPENing the channel to your printer is to enter a number between 1 and 255 (we'll use lucky "7") and the device number,4. Here's how:

```
OPEN7,4
```

Now your printer is ready to receive instructions to "7", the logical file number we used. CMD. The CMD command tells your computer to send output to your printer. You must use the file number (7 in this case) and not the device number (4). For example, enter:

```
CMD7
```

and you will hear your printer "crank" up and print-out READY, as it usually prints to the screen. However, you will notice that it did not print READY to your screen. Now enter,

```
FOR I = 1 TO 10 : PRINT : NEXT
```

and your printer will feed your paper 10 lines, just as it would to the screen had you not directed output to the printer with the CMD command. Until you turn off the output to your printer it will go there instead of your screen.

PRINT#. You will remember that we use PRINT# in programs where we want to print our information to our tape or disk. Well, with your printer the same principle applies also. Let's say that you want to print-out only a few things in a program and do not want everything going to the printer. If you used CMD everything would go to the printer that would normally go to the screen. However, using PRINT# only the information following the PRINT# would. For example, suppose you want to have your screen prompt you with "Name?" and as soon as you enter the name it is printed to your printer, you would want to use PRINT#. The format is

```
   PRINT#7, NA$
```

or

```
   PRINT#7, "CHARLEY HORSE"
```

or

```
   PRINT#7, 12345
```

Let's try a little program to print names to the printer to show how PRINT# can be used in programs where you want to use both the screen and printer.

```
   10 SCNCLR
   20 PRINT : PRINT : PRINT "TURN ON YOUR
    PRINTER"
   30 PRINT : PRINT : PRINT "HIT ANY KEY TO
    CONTINUE"
   40 GETKEY A$
   50 SCNCLR
   60 OPEN7,4
   70 INPUT "NAME TO PRINT";NA$
   80 PRINT#7,NA$
   90 INPUT "ANOTHER(Y/N)";AN$
   100 IF AN$="Y" THEN 70
   110 CLOSE7
   120 END
```

CLOSE. The final command in accessing your printer is CLOSE. As we can see in the above program, it closes the channel to the printer and turns it "off." For the most part CLOSE works pretty much the same way as it does with the tape and disk systems; however, there is an important protocol involved. Before you CLOSE the channel to the printer, you must enter PRINT#{fn} first. Therefore, if you OPEN a channel to the printer, use the CMD command, you must first PRINT# before issuing a CLOSE command. For example,

```
   OPEN7,4
   CMD 7
   PRINT "HELLO HELLO"
```

```
PRINT#7
CLOSE7
```

If you issued the CLOSE command without the PRINT# command, you would run into problems.

Listing Programs

Since listing programs to one's printer is a good way to debug a program or send it to a friend, it would be convenient to have a utility program with which we do just that. So, let's write a program that will list your program to the printer. We will keep it short so that we can tack it on to the beginning of a program. To get started, load a program into memory and then add the following lines:

```
1 OPEN7,4
2 CMD7
3 LIST 5-
4 END
```

When you RUN this program, it "turns on" the printer, LISTs a program from line 5 to the end of the program. When you are finished, enter PRINT#7 and CLOSE7 to close the channel and the file. Unfortunately, you cannot "turn off" the CMD command from within the program using PRINT# and CLOSE as we did using only the PRINT# command. So be sure to turn it off from the immediate mode after a listing.

CHR$ To The Rescue

The secret to using printers is in understanding what their control codes mean and how to use those codes. For example, the following is a partial list of codes provided with the old CENTRONICS 737 printer:

Mnemonic	Decimal	Octal	Hex	Function
ESC,SO	27,14	033,016	1B,0E	Elongated
ESC,DC4	27,20	033,034	1B,13	Select 16.7
ESC,DC1	27,17	033,021	1B,11	Proportional Print

Now, for most first-time computer owners, that could have been written by a visitor from another planet for all the good it does. However, there is important information in those codes and, once you get to know how to use them, it is relatively easy. To get started, forget everything except the "Decimal" and "Function" columns. Now taking the first row, we have decimal codes 27 and 14 to get elongated print. To "tell" your printer you want elongated print you would use CHR$(27) + CHR$(14). To kick that into your printer you would do the following:

```
    1. OPEN7,4
    2. PRINT#7, CHR$(27) + CHR$(14) + "MESSAGE"
```

If you have a Centronics 737 or 739 printer that would have printed the string "MESSAGE" in an elongated print. Likewise, for the condensed printing 16.7 cpi (characters per inch), you would have entered CHR$(27) + CHR$(20) and for the proporitonal type face, CHR$(27) + CHR$(17). Once you get the decimal code, all you have to do is enter that code to the printer and it will do anything from changing the type-face to performing a backspace function. With other printers the same is true, but let's get back to the VIC-1515 printer we have been examining since it was designed with Commodore computers in mind. As we will see, like the Centronics printers or any other, the VIC-1515,VIC-1525 and MS 1000 also use CHR$ commands to access the printer's different abilities. Let's look at the various CHR$ commands associated with the VIC-1515:

CHR$	FUNCTION
10 & 13	Line feed
8	Graphic Mode
14	Double width
15	Back to standard
16	Address of start print position
27	Escape (used in conjunction with other codes)
145	Cursor up
17	Cursor down
18	Reverse printing
146	Turn off reverse

To see how the CHR$ functions work, we will use a simple program that will print-out your name. Since we already know how to print-out normal text, we will begin with expanded text. Looking at our chart, we see that CHR$(14) will expand our print-out; so we will use it in our program. (Notice the lack of punctuation marks after the comma following the PRINT#7.)

```
10 SCNCLR
20 OPEN7,4 : REM OPEN CHANNEL 7 TO DEVICE 4
 (YOUR PRINTER)
30 INPUT "YOUR NAME"; NA$
40 PRINT#7, CHR$(14) NA$ : REM NOTE POSITION
 OF CHR$(14) AFTER PRINT#7
50 PRINT#7, CHR$(15) : REM RETURN TO NORMAL
 PRINT
60 CLOSE7
```

RUN the program and print-out some names and note the expanded characters. (Try that on your typewriter!) Now we have not done very much with upper and lower case so far but in printing text to your printer, there are many times you will want to have upper and lower case characters. For example, in printing out names, you may want your printer to print-out,

```
Captain John W. Smith
```

instead of

```
CAPTAIN JOHN W. SMITH
```

but we need a special CHR$ command to do that. With the VIC-1515 printer the CURSOR DOWN mode, CHR$(17) will allow us to print upper and lower case. To do this, change lines 40 and 50 in our above program to the following:

```
40 PRINT#7, CHR$(17) NA$ : REM PRINTS IN
 CURSOR DOWN MODE
50 PRINT#7, CHR$(145) : REM RETURNS TO
 CURSOR UP MODE
```

Now, press the COMMODORE key and SHIFT key simultaneously to put your computer into the upper/lower case mode and RUN the program. If you used the shift key for upper case and the non-shifted keys for lower case, your printer gave you both upper and lower case. If you tried that before we made the program changes, even if you had your computer in the upper/lower case mode, your print-out would have given you graphic characters for the shifted ones instead of upper case. Go ahead and try it with the original program to see for yourself. Another trick is to use both upper and lower case and the expanded mode together. All that is required is to change the program so that both CHR$

commands are there together. Again, change lines 40 and 50 to the following:

```
40 PRINT#7, CHR$(17) CHR$(14) NA$
50 PRINT#7, CHR$(145) CHR$(15) : REM RETURN
 TO CURSOR UP AND STANDARD PRINT
```

When you RUN this program, you will see that it is possible to have more than a single non-standard (i.e. non-default) mode operating simultaneously. On some printers, such as the EPSON MX-80FT with GRAFTRAX PLUS, it is possible to not only have expanded print but also italicized, condensed, double strike, emphasized and super/subscript type faces and any combination of them together. Using CHR$, all of the different type styles can be used separately or in combination with one another. Finding the "CURSOR UP/DOWN" mode on other printers, however, may be tricky since they were not made specifically for the COMMODORE-128. (Remember to get a demonstration at the store where you buy your printer!) Now that we have seen different ways to operate the type faces on the printer, let's do something practical. We will make a mailing label program for the VIC-1515/1525/MS 1000 printer. Various label manufacturers make adhesive labels with tractor-feed margins so that you can put them into your printer just like your paper. Our program will make labels that will print the addressee's name in expanded upper/lower case, the address in regular upper/lower case, and the city, state and zip code in upper case only.

```
10 SCNCLR : PRINT CHR$(14) : REM SHIFT TO
 UPPER/LOWER CASE
20 OPEN7,4
30 INPUT "NAME"; NA$
40 INPUT "ADDRESS"; AD$
45 PRINT CHR$(142) : REM SHIFT TO UPPER CASE
50 INPUT "CITY"; CT$
60 INPUT "STATE"; SA$
70 INPUT "ZIP CODE" ; ZI$
100 PRINT#7, CHR$(17) CHR$(14) NA$
110 PRINT#7, CHR$(15) CHR$(17) AD$
120 PRINT#7, CHR$(145) CT$ ", " SA$ " " ZI$
130 CLOSE7
```

As you will see when you RUN this program, the screen shifts to the mode your print-out will be in. This helps the user see on the screen what he/she will get on the printer. Note that different CHR$ codes are used to shift the upper/lower case mode on the screen and on the printer. For example, CHR$(142) shifts the screen to upper/lower case while CHR$(17) shifts the printer to that mode.

In order for the program to be more practical, we will need a few line feeds at the end of the printing so that your labels can be properly aligned. Depending on the size of your mailing labels, you will need a greater or fewer number of line feeds. Insert the following line into your program and adjust the size of the loop to align your labels properly.

```
125 FOR I = 1 TO 3 : PRINT#7 : NEXT
127 REM CHANGE "3" TO THE CORRECT
128 REM NUMBER OF LINE FEEDS FOR YOUR LABELS
```

In Chapter 9, we promised to insert a subroutine in the "FILE MASTER" program to print out the names and addresses to your printer. Well, that's just what we're going to do. To make the changes, load your "Address Book" program into memory and make the following additions or changes in the program. (Good grief! Don't rewrite the whole thing!) First, change the loop size in lines 50 and 60 from 5 to 6 so that we can add a printer option to the menu. Now, add,

```
85 DATA PRINTER OUTPUT
```

Next, change line 120 to read:

```
120 ON A GOTO 200,300,400,600,750,1200
```

Finally, add the following subroutine:

```
1200 REM *******************
1210 REM *** PRINTER SUB ***
1220 REM *******************
1230 DOPEN#9,"ADDRESS LIST"
1240 DO
1250 GOSUB 1000
1260 N=N+1 : LOOP UNTIL ST
1270 DCLOSE
1280 DIM NA$(N),AD$(N),CT$(N),SA$(N),ZI$(N)
```

```
1290 DOPEN#9,"ADDRESS LIST"
1300 FOR X=1 TO N
1310 INPUT#9,NA$(X)
1320 INPUT#9,AD$(X)
1330 INPUT#9,CT$(X)
1340 INPUT#9,SA$(X)
1350 INPUT#9,ZI$(X)
1360 NEXT
1370 DCLOSE
1380 REM ** PRINTER ON **
1390 OPEN4,4
1400 FOR X=1 TO N
1410 PRINT#4,NA$(X)
1420 PRINT#4,AD$(X)
1430 PRINT#4,CT$(X)
1440 PRINT#4,SA$(X)
1450 PRINT#4,ZI$(X)
1460 PRINT #4
1470 NEXT
1480 PRINT#4
1490 CLOSE4
1500 GOTO 10
```

Margins

Sometimes you do not want your print-out to begin at the left hand side of your paper or label. To position the starting point of your text, you use CHR$(16) and the number of spaces from the left you wish to begin printing. There are a number of different ways of doing this, but the most simple to do this is to first print CHR$(16) and the starting position in quotes along with what you want printed. You must use a 2 digit number; thus, if you want to begin 5 spaces from the left, you must indicate it with "05" instead of "5." For example, try the following:

```
OPEN7,4
PRINT#7, CHR$(16)"05DOES THIS COMPUTE?"
CLOSE7
```

As you will see when you execute the above commands, your printer will only print out "DOES THIS COMPUTE?" and not the "05" even though it was in quotation marks. That was because the first two digit number encountered after the

CHR$(16) was the "05." Now add another "5" so that the line reads,

```
PRINT#7, CHR$(16)"055DOES THIS COMPUTE?"
```

and you will get, "5DOES THIS COMPUTE." Thus, after the second digit everything is treated as information to be printed out. In some cases you may want to indicate the number of spaces using CHR$ instead of numbers within quotes. For example, you may wish to print text at different positions determined by a loop and want your next position in relation to the last, and so the position is determined by CHR$(I), with "I" being the current loop position. This can be done, but it is tricky because the CHR$ must be the ASCII value for the number. For example, a "05" looks like this:

```
CHR$(0) CHR$(53)
```

Using the above example, you would print,

```
PRINT#7, CHR$(16) CHR$(0) CHR$(53) "DOES
 THIS COMPUTE?"
```

Therefore, if computing the position using a loop, it is necessary to determine the position in terms of both the first and second digit as an offset of the loop value. For most applications, it is best simply to enter the number of positions within the quotation marks of the message to be printed. Before going on to printer graphics, we will examine how to use positioning in a program. This is useful in making lists where columns are important. For example, we can make a list of items for a garage sale. The first column will be the item for sale, the second column the asking price for the item, and the third column the actual price for which the item sold. We will use INPUT statements so that all items can be entered from the keyboard and used with an actual garage sale. (Who knows when you will want to use it? So why not make it useful!)

```
    10 SCNCLR
    20 PRINT : PRINT : INPUT "HOW MANY ITEMS"; N%
: DIM IT$(N%),AP(N%),SP(N%)
    30 PRINT : FOR I = 1 TO N%
    40 PRINT "ITEM #"; : INPUT IT$(I)
    50 INPUT "ASKING PRINCE"; AP(I)
```

```
60 INPUT "SELLING PRICE"; SP(I)
70 PRINT
80 NEXT
100 REM *** PRINTER FORMAT ROUTINE ***
110 OPEN7,4
120 PRINT#7, "ITEM"; CHR$(16) "15ASKING
 PRICE";
130 PRINT#7, CHR$(16) "35SELLING PRICE"
140 PRINT#7, CHR$(10) : REM LINE FEED
150 FOR P = 1 TO N%
160 PRINT#7, IT$(P)
170 PRINT#7, CHR$(16) CHR$(49) CHR$(53) "$";
 AP(P)
180 PRINT#7, CHR$(16) CHR$(51) CHR$(53) "$";
 SP(P)
190 NEXT
200 CLOSE7
```

There are a couple of things to note in this program. First of all, notice how we employed CHR$ code to indicate the positioning of the printed text in lines 170 and 180. The combination of those codes is the same as the "15" and "35" enclosed in quotations in lines 120 and 130. Second, we used the CHR$(10) for a line feed. We could have used PRINT#7 without any code following it to get the same results, but there will be times when you may wish to insert a line feed in the middle of a line and CHR$(10) will come in handy. To really make a neat program, see if you can figure out how to have the program compute the totals of the asking price and selling price of the items. Also, it might be an interesting addition to have a fourth column that keeps a tally of the differences between the asking and selling prices. This is something that you should be able to work out on your own! (Hint: Create a fourth array.)
PRINTING GRAPHICS Now that we have seen how to print text, we will look at graphics printing. The most simple graphics to print are those from the keyboard. Using the CURSOR UP mode, CHR$(145), we can print out the graphics from the keys. For example, from the Immediate mode try the following,

```
OPEN7,4
PRINT#7, CHR$(145) "{COMMODORE-KEY-B}"
CLOSE7
```

That will print out a "checkerboard" character just like the one on the left side of the key. Now, since the default mode is CURSOR UP, it is unnecessary to enter CHR$(145) every time you print a graphic character but, just to be sure, you should have it somewhere in your program. To see all the different graphic characters from your keyboard, run the following program:

```
10 SCNCLR
20 OPEN7,4
30 FOR I = 96 TO 127 : REM CHR$ RANGE OF
 SET #1
40 PRINT#7, CHR$(145) CHR$(I)
50 NEXT I
60 FOR J = 161 TO 191 : REM CHR$ RANGE FOR
 SET #2
70 PRINT#7, CHR$(145) CHR$(J)
80 NEXT J
90 CLOSE7
```

All of the characters on your keyboard were printed out for you, but with patience you could have done the same from the keyboard. The CHR$(145) is a bit superfluous, and you can get the same results if you remove it. However, if CHR$(17) is there, you will have mostly blanks since that is the "upper/lower" case, or CURSOR DOWN mode

Making Your Own Graphic Characters on the Printer

Next we will create binary printer graphics. In using SPRDEF, we used a 24 by 21 matrix to create sprites one dot at a time. Making characters is similar, but it's a lot easier. First of all, we will be using a 7 by 7 matrix instead of a 24 by 21 matrix so there are far fewer calculations. The VIC-1515 printer can actually make graphic characters in 7 by 480 (!) matrix, but we will stick with the 7 by 7 matrix to keep it simple. To get started, instead of sending you off for some graph paper, we will make our own graph for our matrix on the printer, explaining the process as we go along. To begin, we use CHR$(8) to initiate the graphics mode. Then we "build" a concatenated CHR$ that contains our graphic image. Since the bits are added on the basis of the vertical position of each "pin" in the printer head, we will be adding vertical "dots" instead of horizontal ones as we did with sprite graphics. However, we will be using the same concepts as with sprite graphics. The following is an outline of our 7 by 7 matrix:

```
 1      _  _  _  _  _  _  _
 2      _  _  _  _  _  _  _
 4      _  _  _  _  _  _  _
 8      _  _  _  _  _  _  _
16      _  _  _  _  _  _  _
32      _  _  _  _  _  _  _
64      _  _  _  _  _  _  _
+128
```

By inserting "dots" into the blanks, we can create a figure, and this is translated to a way in which the COMMODORE-128 can understand by a vertical total of the positions the dots are in and adding 128. For example, if we draw a square, we would have the first and last columns filled and the top and bottom rows filled. Beginning with the first column, the value would be 1 + 2 + 4 + 8 + 16 + 32 + 64 + 128, equaling 255. The next 5 columns would have a dot at the top and bottom. A dot in the top row would be 1 and one in the bottom row would be 64, and adding the 128 we would get 193. The last column would be the same as the first, 255. Therefore, we would want to create a CHR$ that has the following values:

255 193 193 193 193 193 255

for our box figure. To do this we could have a line that reads as follows:

```
 BOX$ = CHR$(255) + CHR$(193) + CHR$(193) +
CHR$(193) + CHR$(193) + CHR$(193) + CHR$(255)
```

but that (whew!) would take a lot of time. Instead it would be a lot simpler to READ in the values as DATA statements as we did with the sprites and concatenate the string, such as,

```
FOR I = 1 TO 7
READ GRAPHICS
GR$ = GR$ + CHR$(GRAPHICS)
NEXT
DATA 255,193,193,193,193,193,255
```

Now let's put it all together into a program.

```
10 SCNCLR
20 FOR I = 1 TO 7 : READ GRAPHICS
30 GR$ = GR$ + CHR$(GRAPHICS)
40 NEXT
50 OPEN7,4
60 PRINT#7, CHR$(8) GR$
70 CLOSE7
100 DATA 255,193,193,193,193,193,255
```

When you RUN this program, a little box will be printed. Nothing very exciting, I admit, but now let's see how we can use that little box to make a matrix to create new characters. The following program will make a 7 by 7 matrix for you and only requires making a few changes in the above program:

```
10 SCNCLR
20 FOR I = 1 TO 7 : READ GRAPHICS
30 GR$ = GR$ + CHR$(GRAPHICS)
40 NEXT
50 OPEN7,4
52 FOR Y = 1 TO 7
54 FOR X = 1 TO 7
60 PRINT#7, CHR$(8) GR$;
62 NEXT X
64 PRINT#7
66 NEXT Y
70 CLOSE7
100 DATA 255,193,193,193,193,193,255
```

If you printed out the 7 by 7 matrix, you can see that, while it is functional, it really printed more than was necessary. We only need single-sided dividers between the cells. Besides, even though having the single box is handy for making all different kinds of shapes, we might as well create the exact graphics we need. However, if we let the computer do the "figuring" for us, it can be relatively simple. To begin we will break up the task into simple parts. First of all, we know that a straight vertical line is CHR$(255). We will call it E$ since it "encloses" the sides of our box. We also know that CHR$(193) will give us a top and bottom to our box, but if we use it to make a matrix, we will have double lines separating our rows. Therefore, we will only need a top line to begin with. That's easy since the top dot is "1" and all we have to do is add "128" for CHR$(129). We need five of those dots to create our top line, so we will create that with a FOR/NEXT loop of 5. (Remember in our 7 x 7 boxes, the E$ figure will take a top position dot at either end.) Finally, at the end of our matrix we are going to need a bottom line. Here, instead of of drawing a single bottom line, we will draw a bottom line of boxes made up of E$ and CHR$(193), the latter to be designated as TB$ (for "top/bottom"). Therefore, the plan is to first draw 6 lines of boxes with a top only and then, for our seventh line, we will draw a row with both tops and bottoms. It is important to notice that we are now using graphic figures much larger than our 7 x 7 matrix! Here's our improved program:

```
10 SCNCLR
20 E$ = CHR$(255)
30 FOR I = 1 TO 5 : T$ = T$ + CHR$(129)
 : NEXT
40 FOR I = 1 TO 5 : TB$ = TB$ + CHR$(193)
 : NEXT
50 OPEN7,4
60 FOR Y = 1 TO 6
70 FOR X = 1 TO 7
80 PRINT#7, CHR$(8)E$ T$
90 NEXT X
100 PRINT#7, E$ : REM PUTS AN END ENCLOSURE
 ON BOXES
110 NEXT Y
120 FOR X = 1 TO 7
130 PRINT#7, CHR$(8)E$ TB$;
140 NEXT
150 PRINT#7, E$
160 CLOSE7
```

Now that you have a better idea of what can be created, print up a batch of matrixes and design some original printer graphics! You always wanted your own logo, and now you can do it! REPEAT THAT GRAPHIC! The final element we will examine with your printer is the graphic repeat one. Using CHR$(26) it is possible to make any number of graphic characters repeat. However, the format for using repeat requires some care. Use the following steps:

1. Get into the graphics mode with CHR$(8)

2. Issue the repeat command with CHR$(26)

3. Enter the number of repeats within a CHR$ command. Note: This is different than what we saw with the position command. You do not put the ASCII code for the number of repeats, but instead the actual number of times you want a graphic repeated. For example, if you want a graphic to repeat 20 times, you would use CHR$(20).

4. Enter the graphic character, usually followed by the CHR$ code for a semi-colon <CHR$(59)> so that the repetition will occur on the same line. Now let's make a simple program that will give us a "bar" of varying lengths. This will show how you might begin a program that will make a bar graph with bars of different lengths to represent your data.

```
10 SCNCLR : PRINT : PRINT
20 INPUT "LENGTH OF BAR"; N
30 RP$ = CHR$(8) + CHR$(26) + CHR$(N) : REM
 GRAPHICS + REPEAT + NUMBER OF REPEATS
40 VL$ = CHR$(255) + CHR$(59) : REM OUR
VERTICAL LINE PLUS A SEMI-COLON
50 OPEN7,4
60 PRINT#7, RP$ VL$
70 CLOSE7
```

Notice how fast the bar is produced on your printer using the repeat function. Experiment with the command and mix it together with other printer commands to produce anything you want to see in black and white.

Summary

When you got your printer, you may have thought the only thing you could print was text in the same way a typewriter does. However, as we saw, that was just the beginning. Besides printing text, it is possible to generate different style type faces, position the text wherever you want and even print graphics. Not only can you print the graphics from the keyboard, you can also create your own printer graphics. Typewriters just cannot do that! The secret to using printers with your COMMODORE-128 is the CHR$ function. In some ways CHR$'s are used as ASCII code in exactly the same way as they are when output is to the screen, but in other ways they are used either as special printer functions or, within certain sequences, to produce print-outs. Unfortunately, it is not possible to simply access your printer and have it automatically put what's on the screen onto paper. However, by planning your program around output to the printer, just about anything printed to the screen can be printed to your printer.

Program Hints and Help

Introduction

Well, here we are at the last chapter. We've covered most of the statements, functions and commands used for programming in BASIC 7.0 on the Commodore 128 and many tricks of the trade. However, if you are seriously interested in learning more about your computer and using it to its full capacity, there's more to learn. In fact, this last chapter is intended to give you some direction beyond the scope of this book. First, we will introduce you to the best thing since silicone - Commodore 128 Users Groups. These are groups that have interests in maximizing their computer's use. Second, I would like to suggest some periodicals with which you can learn more about your Commodore 128 computer. Third, we will examine some languages other than BASIC that you can use on your Commodore 128. BASIC has many advantages, but like all computer languages it has its limitations, and you should know what else is available. Next, we will examine some more programs. First, there will be listings of programs that you may find useful, fun or both. The ones included were chosen to show you some applications of what we have learned in the previous nine chapters, enhancing what you already know. Then we will look at different types of programs you can purchase. These are programs written by professional programmers to do everything from making your own programming simpler to keeping track of your taxes. Finally, we will examine some hardware peripherals to enhance your Commodore 128.

Commodore User Groups

Of all of the things you can do when you get your Commodore 128, the most helpful, economical, and useful is joining a Commodore 128 User Group. Not only will you meet a great group of people with Commodore 128 computers, but you will learn how to program and generally what to do and not to do with your computer. The club in your area will probably be one with other Commodore computer users, such as Commodore 64, Amiga, PET and VIC-20 users. Usually the best way to contact your Commodore 128 User Group is through local computer stores. Often stores selling Commodore 128 computers will have application forms, and some even serve as the meeting site for the clubs. Other microcomputer clubs in your area may also have Commodore 128 users in them, but if there is not an COMMODORE club, join a general computer group. The help you will get will be worth it. To start your own Commodore 128 User's group, post a notice and meeting time and site in your local computer store. Write to one of the following:

Commodore User Group Coordinator
Commodore Business Machines
1200 Wilson Drive
West Chester, PA 19380

COMPUTE!'s Gazette
P.O. Box 5406
Greensboro, NC 27403
Attn: Commodore Users Groups

RUN Magazine
80 Pine Street
Petersborough, N.H. 03458
Attn: Commodore Users Groups

Ask them to publish a notice that you want to start a Commodore 128 club in your area. Your club will then be listed in the magazine(s) you write and other people in your area will soon join up. If you live in an area with just a few Commodore 128 owners or in a relatively small town, it would be a good idea to affiliate with one of the larger groups. Probably the best bet would be to join the following group:

TORONTO PET USERS GROUP
Department "D"
1912 A Avenue Road, Suite 1
Toronto, Ontario, Canada M5M 4A1

The Toronto group have over 3000 free programs for the various computers made by Commodore along with a club newsletter to keep you informed. The dues are $20, but you will get back a lot more than that in programs and information. Another way to get in touch with fellow Commodore 128 users is via a modem Dial up the computer bulletin boards in your area and look for messages pertaining to Commodore 128's. Usually, you can get in contact with other users very quickly this way. (Ask for the PMS {Public Message System} numbers at your local computer store). If you don't see any references to the Commodore 128, leave a message for people to get in touch with you.

Commodore 128 Magazines

There are several periodicals with information about the Commodore 128. Some microcomputer magazines are general and others are for the Commodore 64/128 specifically. When you're first starting, it is a good idea to stick with the ones dedicated to the Commodore 64 and128 since there are different versions of BASIC for non-Commodore 128 computers. When you become more experienced, you can choose your own, but to get started there are several good ones with articles exclusively on the Commodore 128. These are as follows:

Commodore Microcomputers
Commodore Business Machines, INC.
1200 Wilson Drive
West Chester, PA 19380

Commodore Microcomputers is a publication with a wide variety of articles and programs for the Commodore 128.

Here you will find programming techniques, tips for beginners, new hardware and software available and various applications. Articles range from the simple to the technical, and so regardless of your level of expertise, you will find this extremely useful. Subscriptions are $15.00 per year for 6 issues.

Powerplay
Commodore Business Machines, INC.
1200 Wilson Drive
West Chester, PA 19380

A second magazine for your Commodore 128 is *Powerplay* , a publication dedicated to the more recreational uses of your computer. The articles and programs in this magazine are primarily for home uses of your computer, ranging from games to telecommunications. It is very educational and helpful for novices. Subscriptions are $15.00 per year for 6 issues.

RUN Magazine
80 Pine Street
Petersborough, N.H. 03458

RUN Magazine magazine has an excellent selection of programs, reviews, tutorials and articles for Commodore computers, including the Commodore 128 and 64. The wide range and high quality of the material in this magazine will be an immense help to beginners in programming. The popular column "Magic" is full of interesting and useful tips. The annual *Special Issue* includes a programmer's chart and a list of all the Commodore clubs in the world. Subscriptions are $19.97 for 12 issues.

Ahoy!
45 W. 34th Street- Suite 407
New York, N.Y. 10001

Ahoy! has a number of excellent columns, program listings and even hardware enhancements for the Commodore 128 and 64. There are also excellent review and new product sections. This is another good magazine to use for building up your program library. Subscriptions are $19.95 for 12 issues.

Info
P.O. Box 2300
Iowa City, Iowa 52244

Info has recently expanded to cover the Commodore 128/64 and Amiga. It has the most comprehensive reviews of products you can find in any single place for the Commodore 128. If you want to find out about a product for your computer, check this publication first. Six issues, $18.

Horizon: A Guide to the Commodore Computer
Horizon Press, Inc.
P.O. Box 06680
Portland, OR 97206

This is a smaller and newer magazine, but it is well organized and has a lot of good information for your computer. Articles on CP/M, programs and even stories. It is clean and compact with a little bit of everything. Subscriptions $19 for 12 issues.

Compute!'s Gazette
P.O Box 5406
Greensboro, NC 27403

Compute! Gazzette is dedicated to the COMMODORE-128 /64 and, to a lesser extent, the VIC-20. *Compute!'s Gazette* will provide you with programs and programming techniques that can be applied to your computer. Additionally, it has several general articles on programming, hardware and software that you will find useful. Finally, there are a good deal of bargains on software and peripherals to be found in the magazine. Subscriptions are $15 for 12 issues.

OTHER USEFUL PUBLICATIONS. *Midnight Gazette* is a niftly little Commodore 128/64 review magazine with incisive comments you should look for in computer stores. In addition to the above magazines, there are several others that you may find useful. Publications such as *Compute!*, *Creative Computing* , *Byte* , and *Personal Computing* all have had articles about the Commodore 128. The best thing to do is go through the table of contents in the various computer magazines in you local computer store. This will tell you at a glance if there are any articles or programs for the Commodore 128. As more and more clubs begin springing up, club newsletters can often be an invaluable source of good tips and programs for your computer, and they are a resource that should not be overlooked.

Beyond BASIC

Besides BASIC, your computer can be programmed and can run programs in several other languages. In some cases, special hardware devices are required to run the languages, and there is special software required as well. We'll look at some of these other languages.

Assembly Language. Assembly language is a "low level" language, close to the heart of your computer. It is quite a bit faster than BASIC and virtually every other language we will discuss. To write in assembly language, it is necessary to have an "assembler" to enter code. This language gives you far more control over your Commodore 128 than BASIC, but it is more difficult to learn, and a program takes more instructions to operate than BASIC. (However, the object code is more compact, taking up fewer sectors on your disk.)
One of the best assemblers now on the market for the Commodre128 is,

MERLIN 128
Roger Wagner Publishing
P.O. 582
Santee, CA 92071

In addition to having a full set of macros, editor and other standard assembler features, Merlin comes with an excellent monitor for examining machine code in your programs and ROMs. The manual will help you get started programming in assembly language. All programs written with Merlin can be automatically saved to disk as binary, source and sequential files. Unlike the Commodore 64 version of *Merlin*, the Commodore 128 version is configured for 80 column mode only.

To learn how to program in assembly language, the following two books were found to be the most useful:

1. *Commodore 128 Programmer's Reference Guide*
This book can be purchased from your local book or computer store. Published by Bantam Books, this is the

single most important book for advanced programming you can find.

2. *Assembly Language for Kids: Commodore 64/128*
To get started in assembly language programming, this book is hard to beat, no matter how old you are. It was originally written for the Commodore 64, but updated for the Commodore 128. All of the listings are provided in the built-in mini-assembler inside your machine. This book is full of examples, and it covers the major assemblers for the Commodore 64 and 128. It's available at your local book or computer store or from Microcomscribe.

3. *The Commodore 128 Mode: An Inside View*
For those moving from the beginner to intermediate level of programming, this unique book is an excellent step-by-step tutorial. The authors have a real knack for clearing up the mysteries of how your computer works, and there are lots of programs for you to key in and learn from. It's available at your local book or computer store or from Microcomscribe.

=High and Low Level Languages=

When computer people talk of "high" and "low" level languages, think of high level being close to talking in normal English and low level in terms of machine language, e.g. binary and hexadecimal. Assembly language is a low level language, one notch above machine level. The other languages we will discuss are high-level.

PASCAL

Pascal is a high-level language originally developed for teaching students structured programming. It is faster than BASIC, but is not as difficult to master as assembly language. It is probably the most popular high level language next to BASIC. You will find different versions of Pascal, but the language is fairly well standardized so that whatever version of Pascal you purchase will work with just about any Pascal program. You can also get CP/M versions of Pascal for your Commoodre 128. The following books may be useful to learn about the language.

1. *Elementary Pascal: Learning to Program Your Computer With Sherlock Holmes.* By Henry Ledgard and Andrew

Singer. (New York: Vintage Books.) This is a fun way to learn Pascal since the authors use Sherlock Holmes type mysteries to be solved with Pascal. It is based on the draft standard version for Pascal called X3J9/81-003 and may be slightly different from the version you have, but only slightly so.

2. *Pascal from BASIC*. By Peter Brown. (Reading,MA: Addison-Wesley, 1982). If you understand BASIC, this book will help you make the transition from BASIC to PASCAL. It is written with the PASCAL novice in mind but assumes the reader understands BASIC.

FORTH

FORTH is a very fast high-level language, developed to create programs that are almost as fast as assembly language but take less time to program. Faster than Pascal, Basic, Fortran, Colbol, and virtually every other high-level language, FORTH is programmed by defining "words" that execute routines. New words incorporate previously defined words into FORTH programs. The best part of FORTH is that several versions are public domain. The Fig (FORTH Interest Group) FORTH version is in the public domain, and if you are handy with assembly programming, you might even be able to install your own.

The best source to learn about what is available is through the publication, FORTH Dimensions (see below) and your magazines where Commodore 128 products are advertised. Good books on learning FORTH are now plentiful. For learning FORTH, the following are recommended:

1. *Mastering FORTH* by Anita Anderson, Martin Tracy and Micromotion. (Englewood Cliffs, NJ: Brady, 1984.) This book is the most complete introduction to FORTH using the 83-Standard version of FORTH. It's a good tutorial with helpful illustrations.

2. *FORTH Programming* by Leo J. Scanlon (Indianapolis : Howard S. Sams & Co., 1982). This book uses the FORTH-79 and fig-FORTH models as standards, thereby providing the user with the most widely distributed versions of FORTH. This is a well organized and clear presentation of FORTH.

3. *Starting FORTH* and *Thinking FORTH* by Leo Brodie (Englewood Cliffs: Prentice-Hall). These well written and illustrated works on FORTH for beginners are excellent tutorials. *Starting FORTH*Uses a combination of words from Fig, 79-Standard and polyFORTH, while the later *Thinking FORTH* uses the newer FORTH-83 standard..

4. *Pocket Guide to FORTH* by Linda Baker and Mitch Derick. (Reading, Mass. : Addison-Wesley, 1983). This is a handy alphabetical reference to the FORTH vocabulary and a good explanation of the structure of FORTH. It is good for beginners since each FORTH instruction is explained clearly and easy to find. However, it should be considered a supplement to one of the above books.

5. *FORTH Dimensions* . Journal of FORTH INTEREST GROUP. P.O. Box 1105, San Carlos, CA 94070. This periodical has numerous articles on FORTH and tutorial columns for persons seriously interested in learning the language.

CP/M

For the Commodore 128, there are several excellent CP/M programs available. In fact, CP/M has one of the largest available public domain libraries of any language. Many business programs, including word processors and data base programs, are available in CP/M, and for those primarily interested in business and professional applications, CP/M is certainly something you will want to look into. The Commodore 128 has a built-in Z-80. It comes with a disk with CP/M operating system. When running CP/M on your Commodore 128, the Z-80 microprocessor takes over operations from the 8502. With CP/M, you can then install virtually any program running on CP/M, including other languages such as Pascal and FORTH.

Miscellaneous Languages

Besides the above languages, it is possible to get disks with 'C', COMAL,PROMAL, FORTRAN, LOGO, PILOT and other languages for specialized and general applications. LOGO and PILOT, for example, are used in teaching children programming, while COLBOL is used primarily in business applications. Before you spend time, money, and effort on another language, though, it is highly recommended that you carefully examine your needs. If your main interest is in developing your own programs, first learn BASIC thoroughly and see what you can do with it. If it fits your needs, and its relatively slow speed is sufficient for your uses, then your time will be better spent improving your programming skills in BASIC. If your main interest is in using application programs, then the language capability depends on the programs you are using. Just about all professionally produced programs written with a CP/M operating system will run on a Commodore 128 without any other added hardware. (This includes programs written in Pascal, FORTH, etc.)

Compilers: Turning BASIC Programs into Machine Language

Finally, if you find that programming in BASIC is most suitable for you, but you would like to speed up your programs, a simple way to do that is with a compiler. Essentially, a compiler is a program that transforms your code into a binary file that will run 4 to 5 times faster than Commodore BASIC 7.0. All you do is write the program in BASIC, compile it, and then save the compiled program. From then on, you run your compiled program as a machine language program. There are a lot of BASIC compilers for the Commodore 64 , but ones for the Commodore 128 are just starting to become available. There's an excellent article in the January/February 1986 issue of *Commodore Microcomputers* by Tom Benford on BASIC compilers if you'd like to find out more.

Algorithms

An algorithm is defined as "A set of instructions in computer programming that performs a single task." We have seen many different ones throughout the book. Most have been simple, but with some of our programs, we introduced a little sophistication. The following programs represent two different types of algorithms. First, the sort algorithms show different ways to put strings in alphabetical order. They represent efforts in improving the sorting work done by your computer to make the sorting faster. We also included some different scrolling and entry algorithms in the programs as well. Second, we included a pie chart program representing another way of working with a limited number of space. This time, we have to consider the number of degrees in a circle or ellipse instead of the screen pixel matrix.

Sorts. These programs will sort strings for you. They represent different algorithms, and you can see that certain algorithms work faster than others. The "Bubble Sort" is the simplist and slowest of the three for wholly unsorted lists, but it works very well with partially sorted lists. The "Shell Sort" and "Quick Sort 2" work much faster in general than the "Bubble Sort", but they use far more complex formulas. Test them to see which works best for your sorting needs. You will probably want to use these sorts in your programs that require some alphabetic manipulation.

Bubble Sort

```
10 SCNCLR
20 PRINT "ENTER 10 WORDS"
30 PRINT: FOR X= 1 TO 10
40 INPUT "WORD-> ";W$(X)
50 NEXT X
60 T=X-1
100 REM ***********
110 REM BUBBLE SORT
120 REM ***********
130 FLAG=0 : FOR S=1 TO T-1: IF W$(S) <=
 W$(S+1) THEN 150
140 T$=W$(S):W$(S)=W$(S+1):W$(S+1)=T$:
  FLAG=1 : T=S
150 NEXT S : IF FLAG=1 THEN 130
200 REM *******************
210 REM ALPHABETICAL OUTPUT
220 REM *******************
230 SCNCLR: FOR X=1 TO 10 : PRINTW$(X):NEXT
```

Shell Sort

```
10 SCNCLR  :  RV$=CHR$(18)
20 INPUT"HOW MANY WORDS TO ENTER ";N%
30 DIM A$(N%+2)
40 FOR N=1 TO N%
50 INPUT "ENTER WORD ";A$(N)
60 V=V+1 : NEXT N
70 SCNCLR : FOR I=1 TO 10 : PRINT : NEXT
80 M$=" ALPHABETIZING " : L=20-LEN(M$)/2
```

```
90 PRINT TAB(L)RV$;M$
100 REM ***********
110 REM SHELL SORT
120 REM ***********
130 Y=1
140 Y=2*Y : IF Y<= N THEN 140
150 Y=INT(Y/2) : IF Y=0 THEN 210
160 FOR X=1 TO N-Y:Z=X
170 K=Y+Z : IF A$(Z) <= A$(K) THEN 190
180 SWITCH$=A$(Z) : A$(Z)=A$(K) :
 A$(K)=SWITCH$:Z=Z-Y : IF Z>0 THEN 170
190 NEXT X : GOTO 150
200 REM *****************
210 REM ALPHABETIC OUTPUT
220 REM *****************
230 SCNCLR
240 HOME$=CHR$(19)
250 FOR N=2 TO V+1
260 PRINT A$(N) : C=C+1
270 IF C=20 THEN PRINT HOME$;
280 IF C>19 THEN PRINT TAB(20);
290 IF C=40 THEN GOSUB 310
300 NEXT N : END
310 PRINT RV$  "HIT A KEY" : GETKEY H$
320 SCNCLR : RETURN
```

Quick Sort

```
10 SCNCLR : RV$=CHR$(18)
20 INPUT"HOW MANY WORDS TO ENTER ";N%
30 DIM A$(N%+1)
40 FOR N =1 TO N%
50 INPUT "ENTER WORD  ";A$(N)
60 Z=Z+1 : NEXT N
70 SCNCLR : FOR I=1 TO 10
80 PRINT : NEXT : M$="ALPHABETIZING"
90 L=20-LEN(M$)/2:PRINT RV$;SPC(L);M$:PRINT
 CHR$(146)
100 REM *********
110 REM QUICKSORT
120 REM *********
130 S1=1
140 L(1)=1
150 R(1)=N
160 L1=L(S1)
```

```
170 R1=R(S1)
180 S1=S1-1
190 L2=L1
200 R2=R1
210 X$=A$(INT((L1+R1)/2))
220 C=C+1
230 IF A$(L2)>=X$ THEN 260
240 L2=L2+1
250 GOTO 220
260 C = C1
270 IF X$>= A$(R2) THEN 300
280 R2=R2-1
290 GOTO 260
300 IF L2>R2 THEN 370
310 S=S+1
320 T$=A$(L2)
330 A$(L2)=A$(R2)
340 A$(R2)=T$
350 L2=L2 +1
360 R2=R2-1
370 IF L2<=R2 THEN 220
380 IF L2>=R1 THEN 420
390 S1=S1+1
400 L(S1)=L2
410 R(S1)=R1
420 R1=R2
430 IF L1<R1 THEN 190
440 IF S1> 0 THEN 160
500 REM ******************
510 REM ALPHABETICAL OUTPUT
520 REM ******************
530 SCNCLR
540 FOR N = 2 TO Z +1
550 F=F+1
560 IF F> 22 THEN GOSUB 600
570 PRINTA$(N)
580 NEXT N
590 END
600 REM *********************
610 REM STOP WHEN SCREEN FILLS
620 REM *********************
630 PRINT RV$ "HIT ANY KEY TO CONTINUE "
640 GETKEY AN$
650 F=0 : PRINT CHR$(146)
660 RETURN
```

Pie Chart. We have already had some experience making bar graphs. Making pie graphs or charts involves similar and different algorithms. Bar graphs are programmed on the basis of the number of horizontal and vertical pixels on a screen. Pie charts are based on the number of different 'slices' of a pie can be placed in a 360° circle or ellipse. In the following pie chart program, the algorithm in line 100 recalculates each value in terms of the total values entered divided by 360. That value then goes into making up part of the arcs to complete the circle. However, we did not include algorithms for coloring each different slice or labelling (or even numbering) each silce. See if you can fix up our program to do that.

Pie Chart (Fixer Upper)

```
10 SCNCLR
20 INPUT "HOW MANY EMTRIES ";N%
30 SCNCLR
40 DIM P(N%)
50 FOR X=1 TO N%
60 PRINT "ENTRY  #";X; : INPUT P(X)
70 T=T+P(X)
80 NEXT X
90 FOR X=1 TO N%
100 P(X)=INT(P(X)/(T/360))
110 NEXT
200 REM **********
210 REM DRAW CHART
220 REM **********
230 GRAPHIC1,1
240 COLOR 1,2
250 X=160 : Y = 100 : R=50
260 FOR G=1 TO N%
270 EA=EA+P(G)
280 PRINTBA,EA
290 CIRCLE 1,X,Y,R,,BA,EA,0,1
300 XR=RDOT(0) : YR=RDOT(1)
310 DRAW 1,XR,YR TO X,Y
320 NEXT
```

Key Tricks

Up to this point we have not used a number of short-cuts available on your keys. This is because it was important for you to first get used to the statements and how to use them correctly. Also, as we will see, the short-cuts do not clearly show you what is happening on your computer as fully as writing out the commands. In Appendix K of your *Commodore-128 System Guide* there is a chart that shows how to enter the first one or two letters of a command and then SHIFT the second or third letter to get the entire command. This will save you some time in programming, but it is difficult to read the command until you get used to it. For example, put a program into memory and enter "L {SHIFT-I} and RETURN. The command is the same as entering LIST except you only have to make two key presses instead of four. Now, clear memory and enter the following:

```
10 ? C {SHIFT-H} (147) : A$= "ALLRIGHT"
20 ? T {SHIFT-A} 10); R {SHIFT-I} (A$,5)
```

Before you RUN the program, can you guess what will happen? If you cannot, don't feel bad since it is confusing, especially the way it appears on the screen. When you RUN the program, it will clear the screen and print the message "RIGHT" 10 spaces from the left side of the screen at the top. Now LIST your program, and all the commands are clear. These key short-cuts are handy in some cases and confusing in others. The LIST command is usually from the Immediate Mode, and it is handy to use it in the abbreviated fashion, but until you become better acquainted with programming, these short-cuts may be more confusing than helpful. Use the ones you feel comfortable with, and introduce them gradually.

Function Keys

You know how to change your function keys using KEY. The following are some we liked and found useful:

KEY 1, "SCNCLR : LIST" + CHR$(13) No one wants to list their program on a messy screen.

KEY 3, "ASC(" + CHR$(34). This will get those ASCII values quickly.

KEY 7, "RENUMBER" That's a *long* command used a lot.

KEY 2, "!@#$%$&*" + CHR$(7) Express yourself when your progam bombs.

Utility Programs

What's a Utility? Utility programs are programs that help you program or access different parts of your computer. Some utilities are for writing BASIC programs, some for sound and some for graphics. For example, there are utilties for transforming BASIC programs into sequential files. This is useful if you want to send a program over a modem. In fact, here's a one liner from the immediate mode that will do exactly that. (Not quite as elegant as the commerical versions though.) First load the BASIC program you want converted into memory, then key in the following using a *different* name from your BASIC file. Whatever name you use will be written to disk:

```
DOPEN#1,"FILENAME.S" : CMD1 : LIST : PRINT#1
  : DCLOSE <Return>
```

If you join a Commodore 128 Users Group, you will learn about a lot more utility programs and other's experiences with them. Like all other programs that you are thinking of buying, ask other users about them and get a demonstration of their use first!

Word Processors

Your Commodore 128can be turned into a first class word processor with a word processing program. Word processors transform your computer into a super typewriter. They can do everything from moving blocks of text to finding spelling mistakes. Editing and making changes is a snap, and once you get used to writing with a word processor, you'll never go back to a typewriter again. This book was written with a word processor, and it took a fraction of the time a typewriter would have taken. (Believe me, I've written 10 other books with a typewriter!) There are some limitations with word processors. To give you some help in making up your mind, the following are some features you might want to look for in a word processor:

1. Find/Replace.
Will find any string in your text and/or find and replace any one string with another string. Good for correcting spelling errors and locating sections of text to be repaired.

2. Block Moves.
Will move blocks of text from one place to another. (e.g. Move a paragraph from the middle to end of document.) Extremely valuable editing tool.

3. Link Files.
Automatically links files on disks. Very important for longer documents and for linking standardized shorter documents.

4. Line/Screen Oriented Editing.
Line oriented editing requires locating beginning of line of text and then editing from that point. Screen oriented editing allows beginning editing from anywhere on the screen. The latter form of editing is important for large documents and where a good deal of editing is normally required.

5. Automatic Page Numbering.
Pages are automatically numbered without having to determine page breaks in writing text.

6. Imbedded Code.
In word processors, this enables the user to send special instructions directly to the printer for changing tabs, enabling special characters on the printer and doing other things to the printed text without having to set the parameters beforehand and/or having the ability to override set parameters.

7. Spelling Checker. Even the best spellers make typing and spelling mistakes. A spelling checker compares the words you've written with a dictionary to see if there are any misspelled or "suspect" words. This feature is available in Commodore 128 word processors, and they are well worth any extra cost.

These are just a few of the things to look for in word processors. As a rule of thumb, the more a word processor can do, the more it costs. If you only want to write letters and short documents, there is little need to buy an expensive word processor. However, if you are writing longer, more complex and a wider variety of documents, the investment in a more sophisticated word processor is well worth the added cost. If you have specialized needs (e.g. producing billing forms), you will want to look for those features in a word processor that meets those needs. Therefore, while a word processor may not do certain things, it may be just what you want for your special applications. As with other software, get a thorough demonstration of any word processor on an Commodore 128 before laying out your hard earned cash.

As a cautionary note, word processors take a bit of time to learn to use effectively. It is possible to start writing text immediately with most word processors, but in order to use all of their features, some practice is required. One of the strange outcomes of this is that once a user learns all of the techniques of a certain word processor, he or she will swear it is the best there is! Therefore, avoid arguments about the best word processor. It's like arguing politics and religion. Also, if you have a printer, check to make sure a particular word processor will send text to your printer. This is especially true if you have a parallel printer adaptor.

Data Base Programs

When you need a program for creating and storing information, a "data base" program is required. Essentially, professionally designed data base programs are either sequential or random access files. When you use one, all you have to do is to use the pre-defined fields provided or create fields. For example, a user may want to keep a data base of customers. In addition to having fields for name and address, the user may want fields for the specific type of product the customer buys, dates of last purchase, how much money is owed, date of last payment, etc. Probably more than most other packages, data base programs should be examined carefully before purchasing. Some of the more expensive data bases can be used with virtually any kind of application, but if you're only going to be using your data base to keep a list of names and addresses to print out mailing labels, for example, a data base program designed to do that one thing will usually do it better and for a lot less money. Some word processors can be used as mailing list data base systems in addition to word processors; so you may not even need a data base program if your word processor can handle your needs. On the other hand if your needs are varied and involve sophisticated report generation and changing record fields, then do not expect a simple, specialized program to do the job. There are several other data base programs, including public domain ones available through your club.

Business Programs

Business programs have such a wide variety of functions that it is best to start with a specific business need and see if there is a program that will meet that need. On the other hand there are general business programs that are applicable to many different businesses. Specific business programs include ones that deal only with real estate, stock transactions and nutritional planning. More general programs include "Electronic Spreadsheets," "Financial Planning," and, as discussed above, data base programs.

Unfortunately, business people often spend far too much for systems that do not work. They believe that if one spends a lot of money on software and hardware, it must be better than for a less expensive simpler system. This thinking is based upon a "You Get What You Pay For" mentality, and it leads to systems that are not used at all. Here is where a good dealer or consultant comes in handy. First, since computers are getting more sophisticated and less expensive, often you do not "Get What You Pay For" when purchasing a big expensive one. Often all the business person ends up with is a dinosaur system that is outmoded, too big and too expensive for the needs. Some computer dealers specialize in helping the business person. They will help set up the needed system in your place of business, help train office personnel and provide ongoing support. These dealers will charge top dollar for your system and supporting software, as opposed to the discount dealers and mail order firms; however, if you have any problems you will have someone who will come and help you out. Since the Commodore 128 is so inexpensive to begin with, the extra money spent on buying from a business supportive dealer is well worth the little extra cost. Alternatively, there are several consultants for setting up your system. If you use a consultant, get one who is an independent without any connection to a vested interest in selling computers. Contact one through your phone book and tell him you want to set up a Commodore 128 system in your office and let him know exactly what your needs are. If they are familiar with your system, they will know the available software and peripherals you need. If they try to sell you another computer, that probably means they are unfamiliar with your system, and it is a good idea to try another consultant. However, if you are told by several consultants that you needs cannot be met by a Commodore 128 then you may indeed may need a larger system. I do not mean to sound cynical, but I have encountered too many unhappy business people who bought the wrong system for their needs. One businessman said he paid $14,000 for a computer system that never did work for his requirements and finally bought a microcomputer system for about a tenth of the price and everything worked out fine. This does not mean that a business may not require an expensive mainframe to handle certain business functions, and the Commodore 128 certainly has limitations. However, before you buy any system, make sure it does what you want and have it shown to you working in the manner you expect it to. Often you will find that the less expensive new micros like the Commodore 128 will actually work better than costly big machines.

Graphics Packages

In our chapter on graphics we discussed some of the Commodore 128's capabilities with graphics. However, certain uses require either highly advanced programming skills or a good graphics package. For example, it is possible to draw on the screen in hi-resolution graphics, just as you would with a pallet. The pictures produced can then be saved to disk or printed out to your printer.

Harware

The Commodore 128 is "expandable." That means you can add various attachments to it to make it do more than it does normally. In the back of your machine there are 3 ports where hardware extensions can be attached, and on the right side there are two additional sockets for game paddles and /or a joystick. Game paddles and joysticks are used for games as well as other programs. For games, they guide rockets, space ships and characters against the forces of evil. However, they are also used for drawing graphics and input in other programs as well. Other hardware attachments are interfaces for various peripherals. One, called an IEEE Interface, can connect up to 15 (!) devices to your Commodore 128. Three companies that make IEEE interfaces for the Commodore 128 are:

1. The Computer Works
   2028 West Camel Back Road
   Phoenix, AZ 85015 (602)249-0611
2. Richvale Telecommunications
   10610 Bayview Plaza
   Richmond Hill, Ontario L4C 3N8 (416) 884-4165
   (IEEE Interface with BASIC 4.0)
3. Micro Systems Development, Inc.
   11105 Shady Trail Suite 104
   Dallas, TX 75229 (214) 241-3743

Another important peripheral you may want to consider is a parallel interface board. With these you can connect your Commodore 128 to many different low priced parallel printers. The following are available for your computer:

1. CPI COMMODORE-64 PARALLEL INTERFACE (Works with C-128)
Micro Systems Development, Inc.
11105 Shady Trail Suite 104
Dallas, TX 75229 (214) 241-3743

2. (a) PARALLEL INTERFACE $19.95
   (b) INTELLIGENT PARALLEL INTERFACE $119.95
Micro-Ware Distributing, Inc
P.O. Box 113
Prompton Plains, N.J. 07444 (201) 838-9027
The inexpensive PARALLEL INTERFACE (a) will connect your Commodore 128 to any parallel printer for dumping text to your printer. The INTELLIGENT PARALLEL INTERFACE (b) provides a full emulation of Commodore printers for printing text and graphics. Like software, before you purchase an interface or peripheral, make sure it works with your computer! Unfortunately, many hardware attachments come with such poor documentation that without someone to show you how to work it, it is almost impossible to get them to operate properly. Again this is where a users group proves invaluable. Ask other members about their experiences before buying a peripheral.

Modems and Communications

One of the most exciting things you can do with your computer is to communicate with another computer. Not only can you communicate with another Commodore 128, but you can access other micros and even tie into big mainframes. With the Commodore 128 you are in luck, for with the VICMODEM by Commodore and the right software, you can inexpensively make such connections. Two modems for the Commodore 128, the VICMODEM 1600 and the 1670 Modem/1200, will work with either your Commodore 128 or 64. To be honest, the software that comes with the VICMODEM isn't so hot, and a lot of people have complained that they cannot get their VICMODEM working. However, since the price is so low ($48.95 in one discount store I visited, and FREE with one deal [See *Free Modem!* below]), it seems a shame to give up on it. However, do not despair, for with good communications software, the VICMODEM is a terrific little device! I highly recommend the following communication package:

SMART 64 TERMINAL $39.95
Microtechnic Solutions, Inc.
P.O. Box 2940
New Haven, CT 06515

With Smart 64 Terminal, you can transform BASIC programs into sequential files and send programs to your friends via your modem. You have to send your files in the Commodore 64 mode, but as long as the file is in a sequential file, there's no problem. (In the near future, more Commodore 128 communication software will be available.) It is possible to both upload and download programs, save them on disk and then convert them back to BASIC files to be RUN. The program is simple yet powerful, and it can be used to dump text to any serial printer.

Besides calling your friends and local bulletin boards FREE, you can also tap into some really big information systems for a price. Three such systems include QuantumLink™, The Source,™ Compu-Serve™ and Dow Jones News/Retrieval™. QuantumLink, The Source, and Compu-Serve have all kinds of information, news and programs, charging about $6 an hour when you log on. Talk to some subscribers to these and other such networks to see if they have what you need.

=DEAL!=

I don't know whether they will still have this deal when you get your modem or communication package, but the VICMODEM included free membership in CompuServe and Dow Jones plus one free hour of use. This is a good way to check out these services to see if they are what you need. If they are not, you're not out the cost of membership. So before you purchase a modem, see if they include such a deal.

If finances are not a major consideration, and you do a lot of communications; then you should really consider the more expensive 1670 Modem/1200. The inexpensive modems run at 300 baud, while the 1670 runs four times faster at 1200 baud. The Commodore 1670 comes with first class Commodore 128 communications software too; so you really get a good deal. If you are billed by the amount of time you spend on your phone, a 1200 baud modem will pay for itself in the long run.

=Free Modem!=

Another deal we ran across was a free Commodore 300 baud modem you get by signing up with QuantumLink™. We don't know whether they still have this deal, and you have to sign up for four months of their service, but for $39.80, it looked good to us. Give 'em a call at 1-800-392-8200 and see if they still have it. If not, see if your rich uncle will give you one for your birthday.

Summary

The most important thing to understand from this last chapter is that we have only scratched the surface of what is available for the Commodore 128 computer. There is much, much more than a single chapter could possibly cover and, as you come to know your Commodore 128, you will find that the choice of software and peripherals is limited only by the confusion in making up your mind. There were other items for the Commodore 128 that came to mind, but this chapter and book would have never ended were I to indulge myself and keep prattling on. The software and hardware I suggested were based on personal preferences, and I would suggest that you choose on the basis of your own needs and preferences and not mine. Think of the items mentioned as a random sampling of what one user found to be useful and then after your own sampling, examination and testing get exactly what you need. As you end this book, you should have a beginning level understanding of your computer's ability. Whether you use it for a single function or are a dedicated hacker, it is important that you understand the scope of its capacity to help you in your work, education and play. It is not a monstrous electronic mystery, but rather a tool to help you in various ways. You may not understand exactly how it operates, but you probably do not understand everything about how your car's engine works either, but that never prevented you from driving. Furthermore, like your car, you should think of your computer as a vehicle that will take you where you want, and never again consider it a machine that you must follow.

Glossary
BASIC 7.0 Statements, Functions, and Commands

This glossary is arranged in alphabetical order and contains statements from all of the Commodore 128 BASIC 7.0. The examples are set up to show you how to use the commands and their proper syntax. In some cases when a command has different contexts of usage, more than a single example will be used. Some examples are given in the Immediate mode and some in the Program (deferred) mode <those with line numbers> and some with both. Results are given to show what a particular configuration would create in some examples for clarification. A number of statements were not covered in the text, but they are included here for your convenience.

ABS( ) Gives the absolute value of a number or variable.

```
PRINT ABS(123.45)
```

AND Logical operator used in equations (assignments) and logical expressions.

```
140 IF A$ < > "Y" AND A$ < > "N" THEN GOTO 100
50 ON A$ = "Y" AND SUM AND CT GOTO 100, 200,300
A = A$ = "Y" AND B$ = "Y"
```

POKE 6, A$ = "Y" AND F

APPEND Adds data to end of existing sequential text file.
200 APPEND#9, "MYFORTUNE"

ASC( ) Returns ASCII value of first character in string.
PRINT ASC ("W") or A$ = "Commodore" : PRINT ASC(A$)

ATN( ) Returns arctangent of number or variable.
PRINT ATN (123)

AUTO Automatically enters line numbers in program. Increments can be included if default is different from 10.

BACKUP Copies whole disk from one drive to another on two drive system.
BACKUP D0 TO D1

BANK Specifies current bank 0-15. Default bank is 15 in 128 mode.
BANK 3

BEGIN/BEND All lines between BEGIN and BEND are executed on true conditional, but jumped over if not true.

```
10 INPUT "NUMBER=>";N
20 IF N > 7 THEN BEGIN
30 PRINT "THE VALUE";
40 PRINT "IS GREATER THAN"
50 PRINT "SEVEN"
60 BEND
70 PRINT "THAT'S ALL"
```

BLOAD Load binary file into memory. Specifying drive, device bank and beginning address are optional
BLOAD "SPRITE FILE"
BLOAD "MACHINE CODE",D0,U8,B15,P4864

BOOT Boots both CP/M Plus master disk and bootable binary file.
BOOT
BOOT "HOT STUFF",D0,U9

BOX Draws box in graphics at location specified by opposite corners. Optional angle and paint parameters.

BOX 10,10,100,100
BOX X,Y,X1,Y1,45,1

BSAVE Saves binary file to disk. Must include both starting *and* ending address.
BSAVE "GRAFILE",B15,P1300 TO P1350
BSAVE "SPRITE1",B0,P3584 TO P4096

BUMP( ) Function to indicate which sprites collided. Returns sprite raised to the power of the bit positon (0-7) coresponding to sprites 1-8. BUMP(1)= Which sprites collided, BUMP(2)=Sprite collied with object.
200 SS=BUMP(1)
210 IF SS > 0 THEN GOSUB 300

CATALOG Shows disk files.
CATALOG

CHAR C,X,Y,$,R Displays alphanumeric characters on bit mapped graphics screen in C color at positon X,Y with $ string and R reverse option.
40 CHAR 0,15,2,"LABEL",1

CHR$( ) Returns the character with a given decimal value.

PRINT CHR$(65)

CIRCLE C, X,Y,R,Ry,BA,EA,A,I Draws a circle beginning at X,Y with radius of R and options of vertical radius, beginning and ending angles, object angle and increment other than default of 2 .
40 CIRCLE 1,100,100,20

CLOSE Closes specified OPENed file.
CLOSE 9

CLR All variables and arrays are reset to zero.
40 CLR

CMD Directs output to specified device.
DOPEN#9,"THISLIST" : CMD9 : LIST : PRINT#9 : DCLOSE

COLLECT Cleans up splatted files from disk.
COLLECT

COLLISION T,L Detects type of collison,1=sprite/sprite; 2=sprite/data, and sends program to line L.
500 COLLISION 0,1000
510 COLLISION 1,2000

COLOR A,N Assigns color N to area, A
10 GRAPHIC 4,1 : COLOR 2,5

CONCAT F2$ TO F1$ Combines two data files on disk with F2$ attached to end of F1$.
CONCAT "FILE SECOND" TO "FILE FIRST"

CONT Continue program after a STOP, END, or error. or Ctrl-Break.
CONT

COPY Single file copy on dual disk drive.
COPY DO, "Space Apes" TO D1, "Oz Apes"

COS( ) Returns the cosine of variable or number.
PRINT COS(123)

DATA Strings or numbers to be read.
1000 DATA 2, 345, HELLO, "SAN DIEGO, CALIFORNIA"

DCLEAR Clears all open channels and closes all files. (Similar to the initialize process on the Commodore 64).
DCLEAR

DCLOSE Closes all or specified open disk files.
DCLOSE
DCLOSE#9

DEC( ) Returns decimal values of hexidecimal string.
PRINT DEC("FE")

DEF FN( ) Defines a function for simple real variable.
10 DEF FN A(X) = X * X
20 PRINT FN A(4)
(Results = 16)

DELETE Deletes line or range of lines.
DELETE 40-90

DIM Allocates maximum range of array.
130 DIM A$ (100)

DIRECTORY Displays files on disk
DIRECTORY

DLOAD Load file from disk into memory
DLOAD "GUNNER"

DO/WHILE/UNTIL/EXIT/LOOP Repeats sequence until conditional satisfied.

```
1. 10 DO WHILE X <> 5
   20 PRINT "THIS" : X=X+1
   30 LOOP

2. 10 DO
   20 INPUT #9,G$
   30 PRINT G$
   40 LOOP UNTIL ST

3. 10 DO
   20 INPUT "NAME";N$
   30 IF N$="END" THEN EXIT
   40 LOOP
```

DOPEN Opens disk file for read or write. Defaults to read.
50 DOPEN#9,"CARDFILE",W <Open for write>
50 DOPEN#9,"CARDFILE" <Open for read>
50 DOPEN#9,"CARDFILE",L30 <Open for relative file>

DRAW S,X,Y TO X1,Y1 Draws a line from X,Y to X!,Y1 in source color S
10 GRAPHIC 1,1
20 DRAW 1,1,1 TO 100,100

DSAVE Save program in memory to disk.
DSAVE "THIS GEM"

DVERIFY Compares file on disk with program in memory. Returns 'OK' if identical.
DVERIFY

END Terminates running of program and exits to Immediate mode.

200 END

ENVELOPE Sets envelope number, attack rate, decay rate, sustain, release rate, waveform and pulse width.
ENVELOPE 2,12,12,12,12,3,255

ERR$(ER) Returns error type
10 ROM
RUN
PRINT ERR$(ER)
SYNTAX

EXP( ) Returns e to indicated power.
PRINT EXP (3)

FAST Speeds up to 2MHz, but turns off VIC 40-column screen.
10 FAST
20 FOR X=1 TO 1000 : PRINT X; : NEXT
30 SLOW

FETCH [Used only with expansion RAM module] Gets data from extra memory.
FETCH 255,4864,1,0

FILTER Defines the sound filter parameters in terms of frequency, low-pass filter, bank-pass filer, high-pass filer and resonance.
FILTER 2000,1,0,1,6

FOR/TO/NEXT/STEP Sets up loop with specified bottom and top limit incremented or decremented by optional STEP at NEXT.
40 FOR Z = 1 TO 100 STEP 5
50 PRINT Z
60 NEXT Z

FRE( ) Returns available memory.
PRINT FRE(0)

GET Reads one character at a time from keyboard.
10 DO
20 GET A$
30 PRINT A$
40 LOOP UNTIL A$="A"

GET# Reads in one byte at a time from open file.

```
80 DOPEN#9,"HORSERADISH"
90 GET#9,A$
```

GETKEY Halts execution until kepress detected

```
10 GETKEY A$
```

GO64 Reconfigures into Commodore 64 mode.

```
GO64
ARE YOUR SURE?
```

GOSUB/RETURN Branches to subroutine at given line number and comes back to the next line number after the GOSUB after encountering RETURN.

```
100 GOSUB 200
110 PRINT A$
....
200 A$ = "Commodore 128"
210 RETURN
```

GOTO Branches to given line number.

```
100 GOTO 200
```

GRAPHIC S,C Configures graphic screen of choice (0-5) with clear screen option.

```
20 GRAPHIC 4,1
90 GRAPHIC CLR
```

HEADER N$,N,D,V Formats (and erases!) disk with name N$ and optionally N i.d. number on drive D and device V.

```
HEADER "ELEMENTARY"
```

HELP May type in or press HELP key to highlight most recent mistake.

```
HELP
```

HEX$(D) Returns the hexadecimal value of given decimal number, D.

```
PRINT HEX$(10)
```

IF -- THEN -- ELSE Sets up conditional logic for execution.

```
60 IF A$ = "Q" THEN END  : ELSE GOTO10
```

INPUT Halts program execution until string or numbers entered and ENTER key is pressed. May enter message within INPUT statement.

```
90 INPUT "ENTER WORD-> "; W$(I)
100 INPUT "ENTER NUMBER -> "; A
110 INPUT "ENTER INTEGER NUMBER -> "; N%
120 PRINT "HIT 'RETURN TO CONTINUE ";
130 INPUT R$
```

INPUT# Reads data from specified opened file.

```
230 DOPEN#9,"NAMES"
240 INPUT#9,A$
```

INSTR(A$,B$) Looks for A$ in B$ and returns position of first character of B$. (Optional: INSTR(N,A$,B$) where N equals the starting position in A$ to begin search.)

```
10 FULLNAME$ = "JOHN SMITHSON"
20 LASTNAME$ = "SMITHSON"
30 N = INSTR(FULLNAME$,LASTNAME$)
40 PRINT MID$(FULLNAME$,N)
```

INT( ) Returns integer value of number or variable.

```
PRINT INT(23.45)
```

KEY Redefining or listing function keys.

1. KEY [Shows all current function key words.]
2. KEY 3,"SCNCLR : LIST " + CHR$(13) [Redefines key.]

LEFT$(,) Returns specified number of characters from a given string beginning with character at far left.

```
10 A$ = "GOODBYE"
20 PRINT LEFT$ (A$,4)
```

(Results = GOOD)

LEN Returns the length in terms of number of characters of a specified string.

```
PRINT LEN(A$)
```

LET Optionally used in assigning value to variables.

```
30 LET A = 33
```

LIST Lists program currently in memory to screen.

```
LIST
LIST 30-80
```

LOAD Loads program specified from disk or tape. Optionally, LOAD "FILENAME",R to load and run program.
LOAD "PLOT" <Tape>
LOAD "PLOT",8 <Disk>
LOAD "MACHPLOT",8,1 <Machine code>

LOCATE R,C Sets (R)ow (C)olumn of next pixel. Places cursor at R,C.
30 LOCATE 10,5
40 DRAW TO 200,100

LOG( ) Returns natural logarithm (to base E) of specific number or variable.
PRINT LOG (15) or PRINT LOG (G)

MID$(, ,) Returns a portion of a string beginning with the nth character from the left for the number of characters indicated in the third position.
10 A$ = "WONDERFUL"
20 PRINT MID$(A$,4,3)
(Results = DER)

MONITOR Enters into machine language monitor.
MONITOR

MOVSPR N Moves sprite number N to in one of four configured moves:
1. MOVSPR 1, 100,100 [Moves to specific XY coordinate.]
2. MOVSPR 1, -20,+20 [Moves to left(-)/right(+) and up(-)/down(+) indicated number of pixels.]
3. MOVSPR 1,20;180 [Move sprite indicated number of pixels at angle. **Note:** Uses semi-colon.]
4. MOVSPR 1, 70 #10 [Move sprite at indicated angle and speed 0-15.]

NEW Clears program and variables in memory.
NEW

NOT Logical negation in logical expression.
60 IF A NOT B THEN GOTO 100
70 C = NOT (D AND E)

ON Sets up computed GOTO or GOSUB to branch line number.
190 ON A GOSUB 1000,2000,3000

OPEN Accesses channel to input output device for reading or writing data. [See DOPEN for disk access.]
500 OPEN 1,1,0,"NAMES" <Read cassette>
600 OPEN7,4 <Opens output to printer.>

OR Logical OR in logical expression.
130 IF A=10 OR B = 20 THEN GOTO 190
140 C = D OR E OR K

PAINT S,X,Y "Paints" a specified area of color source S, centered at X,Y.
20 CIRCLE 1,100,100, 50
30 PAINT 1,220,190

PEN( ) Function returns light pen information:

0=X position
1=Y position
2=X position (80)
3=Y position (80)
4=Trigger value

```
 5 GRAPHIC 1,1
10 DO UNTIL PEN(4)
20 X=PEN(0) : Y=PEN(1)
30 DRAW 1,X,Y
```

PLAY Plays string of notes.
PLAY "C D E F G A B"
10 TUNE$="A BAD FAD"
20 PLAY TUNE$

PEEK Returns memory byte's contents of given decimal location.
170 D = PEEK (8000)
180 IF PEEK (8000) = 5 THEN GOTO 200

POINTER( ) Finds the address of variable.
10 V=10
20 PRINT POINTER(V)

POKE Inserts given value in specified decimal memory location.
POKE 8000,10 (Sets memory location 8000 to decimal value 10)

POKE DEC("13FD"),255

POS( ) Gives the current horizontal position of the cursor.
10 PRINT "THIS LINE";: PRINT POS(0)

POT(N) Gives the value of game-paddle potentiometer where N is paddle 1-4.
30 IF POT(3) > 250 THEN GOSUB 200

PRINT Outputs string, number, expression, function or variable to screen.
PRINT 1;2;3; "GO", F$, A; N%

PRINT USING Outputs formatted strings or numbers to screen.
50 PRINT USING "$##.##";N
60 PRINT USING "####.##";2345.00

PRINT # Prints (writes) output to disk or cassette file.
80 PRINT #1, NA$

PRINT#, USING Writes to file in PRINT USING format. (See PRINT USING).
90 PRINT #2, USING "$$####.##";N

PUDEF "n1n2n3n4" Used to replace in positions n1-n4 in PRINT USING other than default symbols.

Default=
n1-blank
n2-comma
n3-decimal point
n4-dollar sign

50 PUDEF "***£" [Changes everything to asterisks except dollar sign which is changed to pound sign.]

RCLR( ) Returns color codes for sources 0-6.
10 R = RCL(1)
20 IF R=4 THEN GOSUB 200

RDOT( ) Returns X(0), and Y(1) coordinates of last plotted pixel and color (2) of pixel.
50 XP=RDOT(0) : YP=RDOT(1)
60 DRAW TO XP,YP

READ Enters DATA statement's contents into variable.
10 READ A : READ B$
20 DATA 5, "BATS"

RECORD#F,N,B Specifies record number N in file F in relative files.
```
50 DOPEN #9,"CHATTER"
60 RECORD#9,7
70 PRINT #9,V$
```

REM Non-executable statement. Allows remarks in program lines.
```
10 BELL$ = CHR$(7): REM RINGS BELL
```

RENAME Used to rename files from BASIC.
```
RENAME "THIS.DAT" TO "THAT.TXT"
```

RENUMBER N,I,O Renumbers BASIC program lines. Optional newnumber, oldnumber and increment. Beginning and increment default is 10.
```
RENUMBER
RENUMBER 200,10,188
```

RESTORE Resets position of READ to first DATA statement.
```
10 FOR I = 1 TO 5 : READ A$(I) : NEXT
20 RESTORE
```

RESUME [L] Goes to first statement of line where error occurred in error-handling routine OR optional line number.
```
10 TRAP 50
20 PRINT N$(88)
30 PRINT "BACK AGAIN"
40 END
50 PRINT "GOTTA DIM THEM ARRAYS!"
60 RESUME 30
```

RETURN Returns program to next line after GOSUB command
```
500 RETURN
```

RIGHT$ (,) Returns the rightmost n characters of given string.
```
10 A$= "COMPUTE!" : PRINT RIGHT$(A$,4)
(Results = MOST)
```

RND( ) Generates a random number less than 1 and greater than or equal to 0.
```
PRINT RND(5)
```

INT (RND (1) * (N) + 1) - Generates whole random numbers from 1 to N, with N being the upper limit of desired numbers. INT(RND*(N2+2-N1)+N1) generates whole random numbers from N1 to N2.

RSPCOLOR ( ) Returns sprite color of multicolor 1 and 2.
110 IF RSPCOLOR(1)=5 AND RESPCOLOR(2)=8 THEN GOSUB 500

RSPPOS (N , V0-V2) Returns sprite N's X,Y position or speed where

    V0=X position
    V1=Y position
    V2=SPEED

```
10 SPRITE 1,1
20 MOVSPR 1,200 #12
30 RX=RSPPOS(1,0)
40  RY=RSPPOS(1,1)
50 RS=RSPPOS(1,2)
60 PRINT RX,RY,RS
70 GOTO 20
```

RSPRITE (N,V0-V6) Function to return parameter characteristics of sprite.

```
100 FOR X=0 TO 6
110 PRINT RSPRITE (1,X), : NEXT
```

RUN Executes program in memory or on disk.

```
RUN
RUN "ENOSE" <From disk>
```

SAVE Records program on tape or disk.

```
SAVE "GRAPH
SAVE "GRAPH",8
```

SCALE Changes scaling in bit mapped and multicolor graphics.

```
10 GRAPHIC 2,1
20 DRAW 1,1,10 TO 300,10
30 SCALE 1,1200,4300
40 DRAW 1,1,15 TO 300,15
```

SCNCLR Clear screen and place cursor in upper left corner of screen.

```
10 SCNCLR
```

SCRATCH Delete file from disk
SCRATCH "DUMBPROGRAM"
ARE YOU SURE?

SGN Returns sign of numeric value with 1 = positive, 0 = 0 and -1 = negative.
K = SGN(-5) : PRINT K

SIN( ) Returns the sine of variable or number.
PRINT SIN(123)

SLEEP X Pauses program X seconds.
60 SLEEP 3

SLOW Returns to 1MHz operation after FAST has been issued.
SLOW

SOUND V,F,D Emits (V)oice sound of (F)requency (0-65535) and (D)uration (0-32767)
SOUND 2,200,200

SPC( ) Skips specified number of spaces in PRINT statement.
PRINT SPC(29); "HERE"

SPRCOLOR M1,M2 Sets multicolor 1 and 2 for sprites.
SPRCOLOR 7,8
SPRDEF
[Example sets up color for multicolor sprite editing.]

SPRDEF Enter into sprite editor.
SPRDEF

SPRITE N,O,C,P,XE,YE,M Defines sprite number, turns sprites on/off, color, sets foreground priority, x and y expansion and mode.
40 SPRITE 2,1,4,0,0,0,1

SPRSAV Stores sprites in one of three ways:
1. SPRSAV 1,A$ Sprite image in sprite 1 stored in A$
2. SPRSAV A$,2 Sprite stored in A$ transferred to sprite
3. SPRSAV 3,4 One sprite duplicated in another. [Sprite3 duplicated in 4]
100 SSHAPE S$,1,1,24,21

110 SPRSAV S$,1
120 SPRSAV 1,2

SSHAPE $,X1,Y1,X2,Y2 Saves rectangular area defined by corners X1,Y1 opposite X,2,Y2 in string buffer.
40 SSHAPE F$,80,80,120,120

SQR( ) Returns the square root of variable or number.
PRINT SQR(128)

ST End of file variable for data files. ST=0 indicates the end of file has not yet been reached.
10 DOPEN #9,"MONEY"
20 DO
30 INPUT#9,M$
40 LOOP UNTIL ST

STASH [Requires expanded memory] Moves memory to expansion RAM.
810 STASH 255,4864,2,4000

STOP Halts execution and prints line number where break occurs. (CONT command will re-start program at next instruction after STOP command.)
100 STOP

STR$( ) Converts number/variable into string variable.
20 T= 123 : T$= STR$(T) : TT$= "$" + T$ + ".00"

SWAP [Requires expanded memory] Exchanges contents of RAM with expansions RAM.
390 SWAP 255,4864,2,0

SYS Execute machine language program in memory.
SYS 4864

TAB( ) Sets horizontal tab from within a PRINT statement.
PRINT TAB(20);"HERE"

TAN( ) Provides the tangent of number or variable.
40 T = 34 : V = 55
50 R = T + V : PRINT TAN(R)

TRON and **TROFF** Turns on trace function for display of line numbers in program execution. (Turned off with TROFF.)

TRON

USR(X) Jumps to machine language program with starting point located in addresses 4633 and 4634. Used for passing parameters between BASIC and machine language routines.

```
220 V=USR(X)
330 PRINT V
```

VAL( ) Used to convert string to numeric value.

```
30 H$ = "123" : PRINT VAL(H$)
```

VERIFY Compares program in memory with program on tape or disk. (See DVERIFY for disk-only.)

```
VERIFY <tape>
VERIFY "FILEFOLDER"
```

VOL Sets volume for sound and music.(0-15)

```
50 VOL 4
```

WIDTH Sets graphic line widths to 1 or 2.

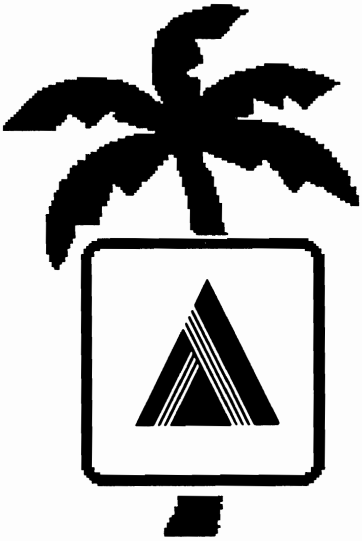
```
30 WIDTH 2
```

WINDOW X1,Y1,X2,Y2,C Defines and enters screen window (40 x 25 max.)

```
WINDOW 10,10,30,30
```

XOR (,) Returns the exclusive OR of values.

```
10 A=10 : B=20
20 C=XOR(A,B)
30 PRINT C
```



Appendix A
ASCII Charts

CHR$ Values

0		51	3	102	□	153	Lt Green	204	□
1		52	4	103	□	154	Lt Blue	205	⧅
2		53	5	104	□	155	Gray 3	206	⧄
3		54	6	105	□	156	Purple	207	□
4		55	7	106	□	157	CRSR left	208	□
5	White	56	8	107	□	158	Yellow	209	■
6		57	9	108	□	159	Cyan	210	□
7		58	:	109	⧅	160	SPACE	211	♥
8	Sh-CMD off	59	;	110	⧄	161	▌	212	□
9	Sh-CMD on	60	<	111	□	162	▄	213	□
10		61	=	112	□	163	□	214	⊠
11		62	>	113	■	164	□	215	○
12		63	?	114	□	165	□	216	♣
13	RETURN	64	@	115	♥	166	▒	217	□
14	Lowercase	65	A	116	□	167	□	218	♦
15		66	B	117	□	168	▄	219	⊞
6		67	C	118	⊠	169	◤	220	▌
17	CRSR down	68	D	119	○	170	▐	221	□
18	RVS on	69	E	120	♣	171	□	222	π
19	Home CRSR	70	F	121	□	172	▗	223	◥

20	Delete	71	G	122	♦	173	└	224	SPACE
21		72	H	123	┼	174	┐	225	▌
22		73	I	124	▒	175	▂	226	▄
23		74	J	125	│	176	┌	227	▔
24		75	K	126	π	177	┴	228	▁
25		76	L	127	◥	178	┬	229	▏
26		77	M	128		179	┤	230	▒
27		78	N	129	Orange	180	▎	231	▕
28	Red	79	O	130		181	▍	232	▄
29	CRSR right	80	P	131		182	▐	233	◤
30	Green	81	Q	132		183	▔	234	▕
31	Blue	82	R	133	f1	184	▀	235	├
32	SPACE	83	S	134	f3	185	▃	236	▗
33	!	84	T	135	f5	186	▕	237	└
34	"	85	U	136	f7	187	▖	238	┐
35	#	86	V	137	f2	188	▝	239	▂
36	$	87	W	138	f4	189	┘	240	┌
37	%	88	X	139	f6	190	▘	241	┴
38	&	89	Y	140	f8	191	▚	242	┬
39	'	90	Z	141	Sh-RETURN	192	─	243	┤
40	(	91	[	142	Uppercase	193	♠	244	▎
41	)	92	£	143		194	│	245	▍
42	*	93	]	144	Black	195	─	246	▐
43	+	94	↑	145	CRSR up	196	─	247	▔
44	,	95	←	146	RVS off	197	─	248	▀
45	-	96	─	147	CLR/HOME	198	─	249	▃
46	.	97	♠	148	INST	199	│	250	▕
47	/	98	│	149	Brown	200	│	251	▖
48	0	99	─	150	Lt Red	201	╮	252	▝
49	1	100	─	151	Gray 1	202	╰	253	┘
50	2	101	─	152	Gray 2	203	╯	254	▘
								255	π

Poke Values

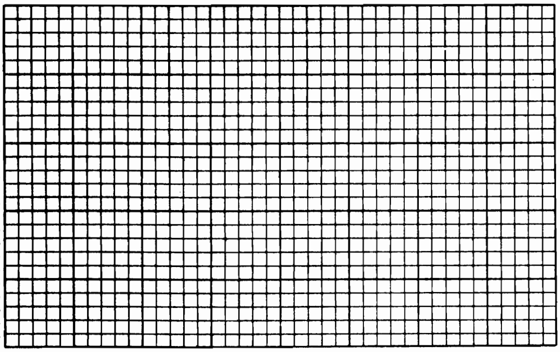
This next set of values corresponds to those POKEd into the text screen beginning at 1024 ($400) [See Appendix B]. The characters differ depending on whether the keys are set for upper case or upper/lower case combined.

#	UC	UC/LC	#	UC	UC/LC	#	UC	UC/LC
0	@	@	51	3	3	102	■	■
1	A	a	52	4	4	103	□	□
2	B	b	53	5	5	104	▃	▃
3	C	c	54	6	6	105	◤	▨
4	D	d	55	7	7	106	▐	▐
5	E	e	56	8	8	107	⊞	⊞
6	F	f	57	9	9	108	▗	▗
7	G	g	58	:	:	109	⊞	⊞
8	H	h	59	;	;	110	⊞	⊞
9	I	i	60	<	<	111	▁	▁
10	J	j	61	=	=	112	⊞	⊞
11	K	k	62	►	►	113	⊞	⊞
12	L	l	63	?	?	114	⊞	⊞
13	M	m	64	⊟	⊟	115	⊞	⊞
14	N	n	65	♠	A	116	▌	▌
15	O	o	66	◫	B	117	▌	▌
16	P	p	67	⊟	C	118	▐	▐
17	Q	q	68	▔	D	119	▔	▔
18	R	r	69	▔	E	120	▀	▀
19	S	s	70	▁	F	121	▄	▄
20	T	t	71	▏	G	122	□	✓
21	U	u	72	▕	H	123	▖	▖
22	V	v	73	╮	I	124	▘	▘
23	W	w	74	╰	J	125	╝	╝
24	X	x	75	╯	K	126	▘	▘
25	Y	y	76	□	L	127	▚	▚
26	Z	z	77	⧅	M			
27			78	⧄	N			
28			79	□	O			

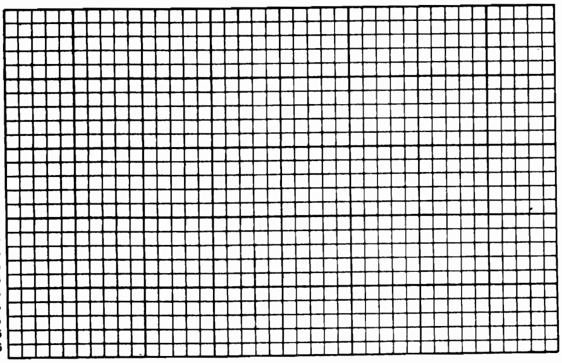
29			80	□	P
30			81	■	Q
31			82	□	R
32	SPACE		83	♥	S
33	!	!	84	□	T
34	“	“	85	◜	U
35	#	#	86	☒	V
36	$	$	87	○	W
37	%	%	88	♣	X
38	&	&	89	◫	Y
39	‘	‘	90	♦	Z
40	(	(	91	⊞	⊞
41	)	)	92	◧	◧
42	*	*	93	◫	◫
43	+	+	94	π	■
44	,	,	95	◪	▧
45	-	-	96		
46	.	.	97	◧	◧
47	/	/	98	⬓	⬓
48	0	0	99	□	□
49	1	1	100	□	□
50	2	2	101	□	□

Appendix B

40 Column Screen Character/Color Addresses



$400	1024
$428	1064
$450	1104
$478	1144
$4A0	1184
$4C8	1224
$4F0	1264
$518	1304
$540	1344
$568	1384
$590	1424
$5B8	1464
$5E0	1504
$608	1544
$630	1584
$858	1624
$680	1664
$6A8	1704
$6D0	1744
$6F8	1784
$720	1824
$748	1864
$770	1904
$798	1944
$7C0	1984



$D800	55296
$D828	55336
$D878	55416
$D8A0	55456
$D8C8	55496
$D8F0	55536
$D918	55576
$D940	55616
$D968	55656
$D990	55696
$D9B8	55736
$D9E0	55776
$DA08	55816
$DA30	55856
$DA58	55896
$DA80	55936
$DAA8	55976
$DAD0	56016
$DAF8	56056
$DB20	56096
$DB48	56136
$DB70	56176
$DB98	56216
$DBC0	56256
$DBE8	56296

THE ELEMENTARY COMMODORE 128

by

WILLIAM B. SANDERS

Learn BASIC 7.0 . . .

This powerful new BASIC makes programming a lot easier than ever before on your **Commodore 128**. Using step-by-step explanations and examples you will learn how to write programs in **BASIC 7.0** on your machine.

Who is this book for?

This book brings it ALL together. Not only does it show you how to write programs that will run on your **Commodore 128**, but it also shows you how to use the **Commodore 1571** drive. Of course, it explains using your 128 with the 1541 drive or cassette system as well. It picks up where your SYSTEM GUIDE leaves off by providing an integrated explanation of how to write practical, useful and just pain fun programs. You also learn how to use your **Commodore 128** with serial and parallel printers, modems, joysticks and how to maximize the use of your computer so that it is not an expensive paper weight or something that gets shut away in the closet.

What does it cover?

If this is your first computer or if you have moved up from a Commodore 64 or other microcomputer, you will find this book a treasure chest full of information. The clear direct writing and the many examples will get the first-time computer owner writing programs in no time at all. The programs are all for the **128 Mode** so you know you will be using the most powerful features of your **Commodore 128**. For those who have moved up from another computer and know something about BASIC programming, this book will clarify **BASIC 7.0** and show some tricks that are unique to this verstion of BASIC.

What do you get?

- **Lots of Programs.** A checkbook, a database, graphics programs and much more. You'd pay double the price of the book for the programs alone!
- **Clear Explanations.** It is the clearest route to an understanding of BASIC 7.0 programming you can get. This book even makes the more advanced concepts simple.
- **Experience!** Dr. William B. Sanders' books have probably taught more people how to program than anyone else's. Over 300,000 copies of the Elementary Commodore 64 alone have sold worldwide in English and foreign translations.

microcomscribe
8982 Stimson Ct., San Diego, CA 92129

ISBN 0-931145-09-0