Programming the VIC

Raeto Collin West

The definitive guide to the Commodore VIC-20 Computer

A **COMPUTE! Books** Publication

$24.95

Programming the VIC

Raeto Collin West

COMPUTE!™ Publications, Inc. abc
One of the ABC Publishing Companies
Greensboro, North Carolina

Printed in the United States of America

ISBN 0-942386-52-3

10 9 8 7 6 5 4 3 2 1

Published by COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809. COMPUTE! Publications is one of the ABC Publishing Companies and is not associated with any manufacturer of personal computers. VIC-20 is a trademark of Commodore Electronics Limited.

Contents

Appendices:

Foreword

Programming the VIC picks up where other programming guides leave off. It covers virtually every aspect of the VIC-20, from simple BASIC commands to complex machine language techniques, and every explanation is written with clarity and style. The result? A comprehensive book, easy to read and understand, that thoroughly reveals the VIC and demonstrates its capabilities.

BASIC programmers, for example, will make frequent reference to the detailed, annotated listing of every VIC BASIC command. Machine language enthusiasts will be especially interested in the ROM maps and in the listings of Kernal routines. Several simple but worthwhile hardware modification techniques have also been included for those with the necessary aptitudes and skills, and every VIC user—regardless of experience—will find the numerous program examples both useful and informative.

This book begins with a brief introduction to VIC-20 BASIC and BASIC programming. It moves on to discuss more advanced programming, including machine language techniques, and also considers specialized applications in the realm of sound and graphics. Chapters are also devoted to disk and tape storage, to memory expansion, and to the selection and use of various peripheral devices.

Author Raeto Collin West, one of the world's foremost authorities on Commodore computers, has distilled years of experience into the pages of *Programming the VIC*. Beginners will discover new information on every page, while more advanced VIC users will appreciate the sophisticated programming techniques herein. This is the first book to thoroughly cover every aspect of VIC programming, and it is certain to become the definitive work in its field.

Chapter 1

Introduction

Chapter 1

Introduction

The VIC-20 is one of the world's most popular microcomputers, and like all best-selling computers, it is both inexpensive and generally easy to use. But many VIC-20 owners have found that the word "simplicity" does not always apply when writing their own programs. Official manuals and even the VIC chip's descriptive sheets have errors. Reliable information has been difficult to find—until now.

This book is designed to teach competent programming on the VIC-20 and to provide a comprehensive reference text for VIC-20 users. It incorporates both BASIC and machine language, for although most people learn BASIC first, it is often necessary to use machine language as well to explain or illustrate particular concepts.

When dealing with computers, it isn't always easy to arrange material so everything falls into an apparently natural sequence. But the organization used here should benefit everyone. This book explains BASIC and machine language by building on easy-to-grasp concepts, and it includes a chapter on combining BASIC with machine language to produce extremely efficient programs. It also contains much useful information on the VIC's internal organization and structure, and concludes with in-depth examinations of topics such as sound, graphics, disk operation, and printers.

The text contains numerous short routines which demonstrate particular concepts or techniques. These examples cover subjects ranging from cassette and disk operation to graphics and sound. They show you how BASIC commands are used, how special features (notably of the VIC chip) operate, and what machine language actually does. They can be used successfully without any knowledge of their internal operation, but if you refer to them while using this book, they will help you acquire BASIC and machine language expertise.

Programming the VIC has been written independently of Commodore and contains criticisms, comments, hints, and a great deal of useful but previously unpublished information. Much of that information I have personally discovered, and I hope to transmit the excitement of discovery to you. This book isn't introductory; there's simply not space to cover the fundamentals, which are often better learned at the keyboard from a friend who knows the machine. But it assumes no prior knowledge of Commodore systems, so anyone with intelligence and some microcomputer experience should be able to grasp the essentials of the VIC fairly easily.

Can the VIC handle graphics without memory expansion? Why do some programs fail to run when memory expansion is used? Can the screen size be expanded, as the literature seems to promise? How can tunes be played during a program without stopping it? Is there any way to get a split screen with the VIC? How do you program the function keys or get 40-column lettering? This is but a small sample of the questions which have puzzled many VIC users. All, and more, are comprehensively answered here.

This book emphasizes information with the widest possible appeal—graphics and tape operations, for instance, instead of hardware interfacing and mathematics. In addition, I've been careful to generalize results rather than to present single facts in unexplained isolation. I've also included comprehensive information on Commodore's major utilities—*Programmer's Aid, Super Expander,* and the *VICMON* machine language monitor—but it has not been possible to detail all commercially available utilities.

The programs have been tested and should work without difficulty, and the screen display photos were all taken using ordinary TV sets connected to ordinary VIC-20s. All effects are obtainable using methods explained in the book. If there are problems, you may have entered a program incorrectly. Alternatively, one of the VIC's many pointers, vectors, or special locations may have been unintentionally reset. The cure in the first case is to carefully check your typing. In the second case, it's usually easiest to save the program, switch off, reload the program, and try again, or use a reset switch of the sort described in Chapter 5.

Much time and effort have been spent to provide a complete and accurate reference book for programmers of the VIC-20. All the information in this book has been carefully checked for accuracy. We cannot, however, accept responsibility for any damage resulting from the use or misuse of the information presented here. Readers are advised to save critical programs and remove important diskettes or cassettes before using this book.

Conventions

BASIC program listings have been made with a program which prints the screen editing commands, colors, and other special characters as {HOME}, {CYAN}, {F1}, and so on in place of the less readable VIC characters. The VIC (video interface chip) and VIC-20 (computer) are usually distinguished, although it is generally obvious which is under discussion. With apologies to people who can spell, we've adopted Commodore's spelling of "Kernal" for the ROM routines which handle the screen, keyboard, and input/output devices. We've used ML as an abbreviation for machine language, and machine code is synonymous with machine language.

When talking about actual memory addresses, we've often used decimal and hexadecimal forms together, for example, 40960 ($A000). This is because the hex form is often more revealing than the decimal form, as experienced software and hardware people will appreciate. However, the decimal form is also very important, because BASIC uses it and many people are happier with it. If you don't understand hexadecimal notation, refer to Appendix M or simply ignore the extra notation.

Hexadecimal numbers are usually represented with a leading $ sign. In accordance with 6502 convention, the symbol # (which is sometimes printed as £) indicates a value and not an address. For example, LDA #$00 is the command to load the accumulator with zero, while LDA $00 loads the accumulator with the contents of location zero.

The Automatic Proofreader

As you use this book, you'll notice that many of the listings contain unusual REM statements at the end of each program line (for example, :rem 123). Those numbers are to be used with the "Automatic Proofreader," a typing verification routine described in Appendix C. Do not type in those rems as you enter the programs in this book; instead, use them in conjunction with the Automatic Proofreader to be certain that you have typed each line correctly. This can save you a great deal of debugging time later on.

Acknowledgments

I have derived considerable benefit from discussions with members of several organizations, chiefly ICPUG (Independent Commodore PET User Group) in the UK, TPUG (Toronto PET User Group) of Canada, and COMPUTE! Publications in the USA. In addition, examination of published software has often yielded interesting techniques and shortcuts.

Trademarks and business names include VIC-20, "The Friendly Computer," and CBM, all trademarks of Commodore Business Machines Limited.

Chapter 2

Getting to Know Your VIC-20

Chapter 2

Getting to Know Your VIC-20

This chapter gives you an overview of the VIC's features. It provides both an introduction to the machine, for complete novices, and a summary for experienced programmers who might want a refresher. Chapters 3 and 4 deal with BASIC programming in depth.

Background

Commodore (CBM) has been making computers since 1977. But it didn't become a household word until the introduction of the low-priced VIC in early 1981, and the 64 in mid-1982.

Both machines proved remarkably successful, far outselling other CBM computers. CBM's earlier experience made it possible for a range of relatively low-priced peripherals (tape drives, disk drives, printers, modems) to be produced more or less concurrently.

All Commodore computers have strong resemblances, and straightforward BASIC programs which run on one CBM machine are likely to run on another, too. If you change from one CBM machine to another, knowing about the VIC will prove very helpful.

A Description of the VIC-20

In the following description, the words *port* and *socket* are used interchangeably. You may find labeling some of the ports helpful if your machine is often disconnected and packed away. As with any electrical equipment, take sensible precautions not to short-circuit connections or otherwise take risks.

Top. The top of the VIC-20 features the power-on light and the keyboard. The power-on light tells you when the machine is turned on; the keys will be discussed in detail in a separate section.

Back panel. The back of the VIC-20 features a number of different sockets and connectors. From left to right, they are the expansion (or cartridge) socket, the video/audio socket, the CBM serial port, the cassette port, and the user port.

The *expansion socket* allows cartridges or memory expanders to be plugged in (only those designed for the VIC-20 will work). See Chapter 5.

The *video/audio socket* connects to a video monitor or, using the modulator supplied with the VIC, to a TV.

The *CBM serial port* is used by CBM disk drives and printers. The serial signal format is unique to CBM, a modified version of the IEEE format used in the older Commodore PETs, and should not be confused with RS-232 type serial communications. See Chapter 15 for information on disk storage and Chapter 17 for information on printers.

The *cassette port* is specially designed to power and control CBM's Datassette tape drive. Other devices may also draw power from this socket. Chapter 14 has full details.

The *user port* is designed to let your VIC communicate with the outside world. It is most often used with a CBM modem, but can also be used, where hardware expertise permits, to control electronic devices for such purposes as process control.

Chapter 4 has hardware information; Chapter 17 deals with modems.

Side panel. The side panel of your VIC has these features, again from left to right:

The *control port,* which is mainly used to connect a joystick but will also accept a light pen, game paddles, or a mouse. The joystick connector is standard. Chapter 16 has full programming information.

On-off switch.

The *power input jack,* which connects the VIC to its external power supply transformer. On older VICs, this connector is a two-pin socket for a single 9-volt AC input. On newer VICs, some of the voltage regulation circuitry has been moved from the computer circuit board into the external power supply. In these units, the two-pin connector has been replaced by a seven-pin DIN socket like that on the Commodore 64's power input.

The Keyboard

Chapter 6 discusses the keyboard and screen in depth. At this point, a brief overview is sufficient.

Alphabetic, numeric, and symbol keys appear just as the keytop legend implies.

SHIFT selects uppercase lettering or the graphics set shown on the right of the key fronts.

The Commodore key selects the graphics set shown on the left of the key fronts, or where this doesn't apply, acts like SHIFT.

The Commodore key and SHIFT can be used together to change the display from uppercase with the full graphics set, to upper- and lowercase lettering plus the left-hand graphics set.

Function keys display nothing; their operation is explained in Chapter 6.

CTRL (Control) acts with the number keys to change the display to black through yellow or from reversed to unreversed. CTRL also slows screen scrolling; this is useful with the LIST command, since it allows a program to be examined more easily.

STOP interrupts BASIC programs. The command CONT allows resumption of BASIC, subject to certain conditions.

Left-SHIFT and STOP, used together, load and then run the next BASIC program on tape.

Right-SHIFT and STOP, used together, cause a BREAK ERROR. This lets you get out of a program's INPUT statement. Chapter 3 explains INPUT in detail.

RESTORE and STOP, used together, provide a panic button that returns the system to its normal state, retaining BASIC in memory. Chapter 6 explains both RESTORE and STOP.

CLR (Clear Screen)/HOME, used alone, leaves the cursor at the top left of the screen. SHIFT and CLR/HOME, used together, erases the screen and leaves it blank.

CRSR (Cursor) keys move the flashing cursor in the directions shown on the keytops. To save time, these keys automatically repeat. Moving the cursor down the screen eventually causes upward scrolling of the display.

INST/DEL (Insert and Delete) are part of the screen editing system, which allows any part of the screen to be altered. Both keys repeat, although INSERT

(SHIFT and INST/DEL) has no effect if it isn't followed by characters on its BASIC line. The VIC's screen editing system is powerful and easy to use, despite a few small anomalies when in quotes mode. Those are discussed below.

The space bar autorepeats. Chapter 6 gives full information.

Double quotes ("). This key is very important in BASIC. Double quotes mark the start (or end) of a string, which is a set of characters stored for future use. Cursor moves, color characters, function keys, and several other special characters are displayed in a distinctive way, as reversed characters, when preceded by double quotes. When active, the system is said to be in *quote mode*. See SHIFT-RETURN below.

RETURN. This key is used by VIC BASIC to signal that information on the screen is ready for processing by the VIC. For example, if you type PRINT "HELLO", pressing RETURN causes HELLO to appear. The steps you went through to put PRINT "HELLO" onscreen are irrelevant. Similarly, RETURN signals that data typed at the keyboard in response to an INPUT statement is ready for the VIC to process.

SHIFT-RETURN moves the cursor to the next BASIC line (as opposed to the next screen line), but without causing the VIC to take in the information. For example, if you begin to correct a line of BASIC but then realize that your correction is not needed, SHIFT-RETURN leaves the program as it was.

VIC BASIC

Everything entered into the VIC is treated as BASIC, unless some other special language is present (for example, machine language from the *VICMON* machine language cartridge).

BASIC can be used in either the direct or the program mode.

Direct mode. You have seen how PRINT "HELLO" is interpreted as an instruction to print the word HELLO. Because of the instant response to the command, this is called *direct* or *immediate* mode.

Program mode. Type 10 PRINT "HELLO" followed by RETURN. Nothing happens. In fact, the line is stored in VIC's memory as a line of a BASIC program; anything preceded by a number (up to 63999) is interpreted as a program line, and stored. This is called *program* mode or (because the commands aren't carried out immediately) *deferred* mode. LIST displays any BASIC in memory; RUN executes it. Thus, RUN in this case prints HELLO onscreen, just like the direct-mode version.

Quote mode. Applies to any BASIC, whether in a program or intended for direct execution. It is important because, in quote mode, VIC's special characters are kept for future use. Quote mode allows VIC's screen control features to be used from within programs.

Notes on line length. The VIC normally has 23 screen lines, each with 22 characters. However, BASIC can link together two, three, or four screen lines into one program line. Thus there is a distinction between screen lines and BASIC (or program) lines. Try PRINT "**********", typing asterisks over several lines. You'll find that the fifth and any subsequent lines aren't linked.

Introduction to BASIC

BASIC is one of many computer languages. It has many powerful features, all of which exist in other computer languages too.

Variables. Variables are an algebraic concept, a symbol that can take on a range of values. FOR X=1 TO 10: PRINT X: NEXT prints ten different values of X, showing how the variable named X can store numbers and allow them to be manipulated. The next chapter discusses variables more fully.

Keywords. These are the commands recognized by BASIC; PRINT is an example. All VIC BASIC keywords are listed, with examples, in the next chapter.

Note that most keywords can be entered in a shortened form. PRINT can be typed in as ?, for example, so 10 ? is read as 10 PRINT.

Screen. The VIC's screen can display up to 256 different characters; these are listed in the appendices. The numerical (or ASCII) value of these characters differs from the value actually present in screen memory, and this point is a little confusing at first.

Screen memory is stored in two separate areas. One holds the characters; the other holds their colors. Chapter 12 explains VIC graphics in detail. Programmers wanting to learn machine language will find examples which use the screen in Chapter 7.

External communication. The VIC communicates with external hardware devices by assigning them device numbers (an appendix lists these; tape, for example, is device 1). LOADing programs from or SAVEing programs to these devices requires commands like LOAD "PROGRAM",8 (which loads PROGRAM from disk), and SAVE "PR" (which saves the BASIC program currently in memory onto tape). Tape is the default device, meaning that tape is assumed if no other device is specifically requested.

Data can also be written to or read from other devices. For example, you can read from and write to tape, but you can only write to a printer. The necessary commands aren't particularly friendly, however; they involve, in addition to the device number, a number for reference called a file number or logical file number. For details, see OPEN and CLOSE in the next chapter's reference section. Full information on tape, disks, printers, and modems is in Chapters 14, 15, and 17.

Error messages. BASIC has built-in error messages which help with debugging (removing mistakes from) your programs. The final section of Chapter 3 lists them alphabetically, along with explanations.

Chapter 3

BASIC Reference Guide

Chapter 3

BASIC Reference Guide

This chapter explains the main features of BASIC, including the component parts and the way they fit together in a program. It is divided into three sections:

BASIC syntax. A summary of the BASIC syntax used on the VIC.

Alphabetical list of keywords. This section includes examples which beginners should try; experienced programmers will find it a useful reference.

Error messages. An annotated listing of error messages.

BASIC Syntax

BASIC is unquestionably the most popular computer language. It's far easier to write, test, and adjust than any other language, and simple programs can be written by people with very little experience.

BASIC is sometimes described as "like English," but the resemblance is tenuous. At any level beyond the simplest, it has to be learned like any other skill. In addition, it is a rather ad hoc language, so that some expressions work differently on different machines. What follows applies to VIC BASIC.

Numerals and Literals

These are actual numbers and strings, not variables. Examples of the first are 0, 2.3E−7, 1234.75, and −744; examples of the second are "hello", "ABC123", and "%!#/" where the quote symbols are delimiters (not part of the literal).

Numerals. The rules which determine the validity of these forms are complex. Generally, numbers are valid if they contain 0–9, +, −, E and . in certain combinations. For instance, 1.23 is valid, but 1.2.3 is not (since only one decimal point may be used). Similarly, 2E3 is valid, but 2EE3 is not (since only one E is permitted). Both 0E and the lone decimal point are accepted as 0.

Exponential notation (using E) may be unfamiliar, but it is not hard to master. The number following E is the power of 10 that is multiplied by the number to the left of the E. In other words, it tells you how many places to move the decimal point left or right to produce an ordinary number. For example, 1.7E3 means "1 point 7 times 10 to the power 3," or 1.7*1000, which is 1700. Similarly, 9.45E−2 means "9.45 times 10 to the power −2," or 9.45*1/100 (which is .0945). SHIFT-E is not accepted. Values outside the ranges .01 to 999999999 and −.01 to −999999999 are output in exponential form.

Strings. Strings can contain any VIC ASCII characters. Tricky characters can be incorporated with the CHR$ function, including the double quotes character (CHR$(34)) and RETURN (CHR$(13)). The maximum length of any string is 255 characters.

Variables

A variable is an algebraic idea. It can be thought of as a named symbol that is used to stand for a quantity or string of characters. X, X%, and X$, respectively, are a number variable (with a minimum value of ±2.93873588E−39 and a maximum value of ±1.70141183E38), an integer variable (some whole number between

−32768 and 32767), and a string of characters (containing from 0 to 255 characters). If the variables haven't been assigned values, numeric variables default to 0. Strings default to the null character, a string of zero length.

A variable, as the name implies, can be changed at will, as X=1: PRINT X: X=2: PRINT X shows.

Variable names are subject to these rules and considerations:

1. The first character must be alphabetic.
2. The next character may be alphanumeric.
3. Any further alphanumerics are valid but will not be considered part of the name.
4. The next character may be % or $, denoting an integer or string variable, respectively.
5. The next character may be (, denoting a subscripted variable.
6. A name cannot include reserved words, since the translator will treat them as keywords and tokenize them. Note that reserved variables (TI and ST) can be incorporated in names, since they are not keywords. However, this is best avoided because it can lead to confusion. For example, a variable like START will be the same as ST.

Each of these rules serves to remove ambiguity and make storage convenient and fast. To appreciate their value, imagine a variable named 1A. If 1A were a valid variable name, 100 1A=1 would require special syntactical treatment to distinguish it from 1001 A=1. And if symbols other than alphanumerics were permitted, so that B= were a valid name, still other problems could appear.

Interconversion between variable types is automatic as far as numerals are concerned. However, string-to-numeric and vice versa require special functions. For instance, L%=L/256 automatically rounds L/256 and checks that the result is in the range from −32768 to 32767. Similarly, L$=STR$(L) and L=VAL(L$) or L%=VAL(L$) converts numerals to strings and vice versa, subject to certain rules. Two other interconversion functions are CHR$ and ASC, which operate on single bytes and enable expressions which would otherwise be treated as special cases.

Operators (Also Called Connectives)

Binary operators combine two items of the same type, creating a single new item of the same type. Unary operators modify a single item, generating a new one of the same type. The numeric operators supported by VIC BASIC are completely standard and are identical in type and priority to those of FORTRAN. The string operators and logical operators are rather less standard.

When a string expression or arithmetic expression is evaluated, the result depends on the priority assigned to each operator and the presence of parentheses.

Parentheses, in either string or arithmetic calculations, guarantee that the entire expression within parentheses is evaluated as a unit. In the absence of parentheses, priority is assigned to operators in this order (the highest priority operator is listed first):

↑	Exponentiation (power)
+ −	Unary plus and minus sign
* /	Multiply and divide
+ −	Binary plus and minus (addition and subtraction)
< = >	Comparisons (less than, equal to, greater than)
NOT	Logical NOT, unary operator
AND	Logical AND, binary operator
OR	Logical OR, binary operator

Logical operators are also called Boolean operators. In an expression like A + B, A and B are called operands, and arithmetic operators are generally straightforward.

Comparisons, too, are straightforward with numbers. However, the rules of comparison are more complex for strings. Strings are compared on a character-by-character basis until the end of the shorter string is reached. If the characters in both strings are identical to the end of the shorter string, then the shorter one is considered the lesser. Characters later in the CBM ASCII sequence are considered greater than those earlier in the series. Thus, even though the string "1" is less than the string "10", the string "5" is greater than the string "449".

Functions

Some BASIC keywords, called functions, are valid only when followed by an expression in parentheses. They may be used either to the right of assignment statements or within expressions and will return a value dependent on the expression in parentheses.

Numeric functions return numeric values and include SQR, LOG, and EXP. String functions return string values and include LEFT$, MID$, RIGHT$, and CHR$. (The last character of all string functions is a $, like that of string variable names.) PEEK, though not a function in the mathematical sense, has the syntax of a numeric function and is considered one.

Certain functions (for instance, FRE) use a so-called dummy parameter. This is an expression required by the interpreter's syntax-checking routine; however, it is ignored by the code which evaluates the function. Typically, dummy parameters are given a value of 0, as in PRINT FRE(0).

Expressions

Expressions may be numeric, string, or logical.

Numeric expressions are a valid arrangement of one or more numerals, numeric functions, real and integer variables, operators, or parentheses. Logical expressions may also be included. Numeric expressions can replace numbers in many BASIC constructions; for example, the right-hand portion of the assignment statement X=SQR(M)+PEEK(SCREEN + J).

String expressions are valid arrangements of one or more of: literals, string functions, string variables, the string operator +, and parentheses. String expressions can replace literals in many BASIC constructions; for example, in X$=MID$("HI" + NAME$,1,L)+CHR$(13), which assigns X$ something like "HI BOH" with a RETURN character added.

Logical (or Boolean) expressions evaluate whether a relationship is true or false (−1 or 0, respectively, in VIC BASIC) and usually contain one or more relational operators (<, =, or >), logical operators, parentheses, numeric, or string expressions. Their main use is in IF statements. For example, IF X$="Y" OR X$ = "N" GOTO 100 is a logical expression.

VIC BASIC doesn't draw sharp distinctions between logical and arithmetic expressions; they are evaluated together and can be mixed. This allows constructions like IF INT(YR/4)*4=YR AND MN=2 THEN PRINT "29 DAYS" or DAYS = 31 + 2*(M=2) + (M=4 OR M=6 OR M=9 OR M=11), where the value −1 generated by a true statement is used in the calculation of days in a month. Such expressions may evaluate differently on other machines (Apple has true = 1), so purists should avoid them.

Another aspect of logical expressions is that it's easy to program them incorrectly. Mistyping may be undetected by BASIC (for instance, if AND is typed OND, the example above interprets the expression as IF INT(YR/4)*4 THEN PRINT "29 DAYS" which is a valid expression but gives the wrong result). In addition, logical expressions have low priority. That means that they're executed last and can have wide but unexpected influence. For example, the expression IF PEEK(X)=0 AND PEEK(X+1)=0 THEN END looks for two zero bytes, but IF PEEK(X) AND PEEK(X+1)=0 ends whenever PEEK(X) is nonzero and PEEK(X+1)=0.

"True" and "false" are actually two-byte expressions like integer variables. A value of −1 means all bits are 1; a value of 0 (false) means all bits are 0. Chapter 5 explains this in greater detail.

Evaluation

Every intermediate result in an expression must be valid. Numerals must be in the floating point range, strings must be no longer than 255 characters, and logical expressions must be within the same integer range.

Statements

A statement is a syntactically correct portion of BASIC separated from other statements by an end-of-line marker or a colon. All statements begin with a BASIC keyword or (where LET has been omitted) with a variable. There are several types of statements; they are listed below.

Assignment statement (LET variable = expression). LET is optional, but its presence often makes the intention much clearer. Languages like Pascal indicate assignments with the symbol ":=", read as "becomes."

Conditional statement (IF [logical expression] THEN [statement]). These statements evaluate the IF expression; if it is true, they invoke the THEN expression.

Statements altering flow of control. These include GOTO, GOSUB, RETURN, and STOP.

Input statement. Such statements get data from a device or a DATA statement; examples are INPUT, GET, INPUT#, GET#, and READ.

Statements governing repetition of blocks of code. For example, FOR-NEXT loops.

Output statements. These statements (PRINT, PRINT#) send data to screen, disk, cassette, or some other device.

REM statements. REM statements allow the programmer to include comments for documentation. The interpreter detects the REM statement and ignores the remainder of the line when the program runs. Program lines which are included for information or as markers, but which never run, can be included in this category.

Type conversion statement. Converts between string variables and literals, real variables and numerals, or integers and numerals, using such functions as ASC, CHR$, INT, STR$, VAL.

Alphabetic Reference to BASIC Keywords

BASIC programs are made up of numbered program lines. Each line is made up of statements, separated by a colon where lines are shared. Spaces generally are ignored; so are multiple colons.

This section lists every VIC BASIC keyword, with explanations and examples, in a uniform format. It will be useful whenever questions arise in the course of program writing. For easier use, error messages are collected in an alphabetic list following this section.

Notes on syntax. Each command's correct syntax is given in a standard way. Parameters are usually either variables or numeric or string expressions, and these are always carefully distinguished. For example, ABS (numeric expression) means that any valid numeric expression is usable with ABS and in turn implies that ABS can be used with variables, as in ABS(X). This allows you to check, for example, whether OPEN N,N is legitimate.

Square brackets denote optional parameters; where they are omitted, a default value is assumed by the system.

Notes on errors. For several reasons, most errors occur in numerical functions. First, there's a chance of a simple syntax error, perhaps an arithmetically incorrect construction or an omitted parenthesis. Second, number parameters have a variety of legitimate ranges: Byte values must be 0–255; memory addresses must be 0–65535; integer and logical expressions must be between −32768 and 32767; no floating point number can be outside about −1E38 and +1E38; zero denominators are not valid; square roots cannot exist for negative numbers; and so on. These errors are relatively easy to correct, so they are mentioned only when, as in DATA, some noteworthy feature exists.

Notes on machine language. Chapter 11 is a guide to VIC's ROMs and includes detailed information on the workings of keywords. However, where it helps elucidate some aspect of BASIC, such information is included in this chapter. Tokens are listed for programmers looking into BASIC storage in memory.

ABS

Type: Numeric function

Syntax: ABS (numeric expression)

Modes: Direct and program modes are both valid.

Token: $B6 (182)

Abbreviated entry: A SHIFT-B

Purpose: Returns the absolute value of the numeric expression in parentheses. In other words, ABS makes a negative number or expression positive.

Examples:

1. **50 IF ABS (TARGET−X)<.01 THEN PRINT "DONE": END**

   This shows how to test for approximate equality. When TARGET is 6, the program ends if X is between 5.99 and 6.01. This kind of test is typically used in iterative computations in which a calculated value is to converge on a given value.
2. **100 IF ABS(X1−X2)<3 AND ABS (Y1−Y2)<3 GOTO 90**

   Seen in game programs, this recalculates starting positions on the screen for two players if randomly generated starting positions are too close.

AND

Type: Binary logical operator

Syntax: Logical or numeric expression AND logical or numeric expression

Modes: Direct and program modes are both valid.

Token: $AF (175)

Abbreviated entry: A SHIFT-N

Purpose: Applies the logical AND operator to two expressions. Each of the 16 bits in the first operand is ANDed with the corresponding bit in the second operand, resulting in a 16-bit, 2-byte integer. The four possible combinations of single bits are as follows:

0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1

Note that the value is 1 only if both bits are 1.

AND has two distinct uses in BASIC. First, it allows the truth value of several logical expressions to be calculated together, as in IF X>2 AND X<3, where X must be between 2 and 3 for the condition to be true. Second, AND turns off selected bits in 16-bit numeric expressions, as in POKE 36879, PEEK(36879) AND 8, which turns the screen black.

AND behaves identically in each example. A logical expression is treated as false when zero and true when −1 or nonzero, and this is responsible for the different interpretations of logical and numeric expressions.

Examples:

1. **100 IF PEEK(J) AND 128=128 GOTO 200**

   Line 200 will be executed if bit 7 of the PEEKed location is set; the other bit values are ignored.
2. **X=X AND 248**

   Converts X into X less its remainder on division by 8, so 0–7 become 0, 8–15 become 8, and so on. This is significantly faster than X=INT(X/8)*8. It works (for X up to 256) because 248 = %1111 1000. Thus, X AND 248 sets the three final bits to zero.
3. **OK=YR>84 AND YR<90 AND MN>0 AND MN<13 AND OK**

   Part of a date validation routine, this uses OK as a variable to validate multiple inputs over several lines of BASIC. If not OK, then branches for re-input when data was unacceptable.

ASC

Type: Numeric function of string expression

Syntax: ASC (string expression). The string must be at least one character long.

Modes: Direct and program modes are both valid.

Token: $C6 (198)

Abbreviated entry: A SHIFT-S

Purpose: This function returns a number in the range 0–255 corresponding to the VIC ASCII value of the first character in the string expression. It is generally used when this number is easier to handle than the character itself. See appendices for a table of VIC ASCII.

Note that the converse function to ASC is CHR$, so ASC(CHR$(N)) has the value N, and CHR$(ASC("P")) is "P". All keys except STOP, SHIFT, CTRL, Commodore key, and RESTORE can be detected with GET and ASC.

Examples:

1. **X=ASC(X$+CHR$(0))**

   Calculates the ASCII value of any character X$, including the null character, which otherwise gives ?ILLEGAL QUANTITY ERROR.
2. **X=ASC(X$)−192**

   Converts uppercase A–Z to 1–26. Useful when computing check digits, where each letter has to be converted to a number.
3. **1000 IF PEEK(L)=ASC("*") THEN PRINT "FOUND AT";L**

   Shows how readability can be improved with ASC; the example is part of a routine to search memory for an asterisk.

ATN

Type: Numeric function

Syntax: ATN (numeric expression)

Modes: Direct and program modes are both valid.

Token: $C1 (193)

Abbreviated entry: A SHIFT-T

Purpose: The arctangent function. This function returns, in radians in the range $-\pi/2$ to $+\pi/2$, the angle whose tangent is the numeric expression. This may take any value within the valid range for floating-point numbers, or approximately ±1.7E38.

To convert radians to degrees, multiply by 180/π. This changes the range of values of ATN from $-\pi/2$ through $\pi/2$ to −90 through 90 degrees. (The π character is available on the VIC keyboard by typing SHIFT and ↑ [up arrow]. Used in expressions, this character acts as a constant with the value of pi. For example, A=180/π:PRINT A will give 57.2957795.)

ATN(X) can be a useful transformation, since it condenses almost the entire number range into a finite set ranging from about −1.57 to +1.57.

Examples:

1. **R+ATN((E2−E1)/(N2−N1))**

   From a program for surveyors, this computes a bearing from distances east and north.

2. **DEF FN AS(X)=ATN(X/SQR(1−X*X))**
   DEF FN AC(X)=π/2−ATN(X/SQR(1−X*X))

   These function definitions evaluate arc sine and arc cos respectively. Remember that arctangent can never be exactly 90 degrees; if necessary, test for this extreme value to avoid errors.

CHR$

Type: String function of numeric expression

Syntax: CHR$ (numeric expression)

Modes: Direct and program modes are both valid.

Token: $C7 (199)

Abbreviated entry: C SHIFT-H (this includes the $)

Purpose: Converts a numeric expression, which must evaluate and round down to an integer in the range 0–255, to a string of length 1 containing the corresponding VIC ASCII character. It is useful for manipulating special characters like RETURN and ", which are CHR$(13) and CHR$(34) respectively. The appendices have a table of VIC ASCII. Note that ASC is the converse function of CHR$.

Examples:

1. **A$=CHR$(18)+NAME$+CHR$(146)**

   Adds {RVS} and {OFF} around NAME$, so PRINTA$ prints NAME$ in reverse.

2. **FOR J=833 TO 848: PRINT CHR$(PEEK(J));: NEXT**
   Prints the name of the last-loaded tape program, by reading the characters from the tape buffer.
3. **PRINT#4, CHR$(27) "E08"**
   Sends the ASCII ESCAPE character, plus a command, to a printer. Special printer features are often controlled in this way.
4. **OPEN 2,2,0,CHR$(38)+CHR$(60)**
   Opens a file to a modem. The two CHR$ parameters are required in this format by BASIC.
5. **CHR$(0)**
   This represents the null character. However, unlike "", it has a length of one and can be added to strings. See ASC for an application. Embedded null characters (as in Y$="12"+CHR$(0)+"34") cause various anomalies.

CLOSE

Type: Input/output statement

Syntax: CLOSE numeric expression. The numeric expression is treated as a logical file number; it must evaluate to 0–255. No error message is given if the file is not open. Actually, CLOSE shares OPEN's syntax checking, so four parameters are valid after CLOSE. However, only the first is used.

Modes: Direct and program modes are both valid.

Token: $A0 (160)

Abbreviated entry: CL SHIFT-O

Purpose: Completes the processing of the specified file and deletes its file number, device number, and secondary address from the file tables. Note that files open for read need not be CLOSED, but files writing to tape or disk should always be CLOSED. Otherwise, tape files will lose the last buffer of data and disks may be corrupted. OPEN 15,8,15: CLOSE 15 is an easy way to correctly close disk files, perhaps after a program stops with ?SYNTAX ERROR while writing to disk. Chapters 14 and 15 have details.

Note, too, that CLOSE is a straightforward command which is made unnecessarily complicated by the behavior of CMD, which needs a final PRINT# to unlisten the serial bus and switch output back to the screen.

Example:

OPEN 4,4: PRINT#4, "HELLO": CLOSE 4
   Closes a file to a printer after sending it a message.

CLR

Type: Statement

Syntax: CLR

Modes: Direct and program modes are both valid.

Token: $9C (156)

Abbreviated entry: C SHIFT-L

Purpose: Appears to erase all BASIC variables currently in memory, leaving the BASIC program, if there is one, unchanged. Any machine language routines in RAM are left unaltered.

Notes:

1. CLR is actually part of NEW. It does most of the things NEW does, while keeping the current program intact. CLR operates by resetting pointers. It doesn't actually erase variables, so in principle they could be recovered.
2. CLR sets string pointers to the top of memory and sets end-of-variables and end-of-array pointers to the end of BASIC. All variables and arrays are thus effectively lost.
3. CLR resets the stack pointer but retains the previous address. Therefore, all FOR-NEXT and GOSUB-RETURN references are lost. Also, if CLR executes within a program, that program continues at the same place.
4. CLR sets the DATA pointer to the start of DATA statements, as if a RESTORE had been executed.
5. CLR aborts input/output activity and aborts (but does not close) files. Keyboard and screen become the input/output devices.

Examples:

1. **POKE 55,0: POKE 56,28:CLR**

   Sets the top of BASIC to 28*256=$1C00; typically to reserve space for graphics.
2. **1000 CLR: GOTO 10**

   This is useful in some simulation programs; variables are erased and recalculated. RUN 10 has a similar effect.

CMD

Type: Output statement

Syntax: CMD numeric expression [, any expression(s)]. The numeric expression, a file number, must evaluate to 1–255. The optional expressions are output (like PRINT) to the specified file.

Modes: Direct and program modes are both valid.

Token: $9D (157)

Abbreviated entry: C SHIFT-M

Purpose: CMD is identical to PRINT#, except that the output device is left listening. Therefore, CMD followed by PRINT directs output from the screen to (typically) a printer. The effect usually lasts until PRINT# unlistens the device.

Notes:

1. CMD is a convenient way to cause a program with many PRINT statements to divert its output to a printer. This is easier than altering PRINTs to PRINT#s. However, CMD has bugs; GET and sometimes GOSUB will redirect output to the screen. The preferred method is PRINT#.

2. CMD is essential to LIST programs to printers.

Examples:

1. **OPEN 4,4: CMD 4, "TITLE": LIST**

   Lists a BASIC program to a printer. Then PRINT#4: CLOSE4 returns output to the screen.

2. **100 INPUT "DEVICE NUMBER";D: OPEN D,D: CMD D**

   Allows PRINT to direct output either to device 3 (screen), device 4 (printer), or elsewhere.

CONT

Type: Command

Syntax: CONT

Modes: Direct mode only. (In program mode CONT enters an infinite loop.)

Token: $9A (154)

Abbreviated entry: C SHIFT-O

Purpose: Resumes execution of a BASIC program interrupted by a STOP or END statement, or by the STOP key. For debugging purposes, STOP instructions may be inserted at strategic points in the program, and variables may be printed and modified after the program has stopped. CONT will cause the program to continue, provided you make no error.

Because STOP aborts files, CONT may be accepted but may not actually continue execution as before. For example, in such a case, output which ought to go to a printer may be displayed on the screen.

Example:

10 PRINT J: J=J+1: GOTO 10

Run this, then press STOP. CONT will cause execution to continue. You can change J, with J=10000, say, and CONT will resume with the new value.

COS

Type: Numeric function

Syntax: COS (numeric expression)

Modes: Direct and program modes are both valid.

Token: $BE (190)

Abbreviated entry: None

Purpose: Returns the cosine of the numeric expression, which is assumed to be an angle in radians.

Examples:

1. **PRINT COS(45* π/180)**

   Prints the cosine of 45 degrees. Conversion from degrees to radians is performed by multiplying by π/180.

2. **FOR J=0 TO 1000 STEP π/10: PRINT COS(J):NEXT**
   Shows the cyclical nature of COS. Large values of the argument don't introduce significant error, because COS uses only the remainder in the range 0 to 2 π.

DATA

Type: Data marker statement

Syntax: Data list of data constants separated by commas

Modes: Program mode only

Token: $83 (131)

Abbreviated entry: D SHIFT-A

Purpose: Enables numeric or string data to be stored in a program. READ retrieves DATA in the same order it's stored in the program.

Notes:
1. DATA statements to store ML can be generated automatically (see Chapter 9).
2. ?SYNTAX ERROR, in a valid DATA statement, means READ and DATA don't match properly.
3. Unnoticed commas can introduce baffling bugs: DATA R,O,Y,G,,B,P, contains eight data items, two of them null characters.
4. Because DATA statements are handled in sequence (RESTORE restarts the sequence), be careful when adding data by appending a subroutine, in case data from a wrong routine is READ.

Examples:
1. **100 DATA "7975, LAZY RIVER ROAD"**
   Shows how double quotes enable commas, colons, and leading spaces to be included in strings.
2. **1000 DATA CU,COPPER,136.2, FE,IRON,35.1**
   Shows how sets of data can be stored. Typically, a loop with READ A$,M$,W might be used to READ each set of three items.
3. **10000 DATA SUB1**
   Might be used to insure that the correct data is being READ—use 1000 READX$: IF X$<> "SUB1" GOTO 1000 to locate SUB1.

DEF FN

Type: Statement

Syntax: DEF FN variable (variable)=arithmetic expression

Modes: Program mode only

Token: DEF: $96 (150), FN: $A5 (165)

Abbreviated entry: DEF: D SHIFT-E (FN has no abbreviated form)

Purpose: Sets up a numeric (not string) function, with one dependent variable, which can be called by FN. Function definitions help save space where an expression needs to be evaluated often. However, their main advantage is improving BASIC's readability.

Notes:

1. Direct mode is forbidden (but without good reason, since function definitions are stored along wth ordinary variables). See Chapter 6 on storage. Defined functions can be called by FN in direct mode.
2. UNDEF'D FUNCTION ERROR results if DEF FN hasn't been executed before FN is used. A ?SYNTAX ERROR caused by an invalid definition refers to the line using FN, even when the line where FN is used is valid.
3. After loading a new program from BASIC, redefine any functions or they'll probably not work. Chapter 6 explains why.
4. Function definitions work by calling a routine to evaluate expressions. Therefore, each definition must fit into one line of BASIC. The IF statement is not allowed, so logical expressions may be necessary. Calling another function definition is valid, however.
5. The dependent variable need not be used in the definition; if not, it is called a dummy variable.

Examples:

1. **100 DEF FN DEEK(X) = PEEK(X)+256*PEEK(X+1)**

   PRINT FN DEEK(50) prints the double byte stored in 50 and 51.
2. **DEF FN MIN(X)=−(A>B)*B−(B>A)*A**

   Returns the smaller of A and B. Note the dummy variable X; any other variable could be used. The awkward form of the expression is necessary to fit it into a single statement.
3. **DEF FN PV(I)=100/(1+I/100)**

   Sets up a present value function, where I is an annual interest rate.
4. **1000 DEF FN E(X) = 1+ X+X*X/2+X*X*X/6+FN E1(X)**
   1010 DEF FN E1(X)=X*X*X*X/24 +X*X*X*X*X/120

   Shows how a very long expression can be spread out.

DIM

Type: Statement

Syntax: DIM variable name [, variable name...]

Modes: Direct and program modes are both valid.

Token: $86 (134)

Abbreviated entry: D SHIFT-I

Purpose: Sets up variable(s) in memory in the order they are listed in the DIM statement. This command is implicitly carried out when a variable is first used. Using a subscripted variable such as X(3) causes the equivalent of DIM X(10), so DIM is generally used only when arrays which require more than ten elements are to be used, or when you wish to specify less than ten elements to conserve memory.

All variables and elements of arrays set up by DIM are set to zero (if numeric) or null (if strings).

Notes:

1. Arrays can use numeric expressions in their DIM statements, so their size can be determined by some input value. They don't have to be of fixed size. Arrays start with the zeroth element, so DIM X(4) sets up a numeric array with five storage locations, X(0) through X(4). One-dimensional arrays can have a maximum of 32767 elements, and not more than 255 subscripts may be used in multidimensional arrays. In practice, ?OUT OF MEMORY ERRORs will result long before these limits.
2. Arrays are stored after variables. Chapter 6 explains the consequences of this. Briefly, new variables, used after an array has been set up, cause a delay, and arrays can be deleted with POKE 49,PEEK(47): POKE 50,PEEK(48), so if intermediate results are computed with a large array, this array can be deleted when it is no longer needed.
3. RAM space occupied by arrays is explained in Chapter 6. Briefly, integer arrays are efficient, but string arrays are very dependent on the lengths of the strings. PRINT FRE(0) gives a quick indication of spare RAM at any time.
4. Large string arrays are vulnerable to so-called garbage collection delays, also explained in Chapter 6. The total number of separate strings, not their lengths, is the significant factor.

Examples:

1. **100 INPUT "NUMBER OF ITEMS";N: DIM IT$(N)**
   Might be used in a sorting program, where any number of items may be sorted.
2. **DIM X,Y,J,L,P$:REM SET ORDER OF VARIABLES**
   Helps speed BASIC by ordering variables, most-used first.
3. **100 DIM A(20): FOR J=1 TO 20: INPUT A(J): A(0)=A(0)+A(J):NEXT**
   Uses zeroth element to keep a running total.
4. **DIM X%(10,10,10)**
   Sets up an array of 1331 integers, perhaps to store the results of three 10-point questionnaires.

END

Type: Statement

Syntax: END

Modes: Direct and program modes are both valid.

Token: $80 (128)

Abbreviated entry: E SHIFT-N

Purpose: Causes a program to exit to immediate mode with READY. This command may be used to set breakpoints; CONT causes a program to continue at the instruction after END.

VIC BASIC doesn't always need END. A program can simply run out of lines, but END is needed if the program is to finish midway. END leaves the BASIC program available for LISTing; you may prefer to prevent this with NEW or SYS 64802 in place of END.

Examples:

1. **10000 IF ABS(BEST−V)<.001 THEN PRINT BEST: END**
   Exits when a repeating process has found a good enough solution to a problem.
2. **100 GOSUB 1000: END: GOSUB 2000: END: GOSUB 3000: END**
   From a program under development, this shows a use of END to set breakpoints. CONT resumes the program after each subroutine is tested.

EXP

Type: Numeric function

Syntax: EXP (numeric expression)

Modes: Direct and program modes are both valid.

Token: $BD (189)

Abbreviated entry: E SHIFT-X

Purpose: Calculates e (2.7182818...) to any power within the range −88 to +88 approximately. The result is always positive, approaching zero with negative powers and becoming large with positive powers. EXP(0) is 1.

Notes:

1. EXP is the converse of LOG. Sometimes the logarithms of very large numbers are used in calculations in place of the numbers themselves. This insures that the result of the calculations will not be a larger number than the computer can handle. EXP transforms the result back to normal. EXP(Q) could be replaced by 2.7182818↑Q, but the EXP form is more readable.
2. EXP is important for its special property that it equals its own rate of growth; it tends to turn up in scientific calculations.

Examples:

1. **PRINT EXP(LOG(N))**
   Prints N, possibly with rounding error, showing EXP and LOG are converses.
2. **100 P(N)=M↑N*EXP(−M)/FACT(N)**
   This is a statistical formula for the probability of exactly N rare events happening (for example, misprints) when the average rate of occurrence is M. FACT(N) holds N! for a suitable range of values.

FOR...TO...(STEP)

Type: Loop control statement

Syntax: FOR simple numeric variable=numeric expression TO numeric expression [STEP numeric expression]

Modes: Direct and program modes are both valid.

Tokens: FOR: $81 (129), TO: $A4 (164), STEP: $A9 (169)

Abbreviated entry: FOR: F SHIFT-O, TO: None, STEP: ST SHIFT-E

Purpose: Provides method to count the number of times a portion of BASIC is executed.

Notes:

1. Processing of FOR-NEXT. The syntax after FOR is checked. Then the stack is tested to see if FOR with the present variable exists. If it does, the previous loop is deleted, so FOR X=1 TO 10: FOR X=1 TO 10 is treated as a single FOR statement. Next, 18 bytes are put on the stack, if there's room. Once there, they won't change, so FOR X=1 TO N has its limit set according to the value of N at the time the statement is executed. Once the loop begins, a change in the value of N will not affect the number of times the loop is executed. 10 FOR X=489 TO 506: PRINT PEEK(X): NEXT lists 18 bytes; these are FOR token, address of loop variable, step size in floating point format, sign of step, value of exit, FOR's line number, and the address after FOR. The step size defaults to 1 if no STEP is specified.

   Because NEXT determines whether the loop will continue, every FOR-NEXT loop is executed at least once, even FOR J=1 TO 0: NEXT. NEXT also checks the loop variable, so NEXT X,Y helps insure correct nesting of loops. It must be preceded by FOR Y and FOR X statements. NEXT adds the step size to the variable value; if the result exceeds the stored limit (or is less than the stored limit in the case of negative step size), processing continues after NEXT. There's no way the system can detect a missing NEXT; if a set of loops is unexpectedly fast, this may be the reason.

   When the step size is held exactly, there is no loss of accuracy in using loops. For example, FOR J=1 to 10000 STEP .5 is exact, as is the default step size of 1. But FOR M=1 TO 1000 STEP 1/3: PRINT M: NEXT has errors. Chapter 6 explores this in greater detail, but this description should help you to pinpoint bugs in loops.
2. When fine-tuning a long program for speed, pay special attention to loops. Obviously, any inefficiencies are magnified in proportion to the loop's size. Take unnecessary work out of loops, and also perhaps DIM variables in decreasing order of importance.
3. It is best to exit a loop from only one point (the final NEXT). Changing the loop variable allows this; for example:

   ```
    5 FOR J=1 TO 9000: GET X$: IF X$="A" THEN J=9000
   10 NEXT
   ```

   finishes early if A is pressed. Note, however, that the loop variable can be changed in error by a subroutine using the same variable.
4. Other loops. The DO WHILE loop, common in some other programming languages, can be simulated with FOR J=−1 TO 0:...:J=CONDITION:NEXT. Processing continues until J is false. Obviously other types of loops may be constructed.

Examples:

1. **PRINT "{CLR}": FOR J=1 TO 500: PRINT "*";: NEXT**

   Prints 500 asterisks.

2. **K=0: FOR J=7680 TO 7680+255: POKE J,K: K=K+1: NEXT**
   POKEs characters 0 to 255 into screen memory. K counts along with J.
3. **FOR J=4096 TO 9E9: IF PEEK(J) <> 123 THEN NEXT**
   PEEKs memory from 4096 searching for a byte equal to 123. When the loop ends, PRINT J gives the location.
4. **5 FOR J=1 TO 12:IF M$<>MID$ ("JANFEBMARAPRMAYJUNJULAUGSEP OCTNOVDEC", 3*J−2,3) THEN NEXT**
   Matches a month, input as M$, giving J=1 to 12 accordingly, or 13 if no match was found. M$ must be the three-letter abbreviation for the month.

FRE

Type: Numeric function

Syntax: FRE (expression)

Modes: Direct and program modes are both valid.

Token: $B8 (184)

Abbreviated entry: F SHIFT-R

Purpose: FRE computes the number of bytes available to BASIC. This is useful with strings, which take up variable space in RAM and are a potential source of ?OUT OF MEMORY ERRORs. FRE first collects any garbage (see Chapter 6) before returning its value. Any expression after FRE is accepted; FRE(0) is usual.

Examples:
1. **1000 IF FRE(0)<100 THEN PRINT "SHORT OF RAM"**
   Prints message when fewer than 100 bytes remain available.
2. **F=FRE(0): DIM X$(50): PRINT F−FRE(0)**
   Prints the number of bytes used up when DIM X$(50) is run.

GET

Type: Input statement

Syntax: GET variable name [,variable name ...]

Modes: Program mode only

Token: $A1 (161)

Abbreviated entry: G SHIFT-E

Purpose: Reads a single character from the current input device, usually the keyboard, and assigns it to the named variable. If the keyboard buffer is empty, string variables are assigned null and numeric variables are assigned 0. GET (unlike INPUT, or Apple's GET) doesn't wait for a keypress, so BASIC can test for a key and continue if there isn't one.

GET X$ is more robust than GET X (which crashes on any nonnumeric key), so GET X$ is nearly always used. GET fetches any ASCII character; see the appendices for a table. STOP, Commodore key, SHIFT, CTRL, and RESTORE aren't detected by GET.

Chapter 6 explains the keyboard buffer and associated keyboard features in depth. Chapter 4 explains how GET may be used to write extremely reliable INPUT-like routines.

Examples:

1. **5 GET X$: IF X$="" GOTO 5: REM AWAIT KEY**
   10 PRINT "{CLR}";X$;ASC(X$): GOTO 5

   This short program prints a key, and its ASCII value, at the top of the screen. You'll see how RETURN is read, plus all the normal keys. Chapter 4 extends this.
2. **100 DIM IN$(200): FOR J=1 TO 200: GET IN$(J): NEXT**

   Gets 200 characters into an array; most of them will be nulls.
3. **200 GET A$,B$,C$**

   This is syntactically valid, but it is more appropriate with GET#. Like GET X, which tends to crash (or give ?EXTRA IGNORED with comma, colon, +, E, etc.), the syntax is accepted because GET, GET#, INPUT, and READ generally use the same Kernal routines. It's not worth removing the relatively useless alternatives.

GET#

Type: Input statement

Syntax: GET# numeric expression, variable name [, variable name ...]

Modes: Program mode only

Token: $A1 (161) then $23 (35) (This is GET then #; GET# has no token of its own.)

Abbreviated entry: G SHIFT-E #

Purpose: Reads a single character from the specified file, which must be open to an input device. Unlike INPUT#, GET# can read characters like colons and double quotes and RETURNs. It can also read files character by character in a way impossible with INPUT# (and without the limitation to 88 characters per string).

Notes:

1. GET# can read from screen or keyboard; however, there's usually no real advantage in this.
2. GET# from tape sets status variable ST=64 at end of file, so ST=64 can test for the end of data if no special marker was used. ST is immediately reset, so the test is needed after each GET#. Chapter 14 has full details.
3. GET# from disk sets ST=64 at end of file; from then on, ST is set to 66 (end of file plus device not responding). Chapter 15 has full details.

Examples:

1.
```
1000 IN$=""
1010 GET#1,X$: IF ASC(X$)=13 GOTO 2000:  REM RETURN FOUND
1020 IN$=IN$+X$: GOTO 1010               :REM BUILD STRING
```

This routine reads in a string from tape or disk, character by character, building IN$ from each character and exiting to the next part of the program when RETURN indicates the end of a string.

2. **100 GET#8,X$: IF ST=64 GOTO 1000: REM END OF DATA**
   Shows how ST detects that there's no more data on file.

GO

Type: Dummy statement

Syntax: Always part of GO TO

Token: $CB (203)

Purpose: Sole function is to allow GO TO as a valid form of GOTO. Occasionally gives problems; some renumbering routines ignore it, and some early CBM machines don't have it. Chapter 8 shows how GO may be modified for your own purposes.

GOSUB

Type: Statement

Syntax: GOSUB line number

Modes: Direct and program modes are both valid.

Token: $8D (141)

Abbreviated entry: GO SHIFT-S

Purpose: GOSUB jumps to its specified BASIC line. It saves its original address so RETURN can transfer control back to the statement immediately after GOSUB. This means a subroutine can be called from anywhere in BASIC while keeping normal program flow. IF or ON allows conditional calls to be made to subroutines.

Notes:

1. Subroutines can be tested in direct mode. For example, L=1234:GOSUB 500 tests the decimal/hex converter in PRINT USING in Chapter 6.
2. Processing GOSUB. Line numbers following GOSUB are scanned by a routine similar to VAL; numerals are input until some other type of character is found. For example, GOSUB and GOSUB NEW and GOSUB 0XX are treated as GOSUB 0. This allows ON-GOSUB to work, since it can then skip commas. After this, GOSUB puts five bytes on the stack.

   The following program prints five bytes, which are a GOSUB token (141), GOSUB's line number, and a pointer to the GOSUB statement:

```
10 GOSUB 20
20 FOR J=500 TO 504: PRINT PEEK(J);: NEXT
```

   The line number is used in the error message if the destination line doesn't exist. It's slightly faster to collect subroutines at the start of BASIC, to reduce the time spent searching for them, and it's also slightly faster to number lines with the smallest possible numbers to cut down time spent processing line numbers.

Note that GOSUBs without RETURNs can fill the stack (see the reference to ?OUT OF MEMORY). 100 GOSUB 100 shows the effect.

3. Miscellaneous. Chapter 6 describes a computed GOSUB, as well as a POP to delete GOSUBs without RETURN. GOSUB 500: RETURN is identical to GOTO 500. Structured programming makes a lot of use of subroutines; collecting BASIC needing multiple IFs or other complex constructions into subroutines helps make programs clearer, at least when the subroutines are commented. *Recursion* is a technique in which a subroutine calls itself, and has an exit routine (otherwise, the stack would fill), but this is irrelevant to BASIC.

Examples:

1. **20000 PRINT "{HOME}{RVS}*** ERROR ";EM$;"{OFF}"**
   20010 FOR J=1 TO 2000: NEXT: RETURN
   A simplified error-message subroutine, this prints an error (EM$ must be set before GOSUB 20000) in reverse at the top of the screen.
2. **500 GOSUB 510**
   510 REM PLAY A NOTE
   Shows how a subroutine can have several entry points. Here, GOSUB 510 plays a note (the actual playing routine is omitted); GOSUB 500 plays it twice.

GOTO; GO TO

Type: Statement

Syntax: GOTO line number or GO TO line number

Modes: Direct and program modes are both valid.

Token: $89 (137) (Separate GO and TO tokens are also accepted.)

Abbreviated entry: G SHIFT-O

Purpose: GOTO jumps to the specified BASIC line. IF or ON allows conditional GOTOs.

Notes:

1. GOTO is usable in direct mode. Direct mode GOTO continues to execute the program in memory without executing CLR, so the program variables are retained. This is similar to CONT, except that any line can be selected from which to continue. Variables can be changed, but these will be lost if BASIC is edited.
2. Line numbers are read by the same routine that handles GOSUB's line numbers, and similar comments apply.

Examples:

1. **TI$="235910": GOTO 1000**
   This is a direct-mode example; the clock is set just short of 24 hours, then the program is run from line 1000, retaining this value of TI$.
2. **100 GET A$: IF A$="" GOTO 100**
   This is a simple loop, awaiting a keypress.

IF

Type: Conditional command

Syntax: IF logical expression THEN line number. IF logical expression GOTO line number. IF logical expression THEN [:] statement [: statement] ...

Modes: Direct and program modes are both valid.

Token: IF: $8B (139), THEN: $A7 (167)

Abbreviated entry: IF: None, THEN: T SHIFT-H

Purpose: Allows conditional branch to any program line or conditional execution of rest of line after IF.

The expression after IF is treated as Boolean (if it equals zero then it is false; if it does not equal zero, then it is true). If true, the statement after THEN is performed; if false, the remainder of the line is ignored and processing continues with the next line.

If the expression is a string, the effect depends on the last calculation to use the floating-point accumulator. Thus, IF X$ THEN may be true or false.

Examples:

1. **1000 LC=LC+1: IF LC=60 THEN LC=0: GOSUB 5000**
   Increments LC; if it is 60, resets it to 0 and calls the routine at 5000 before continuing.
2. **700 IF X=1 THEN IF A=4 AND B=9 THEN PRINT "*"**
   A composite IF statement, identical in effect to IF X=1 AND A=4 AND B=9 THEN... but probably a little faster.
3. **500 IF X THEN PRINT "NONZERO"**
   IF X THEN is the same as IF X<>0 THEN.

INPUT

Type: Input statement

Syntax: INPUT [string literal in quotes;] variable name [,variable name ...]

Modes: Program mode only

Token: $85 (133)

Abbreviated entry: None

Purpose: Accepts data and assigns it to the specified variable. The data is also echoed on the screen. A RETURN is required to end the INPUT.

Notes:

1. INPUT's prompts. INPUT N$ and INPUT "NAME";N$ illustrate the two forms of INPUT. Both print a query followed by a flashing cursor, but the second version also prints NAME, giving NAME? as a prompt. Constructions like INPUT X$,Y$ will print ?? if the first string is input without the second (typing FIRST, SECOND assigns both strings, with no further prompt).
   These examples show how the prompt string can be used:

100 INPUT "{CLR}{DOWN}{RIGHT}{RIGHT}";X$

Clears the screen and prints ? near the top of the screen.

100 INPUT " --{LEFT}{LEFT}{LEFT}{LEFT}";X$

Offers underlining or some other indication of the required length of the input data. There should be two spaces after the opening quotes, and two more lefts than hyphens.

100 INPUT "NAME {RED}{RVS}";N$

Prints the prompt in red and reverse video.

Long prompt strings may cause a bug: If the line wraps around, then the prompt may be added to the input. To avoid this, PRINT "PROMPT";:INPUT X$ is desirable.

Another approach is to POKE the keyboard buffer; in this way ? can be eliminated. Try 100 POKE 198,1: POKE 631,34: INPUT X$. This inserts a quote in the keyboard buffer, effectively pressing quote just after INPUT is run to allow strings like "LDA $A000,X" to be input despite the fact that they contain commas or colons. Chapter 6 discusses this in more detail.

If CMD is in force, the prompt may appear on the printer, not the screen. ?FILE DATA ERROR signals that INPUT is trying to read an output device.

2. How input data is handled. When RETURN is pressed, the line is put into the input buffer for processing. Chapter 6 covers this in more detail; however, note here that one effect is the prohibition of direct-mode INPUT.

   Chapter 7 explains how machine language can do the same work. The data in the buffer is matched with the list of variables after INPUT. ?EXTRA IGNORED means too many separate items were entered; ?? means too few were and asks you for the rest of them. ?REDO FROM START tells you that the variable types didn't match the data.

   Thus, 100 INPUT X expects numeric input; it will accept 123.4 or 1E4 but not HELLO. 100 INPUT X$,Y$ expects two strings; it will accept HELLO,THERE but HELLO,THERE,VIC will lose VIC with ?EXTRA IGNORED.

   Generally, these aren't serious problems unless a program is intended to be foolproof. In that case GET is essential (see Chapter 4). If INPUT were chosen, users could type HOME, CTRL-WHT, SHIFT-STOP, or double quotes (to name a few), and INPUT would be wrecked.

Examples:

1. **INPUT "ENTER NAME";N$: PRINT "HELLO,"N$**

   This is a straightforward string input of N$.
2. **FOR J=1 TO 10: INPUT X(J): NEXT**

   Inputs ten numbers into an array.

INPUT#

Type: Input statement

Syntax: INPUT# numeric expression, variable name [,variable name ...]. The numeric expression is taken to be a file number, and must evaluate after rounding down to 1–255.

Modes: Program mode only

Token: $84 (132)

Abbreviated entry: I SHIFT-N (this includes #)

Purpose: INPUT# provides an easy way to read variables from a file, usually on tape or disk. The format is consistent with PRINT# (that is, it has VIC ASCII characters separated by RETURNs). So as long as INPUT# matches PRINT#, this command should be trouble-free.

Notes: INPUT# is very similar to INPUT. Differences are outlined below:

1. No prompt is printed; obviously, the device can't use it.
2. Some characters aren't recognized (for example, spaces without text, and screen editing characters that are not preceded by quotes). Similarly, PRINT#1, "HELLO:THERE" is read by INPUT# as two strings. Usually, PRINT# with straightforward variables will avoid these bugs.
3. INPUT# can't take in a string longer than 88 characters; ?STRING TOO LONG ERROR appears. Screen input doesn't have this problem, since parts of an excessively long string are simply ignored.
4. ST signals end of file, as with GET#.

Example:

```
10 OPEN 1       :REM READ TAPE FILE
20 DIM D$(100): FOR J=1 TO 100: INPUT#1,D$(J): NEXT
```

Reads 100 strings from a previously written tape file into an array.

INT

Type: Numeric function

Syntax: INT (numeric expression)

Modes: Direct and program modes are both valid.

Token: $B5 (181)

Abbreviated entry: None

Purpose: Converts the numeric expression into the nearest integer less than or equal to the expression. INT(10.4) is 10; INT (−2.2) is −3. The expression is assumed to be within the full range for numerals, between about −1.7E38 and +1.7E38 for floating-point numbers, or between −32768 and +32767 for integer numbers. Thus, L=INT(123456.7) is valid. But L%=INT(123456.7) gives an error, since the result is too large for an integer variable.

Examples:

1. **100 PRINT INT(X+.5) :REM ROUND TO NEAREST**

   This rounds any number, including negative numbers, to the nearest whole number.
2. **100 PRICE= INT(.5+P*(1+MARKUP/100))**

   Calculates price to the nearest cent from percentage markup and purchase price.

LEFT$

Type: String function

Syntax: LEFT$ (string expression, numeric expression)

Modes: Direct and program modes are both valid.

Token: $C8 (200)

Abbreviated entry: LE SHIFT-F (this includes $)

Purpose: Returns a substring made up from the leftmost characters of the original string expression. The numeric expression (must evaluate to 0–255) is compared with the length of the string argument; the smaller of the two determines the substring's length.

Examples:

1. **FOR J=0 TO 20: PRINT LEFT$("HELLO THERE",J): NEXT**

   Prints 20 strings, "", "H", "HE", "HEL", and so on.
2. **PRINT LEFT$(X$ + "--------------------",20)**

   Pads X$ to exactly 20 characters with hyphens.
3. **PRINT LEFT$ ("--------------------",20−LEN(X$)); X$**

   Right justifies X$, preceding it with hyphens; X$ is assumed not to be longer than 20. Other characters, notably spaces, are also usable in this manner to format output.

LEN

Type: Numeric function of string expression

Syntax: LEN (string expression)

Modes: Direct and program modes are both valid.

Token: $C3 (195)

Abbreviated entry: None

Purpose: Determines the length of a string expression. The result is always in the range 0–255. Chapter 6 explains where LEN is taken from.

Examples:

1. **10 PRINT SPC(11−LEN(MSG$)/2) MSG$**

   Centers any (short) message onscreen, by adding leading spaces.
2. **50 IF LEN(IN$)<>L THEN PRINT "MUST BE" L "DIGITS": GOTO 40**

   Rejects an input string of the wrong length.
3. **100 FOR J=1 TO LEN(W$): IF L$=MID$(W$,J,1) GOTO 200**
   110 NEXT: PRINT "NOT FOUND"

   A simplified form of a word game, W$ is tested for the presence of letter L$. The use of LEN(W$) generalizes for any W$.

LET

Type: Assignment statement

Syntax: [LET] Numeric variable = numeric or logical expression. [LET] Integer variable = numeric expression in range −32768–32767, or logical expression [LET] String variable = string expression

Modes: Direct and program modes are both valid.

Token: $88 (136)

Abbreviated entry: L SHIFT-E

Purpose: Assigns a value or string to a variable. LET is usually ignored; VIC assumes LET by default. Variables can be simple or array. If a variable doesn't already exist, LET sets it up, including arrays, which unless DIMensioned have default dimension(s) of 10.

Notes:
1. Chapter 6 has full details on variables' storage; it also has a routine, VARPTR, showing how LET can be used from machine language. Since LET is rarely used, it can be modified by the user. Chapter 7 has examples of how this is done.
2. Variables can be reassigned with total freedom, so be careful not to use a variable for two purposes simultaneously. This is often a bug with subroutines, because they are typically somewhere else, out of sight.

Examples:
1. **X=123456** **LET X=123456**
   Both set X to 123456.
2. **Q%=Q/100**
   Sets Q% equal to the hundreds part of Q, so if Q=1234, Q%=12.
3. **LET QH%=Q/256: LET QL%=Q−QH%*256**
   Sets QH% and QL% equal to the high and low bytes of Q.

LIST

Type: Command

Syntax: LIST [linenumber] [-[linenumber]]

Modes: Direct and program modes are both valid.

Token: $9B (155)

Abbreviated entry: L SHIFT-I

Purpose: Displays part or all of BASIC in memory to the screen or (with CMD) to disk, tape, or modem.

Notes:
1. Line numbers must be ASCII characters, not variables.
2. LIST uses many RAM locations, so it always exits to READY mode if used within a program.
3. LOAD errors and other errors show up in LIST. For example, 43690 ++++++++++++++ is caused by $AAs in memory, which lists as + and is treated as a line number (43690=170*256 + 170).

4. Chapter 6 lists BASIC tokens and has examples of BASIC storage in memory and has programs to modify LIST in useful ways. (Chapter 8 shows how it's done.) REM has notes on the way LIST interprets screen-editing and other characters. TRACE is a modified LIST which works while a program runs. UNLIST shows ways to protect your programs.

Examples:

1. **LIST 2000-2999**

   Displays all BASIC program lines currently in memory which have line numbers equal to or greater than 2000 and less than or equal to 2999.
2. **LOAD "$",8 :LIST**

   Displays a disk directory, which is stored as though it were BASIC.
3. **1000 LIST -10**

   Lists all lines in the current BASIC program which have line numbers of 10 or less; useful in printing a program description on the screen. However, execution of this line will stop the program, and CONT will not restart it.
4. **LIST 1100-**

   Lists all lines in the current BASIC program with line numbers of 1100 or greater. If there is no line 1100 in the current program, the listing begins with the first existing line greater than 1100.

LOAD

Type: Command

Syntax:

Tape: LOAD [string expression[,numeric expression [, numeric expression]]]. All parameters are optional. The first numeric expression must evaluate to 1 (device number). The second normally evaluates to 0 (BASIC load) or 1 (forced load). Chapter 14 has full details.

Disk: LOAD string expression, numeric expression [, numeric expression]. Disk LOAD requires a name and a numeric expression, typically 8, the device number. The second numerical parameter has the same meaning as in tape LOAD. Chapter 15 has full details.

Modem: LOAD is not implemented. An attempt to load from device number 2 gives an error message.

Modes: Direct and program modes are both valid.

Token: $93 (147)

Abbreviated entry: L SHIFT-O

Purpose: Loads memory with a BASIC program or with machine language, graphics, etc., from tape or disk storage. In its simplest form, LOAD then RUN loads BASIC from tape and runs it. (SHIFT-STOP does this too.)

Notes:

1. LOAD is followed by a standard set of messages, like PRESS PLAY ON TAPE, OK when the cassette starts, and so on. These are listed in the tape and disk chapters. Program-mode LOADs don't have these messages (apart from PRESS PLAY ON

TAPE, which can't be avoided), so the screen layout can be kept neat.
2. Loading BASIC nearly always requires that LOAD's third parameter be 0. This allows LOAD to relink BASIC, so that any start-of-BASIC position is acceptable. For example, LOAD "BASIC PROG" loads that program from tape into a VIC with any memory configuration, and prepares it for RUN. In fact, POKE 43,LO:POKE 44,HI:POKE HI*26+LO,0:NEW:LOAD can put BASIC anywhere you choose, if there's room for it.
3. Loading graphics definitions and the like in ML is generally trickier than in BASIC. It needs something like LOAD "CHARSET",1,1 to insure it's put back where it came from. *VICMON's* .L load command in effect does this. Chapter 6's BLOCK LOAD explains how blocks of bytes can be loaded without disturbing BASIC's running.
4. Program-mode LOADs generally chain BASIC; see CHAIN in Chapter 6. You may also want to look at OLD, which explains how to chain a long program from a shorter one.

Examples:

1. **LOAD or LOAD "",1**

   These are tape loads, having identical effects. Either loads the first BASIC program from tape.
2. **LOAD "PROG"**

   Loads the program with the filename PROG from tape. Actually, because of the filename checking scheme used, the command will load the first program encountered on tape having PROG as the first four letters of its filename. Thus, this command would also load PROGRAM, PROGDEMO, etc.
3. **LOAD "PROG",8**

   Loads PROG from disk, and nothing else. The disk drive will indicate an error if PROG is not found on the disk.
4. **LOAD "PAC*",8**

   Typical disk pattern-matching, this loads the first program found on disk with a filename beginning with PAC, for example: PACMAN, PACKER, etc.
5. **10000 PRINT "PLEASE WAIT": LOAD "PART2"**

   Loads, then runs, the tape program PART2 (or PART20, etc.), from within BASIC. If the key on the tape deck is pressed, no message appears on the screen.
6. **0 IF X=0 THEN X=1: :LOAD "GRAPHICS",1,1**

   Loads graphics into a fixed area of memory, then reruns this program from the start. The maneuver with X prevents repeated LOADs.

LOG

Type: Numeric function

Syntax: LOG (numeric expression)

Modes: Direct and program modes are both valid.

Token: $BC (188)

Abbreviated entry: None

Purpose: Returns the natural logarithm (log to the base e) of a positive arithmetic expression. This function is the converse of EXP. Logarithms transform multiplication and division into addition and subtraction; for example, LOG(1) is 0 since multiplication by 1 has no effect. Logarithms are mainly used in scientific work; their liability to rounding errors makes them less suitable for commercial work.

Examples:

1.
```
PRINT LOG(X)/LOG(10)        :REM LOG TO BASE 10
PRINT LOG(X)/LOG(2)         :REM LOG TO BASE 2
PRINT EXP(LOG(A)+LOG(B))    :REM PRINTS A*B
```

These are all standard results.

2.
```
LF=(N+.5)*(LOG(N)-1)+1.41894+1/(12*N)
```

Defines LF, an approximation to LOG(N!), so that EXP(LF) approximately equals N!. Shows LOG helping handle very large numbers.

MID$

Type: String function

Syntax: MID$ (string expression, numeric expression [,numeric expression])

Modes: Direct and program modes are both valid.

Token: $CA (202)

Abbreviated entry: M SHIFT-I (this includes $)

Purpose: Extracts any required substring from a string expression. The first numeric parameter is the starting point (1 represents the first character of the original string, 2 the second, and so on). The final parameter is the length of the substring. If this isn't used, the substring extends to the end of the original string.

Example:

```
10 INPUT X$: L=LEN(X$)
20 FOR J=1 TO L: PRINT MID$(X$,L-J+1,1);: NEXT
```

Inputs a string, then prints the characters of the string in reverse order, one character at a time.

NEW

Type: Command

Syntax: NEW

Modes: Direct and program modes are both valid.

Token: $A2 (162)

Abbreviated entry: None

Purpose: Allows a new BASIC program to be entered.

Notes:

1. Any previous program in memory is not erased, although it will appear to have been. Actually, most of BASIC and all ML routines and data are unaltered. NEW

puts zero bytes at the start of BASIC, resets pointers, and CLRs, which erases variables and aborts files (among other things). The program OLD in Chapter 6 will recover BASIC programs after an accidental NEW (or after resetting by Chapter 5's method), provided new program lines haven't been entered.
2. NEW may generate ?SYNTAX ERROR. See the error message notes.

Examples:

1. **NEW**

   In direct mode, readies VIC for a new program. (Without NEW, any program lines typed in will be simply added as extra lines to the program already in memory.)
2. **20000 NEW: REM PROGRAM NO LONGER WANTED**

   This ends program execution and exits to READY mode. The program won't LIST and appears erased.

NEXT

Type: Loop control statement

Syntax: NEXT [numeric variable][,numeric variable ...]

Modes: Direct and program modes are both valid.

Token: $82 (130)

Abbreviated entry: N SHIFT-E

Purpose: Marks the end of a FOR-NEXT loop. See FOR, which has a detailed account of loop processing.

Examples:

1. **FOR I=1 TO 10: FOR J=1 TO 10: PRINT I*J;:NEXT J: PRINT: NEXT**

   Prints an (unformatted) 10-times table. Note that NEXT:PRINT:NEXT works too; in fact, it's a little faster. NEXT J: NEXT I can be replaced with NEXT J,I. Once a program is debugged, the variables can generally be removed; however, they do improve readability.
2. **80 FOR J=1 TO 2000: GET X$: IF X$=" " THEN NEXT**
   81 FOR J=0 TO 0: NEXT

   This delays 10 seconds, unless a key is pressed; if it is, line 81 gets rid of the still-live J loop.
3. NEXT can appear anywhere, allowing clumsy constructions like this one:

```
10 FOR J=1 TO 3: GOTO 40
20 NEXT K
30 NEXT J: END
40 FOR K=1 TO 2: GOTO 20
```

NOT

Type: Unary logical operator

Syntax: NOT logical or numeric expression. Numeric expressions must evaluate after rounding down to −32768 to 32767.

Modes: Direct and program modes are both valid.

Token: $A8 (168)

Abbreviated entry: N SHIFT-O

Purpose: Computes the logical NOT of an expression. Logical expressions are converted from false to true, and vice versa. Numeric expressions are converted to 16-bit form, and each bit is inverted. The result, like the original, is always in the range −32768 to 32767, and always equals −1 minus the original value. Thus, NOT of arithmetic expressions does not necessarily convert true to false.

Note:
NOT has precedence over AND and OR, so NOT A AND B is identical to (NOT A) AND B. The usual rules of logic apply to NOT, AND, and OR.

Examples:

1. **55 IF X$=CHR$(34) THEN Q=NOT Q**
   Flips a flag, denoting quote mode on or off.
2. **IF NOT OK THEN GOSUB 20000: REM ERROR MESSAGE**
   Uses the result of variable OK, set in earlier tests, to test for errors.

ON

Type: Conditional statement

Syntax: ON numeric expression GOTO line number [,line number ...]. ON numeric expression GOSUB line number [,line number ...]
The numeric expression must evaluate and round down to 0–255.

Modes: Direct and program modes are both valid.

Token: $91 (145)

Purpose: Allows a conditional branch to one of the listed line numbers, depending on the value after ON. If 1, the first line number is used; if 2, the second, and so on. If the value is zero or is larger than the number of line numbers in the list, processing continues with the next statement. This provides a readable method of programming multiple IFs, provided a variable takes consecutive values.

Examples:

1. **ON SGN(X)+2 GOTO 100,200,300**
   Branches to one of three lines, depending on X being negative, zero, or positive.
2. **90 ON ASC(IN$)−64 GOTO 100,200,300,400**
   Jumps to one of the lines, depending on whether IN$ is A, B, C, or D.
3. **30 ON 6*RND(1)+1 GOSUB 100,200,300,400,500,600**
   Selects one of six subroutines at random in a game.

4. **100 ON X GOTO 400,410,420,430,440,450**
   101 ON X−6 GOTO 460,470,480
   Shows how the options can be spread over several lines (provided X isn't 0).

OPEN

Type: Input/output statement

Syntax:

Tape: OPEN numeric expression [,numeric expression [,numeric expression [,string expression]]]. The first numeric expression (file number) must evaluate to 1–255; the second is the device number 1; the third sets read or write type; and the optional string expression is the filename. Chapter 14 has full details.

Disk: OPEN numeric expression, numeric expression, numeric expression, [,string expression]. Here, the first expression (file number) must be in the range 1–255, the second expression (device number) is usually 8, and the third (secondary address) is usually 2–14. The string expression is generally a command like "SEQ FILE,W" which is processed by the disk drive itself. Chapter 15 has full details.

Modem or other RS-232 device: The same as for disk, except that the device number is 2 and the string expression is a pair of bytes which set transmit/receive features. Chapter 17 has full details.

Printer and other write-only devices: File and device numbers are essential. The third parameter may or may not matter. The string is irrelevant. See Chapter 17.

Modes: Direct and program modes are both valid.

Token: $9F (159)

Abbreviated entry: O SHIFT-P

Purpose: OPEN sets up a file to write or read (sometimes both) data to or from tape, disk, etc. Example: OPEN 1,1,1,"TAPE FILE" opens file #1, called "TAPE FILE", to the cassette. PRINT#1 followed by data will write any data you want to that tape file, and CLOSE 1 leaves a complete new file called "TAPE FILE", which can be read back later by OPEN 1 and INPUT#1,X$ or similar statements.

Ten files (enough for almost any purpose) can be open at once, but each must have a different logical file number (the first parameter of OPEN) so they can be distinguished. Three tables in RAM store the file numbers along with their device numbers and other parameters.

Note:
Tape and disk filenames can't exceed 16 characters.

Examples:

1. **Tape: OPEN 2,1,0,"TAX"**
   Opens a file from tape called TAX (or TAXI, etc.) for reading (since the third parameter is 0) and assigns it #2 so INPUT#2 or GET#2 will fetch data. This is identical to OPEN 2 except that the file is asked for by name. OPEN 2 opens the first file it finds. With tape, OPEN reads tape until it finds a header.
2. **Disk: OPEN 1,8,3,"ORDINARY FILE,S,R"**
   Opens the sequential file on disk called "ORDINARY FILE," making it ready for reading by INPUT#1 or GET#1. The ,S,R is shorthand for ,SEQ,READ.

3. **Modem: OPEN 2,2,0,CHR$(6)**
   This is a common OPEN, preparing the modem for PRINT#2 and INPUT#2.
4. **Printer: OPEN 4,4: REM OPENS FILE#4 TO DEVICE#4**

OR

Type: Binary logical operator

Syntax: Logical or numeric expression OR logical or numeric expression. Numeric expressions must evaluate after rounding down to −32768 to 32767.

Modes: Direct and program modes are both valid.

Token: $B0 (176)

Abbreviated entry: None

Purpose: Calculates the logical OR of two expressions, by ORing each of the 16 bits in the first operand with the corresponding 16 in the second. The four possible combinations of single bits are as follows:

0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1

The result is 0 only if both bits are 0.

It follows that a logical OR is true if either or both of the original expressions were true. It also follows that 380 OR 75 is 383, though verifying this by finding the binary arithmetic forms of 380 and 75 is tedious.

Note:
Exclusive-OR (EOR) is useful. A EOR B = A OR B AND NOT (A AND B).

Examples:
1. **560 IF (A<1) OR (A>20) THEN PRINT "OUT OF RANGE"**
   This is a typical validation test; A must be 1 to 20.
2. **POKE 328,PEEK(328) OR 32**
   This sets bit 5 of location 328 to 1, whether or not it was 1 before, leaving the other bits unaltered. OR can set bits high; AND can set them low.

PEEK

Type: Numeric function

Syntax: PEEK (numeric expression). The expression must evaluate to 0–65535; the value returned is 0–255.

Modes: Direct and program modes are both valid.

Token: $C2 (194)

Abbreviated entry: P SHIFT-E

Purpose: Returns the decimal value of the contents of a memory location. PEEK allows BASIC programs and their variables and pointers to be examined. It also opens up other features like ML programs, BASIC's interpreter, hardware registers, and so on.

Note:
PEEK and POKE are unusual in that they can be replaced by very simple machine language. Chapter 17 has ML routines to PEEK joystick values, which are far faster than BASIC.

Examples:

1. **PRINT CHR$(34);: FOR J=4096 TO 4195: PRINT CHR$(PEEK(J));: NEXT**
   Prints 100 characters (PEEKed from the start of unexpanded VIC BASIC) to the screen. The quote mark is an attempt to prevent problems such as spurious control characters clearing the screen.
2. **500 IF (PEEK(653) AND 1)=1 THEN PRINT "SHIFT KEY"**
   Tests bit 0 of location 653, which stores SHIFT, Commodore key, and CTRL key indicators.

POKE

Type: Statement

Syntax: POKE numeric expression,numeric expression. This is actually POKE address,byte so the first expression must evaluate to a value in the range 0–65535, and the second expression must evaluate to a value in the range 0–255.

Modes: Direct and program modes are both valid.

Token: $97 (151)

Abbreviated entry: P SHIFT-O

Purpose: POKE writes the byte specified by the second expression into the address given by the first. If the address is a location in ROM or a location at which nothing is connected, POKE has no effect. POKE can store ML into memory from DATA statements, alter BASIC pointers, alter hardware registers, and generally perform very useful functions which need some ML knowledge.

Note:
POKE (like PEEK) can be replaced by simple ML. POKEing to the screen in ML is an ideal introduction to machine language. For more on this, see Chapter 7.

Examples:

1. **POKE 36879,123**
   Changes the screen color by altering a VIC chip register.
2. Chapter 6 has a large number of programs which READ data, then POKE it into memory.
3. **FOR J=0 TO 499: POKE 7680+J, PEEK(4096+J): POKE 38400+J,0: NEXT**
   Puts 500 bytes from the BASIC program and variable storage areas of the unexpanded VIC onto the screen in black. Lowercase mode makes the result clearer.

POS

Type: Numeric function

Syntax: POS (numeric expression). The numeric expression is a dummy, as with FRE.

Modes: Direct and program modes are both valid.

Token: $B9 (185)

Abbreviated entry: None

Purpose: Returns the position of the cursor on its current line as seen by BASIC. Normally POS(0) is 0–87, but some PRINT statements can return values up to 255. POS's usefulness is in practice confined to the screen; it won't work with printers.

Examples:

1. **90 FOR J=1 TO 100: PRINT W$(J−1)" ";**
   92 IF POS(0) + LEN(A$(J))>21 THEN PRINT
   94 NEXT

   Prints the words in array W$ in a neat format, without wraparound to following lines. (Assumes no string longer than 21 characters.)
2. Chapter 9's routine to convert ML into DATA uses POS.

PRINT

Type: Output statement

Syntax: PRINT [expression(s)]. The expression(s) may be any type separated by one or more of the following: SPC (numeric expression), TAB (numeric expression), space, comma, or semicolon. The separator can be omitted if its absence causes no ambiguity.

Modes: Direct and program modes are both valid.

Token: $99 (153)

Abbreviated entry: ?

Purpose: Evaluates and prints string, numeric, and logical expressions to an output device, usually the screen. Punctuation of PRINT partly controls the appearance of the output, which also depends on the graphics set being used. (See PRINT USING, Chapter 6, for a discussion of ML editing of numerals.)

Notes:

1. Built-in graphics: The entire character set can be printed, but {RVS} is necessary to complete the set. Color and other controls are easy to include in strings, either in quote mode or with CHR$. PRINT "{RED} HELLO {BLU}" and PRINT CHR$(28)"HELLO"CHR$(31) are equivalent.

   Because {RVS} is necessary to print some graphics, it's not always easy to convert a picture on the screen into PRINT statements in a program. Reverse characters won't be interpreted if you merely home the cursor and type line numbers followed by ?" and RETURN, so be careful when designing graphics directly on the screen.

Chapter 12 discusses this in detail. Note that SHIFT-Commodore key normally toggles between the two sets of characters, lowercase/uppercase and uppercase/graphics. PRINT CHR$(14) sets lowercase. CHR$(142) sets uppercase, CHR$(8) locks out SHIFT-Commodore key, and CHR$(9) enables SHIFT-Commodore key.

2. User-defined graphics: PRINT operates with ASCII characters. Their appearance isn't relevant, so user-defined characters can be handled by PRINT too. For starters, it's easiest to keep most characters as usual, so LIST will be readable. See Chapter 12 for full details.
3. Punctuation in PRINT statements:

a. Expressions. Numeric expressions can include numbers, TI, ST, π, and so on; string expressions can include TI$.

b. SPC and TAB allow the print position to be altered (within limits).

c. Comma (,) tabulates output into the first or eleventh columns. Try:

PRINT 1,2,3,4,5

d. Semicolon (;) prevents print position from returning to the next line and acts as a neutral separator. Try:

PRINT 1;2;3;:PRINT 4

remembering that numbers are output with a leading space (or − sign) and a trailing space. Often the semicolon isn't needed, as in:

PRINT X$ Y$ "HELLO" N% A

where the interpreter will correctly identify everything.

e. Colon (:) ends the statement, and in the absence of a semicolon moves to the next line. Thus, PRINT: PRINT advances two lines.

f. Spaces (unless in quotes) are skipped, so PRINT X Y;2 4 does the same thing as PRINT XY;24.

Examples:

1. **PRINT X+Y; 124; P*(1+R%/100):REM NUMERIC EXPRESSIONS**

   Prints three numbers on a line. If the first semicolon is omitted, X + Y124 is assumed.
2. **PRINT "HI " NAME$ ", HOW ARE YOU?":REM STRING EXPRESSIONS**

   Prints all output on the same line if there's room; otherwise, the output may be broken over two or more lines.
3. **FOR J=1 TO 20:PRINT J,:NEXT:REM SHOWS USE OF COMMA**

PRINT#

Type: Output statement

Syntax: PRINT# numeric expression [,expression(s)]. There must be no space between PRINT and #. The numeric expression is treated as a file number. The file must be open; and the expression(s) uses a format identical to PRINT.

Modes: Direct and program modes are both valid.

Token: $98 (152)

Abbreviated entry: P SHIFT-R (this includes #). Note that ?# is incorrect.
Purpose: Sends data to an output device, usually printer, tape, disk, or modem.
Notes:

1. Punctuation: The effect of punctuation is identical to PRINT, except for a few cases where the appearance onscreen would have been relevant. For example, a comma in a PRINT statement causes just enough spaces to be printed to align the output in columns, while a comma in a PRINT# statement always causes 11 spaces to be printed without regard for columns. Expressions in TAB or POS are not valid; TAB(X+5) is acceptable with PRINT but not with PRINT#. PRINT#4,X$ writes X$ followed by CHR$(13), but PRINT#4,X$; writes X$ alone. PRINT#128,X$: writes X$ followed by RETURN and linefeed; this feature of files numbered 128 or higher is used with certain non-CBM printers.
2. PRINT# and INPUT#: You can send out strings up to 255 characters long using PRINT#. However, INPUT# cannot handle strings longer than 88 characters, so any string you expect to read back with INPUT# should be held to an 88-character limit.
3. TAB, SPC bug: PRINT#1,SPC(6) generates ?SYNTAX ERROR even though it is syntactically correct. PRINT#4,TAB(5) generates the same message. Interestingly, PRINT#1,X$;TAB(5);Y$ is processed correctly. Use any character—even a null, as in PRINT#1,""SPC(6)—before the first SPC or TAB to avoid this problem. Or you could avoid using these functions in PRINT# statements.
4. PRINT# and CMD: PRINT#4 unlistens file #4's device, while CMD4 leaves it listening. See Chapter 17 for further discussion of printers and modems.

Examples:

1. **OPEN 1,1,1,"TAPE FILE": INPUT X$: PRINT#1,X$: CLOSE 1**
   Opens "TAPE FILE" to tape and writes one string to it. Chapter 14 discusses tape files; Chapter 15 has information on disk files.
2. **100 FOR J=40960 TO 49151: PRINT#1,CHR$(PEEK(J));: NEXT**
   Prints the contents of RAM or ROM in the plug-in area to file#1, perhaps a tape file. Note the semicolon to prevent having a RETURN written after every character (making the file twice as long). The resulting file must be read back with GET#, since it is ML and not designed for INPUT# to handle.

READ

Type: Data input statement
Syntax: READ [variable [,variable ...]]
Modes: Direct and program modes are both valid.
Token: $87 (135)
Abbreviated entry: R SHIFT-E
Purpose: Reads data stored in DATA statements. Each READ fetches one item and assigns it to a variable.

If the type of variable doesn't match the data (for example, DATA "ABC": READ X), ?TYPE MISMATCH is printed when the program runs. ?OUT OF DATA error is given when a READ is encountered after all the data has been read; RESTORE is used to once more read data from the beginning.

Examples:

1. **100 READ X$: IF X$<>"ML ROUTINE" GOTO 100**
   Shows how a batch of data can be found anywhere in the DATA statements. This construction (with RESTORE) allows data to be mixed fairly freely throughout BASIC.
2. **0 READ X: DIM N$(X): FOR J=1 TO X: READ X$(J): NEXT**
   Shows how string variables (for example, words for a word game) can be read into an array, by putting the word count at the start (for example, DATA 2,RED,YELLOW).

REM

Type: Remark statement

Syntax: REM [anything]

Modes: Direct and program modes are both valid.

Token: $8F (143)

Abbreviated entry: None

Purpose: Everything on the BASIC line after REM is ignored. REM allows programs to be commented or notated for future reference. REM statements take space in memory, and a little time to execute, so you may want to remove REMs from finished programs.

Note:
See REM in Chapter 6 for some special effects. Chapter 7 explains how machine language can be stored in REM statements.

Examples:

1. **GOSUB 51000: REM PRINT SCREEN WITH INSTRUCTIONS**
   Shows a typical comment.
2. **70 FOR J=1 TO 1000: REM MAIN LOOP**
   80 A(J)=J*A: NEXT
   Shows poor placing of REM, since the REM statement must be interpreted 1000 times as the loop is executed. Move the REM to line 69 to increase speed.
3. **15998 REM ------------------------------**
   15999 REM PRINT TITLE ON TOP OF PAGE ***
   One way that REMs can be made easy to read in long programs.

RESTORE

Type: Data statement

Syntax: RESTORE

Modes: Direct and program modes are both valid.

Token: $8C (140)

Abbreviated entry: RE SHIFT-S

Purpose: Resets the data pointer so that subsequent READs take data starting from the first DATA statement. NEW, RUN, and CLR all incorporate RESTORE.

Examples:

1. **2000 READX$: IF X$="**" THEN RESTORE: GOTO 2000**

   Reads the same DATA in an endless loop—perhaps to play a tune—where ** is the last data item.

2. **130 RESTORE: FOR L=1 TO 9E9: READ X$: IF X$<>"MLROUTINE1" THEN NEXT**
   140 FOR L=328 TO 336: READ V: POKE L, V: NEXT
   9000 DATA MLROUTINE1, 169, 0, 141, 202, 3, 162, 1, 160, 20

   Shows how data can be labeled to be certain that the correct block is read.

3. **RESTORE: GOTO 100**

   Is a direct-mode command of the sort helpful in testing programs which have DATA. Variable values are kept, but DATA is reread from the start.

RETURN

Type: Statement

Syntax: RETURN

Modes: Direct and program modes are both valid.

Token: $8E (142)

Abbreviated entry: RE SHIFT-T

Purpose: Redirects program flow to the statement immediately after the most recent GOSUB statement. GOSUB and RETURN therefore permit subroutines to be automatically processed without the need to keep a note in the program of the return addresses. See GOSUB for a full account of subroutine processing.

Note:
This command has no connection with the carriage return key.

Example:

```
10 INPUT L: GOSUB 1000: GOTO 10 :REM TEST SUBROUTINE 1000
1000 L=INT(L+.5)
1010 PRINT L: RETURN
```

This example repeatedly inputs a number and calls the subroutine at 1000 to process it. The RETURN causes execution to resume with the GOTO 10 statement.

RIGHT$

Type: String function

Syntax: RIGHT$ (string expression, numeric expression)

Modes: Direct and program modes are both valid.

Token: $C9 (201)

Abbreviated entry: R SHIFT-I (this includes $)

Purpose: Returns a substring made up of the rightmost characters of the original string expression. The numeric expression (which must evaluate to 0–255) is compared with the original string's length, and the smaller determines the substring's length.

Example:

100 PRINT RIGHT$("   "+STR$(N),10)

This is another method for right-justification; each string is padded with leading spaces, for a total length of ten characters.

RND

Type: Numeric function

Syntax: RND (numeric expression)

Modes: Direct and program modes are both valid.

Token: $BB (187)

Abbreviated entry: R SHIFT-N

Purpose: Generates a pseudorandom number in the range 0–1, but excluding these limits. RND can help generate test data, mimic random events in simulations, and introduce unpredictability in games.

Notes:

1. Sign of RND's numeric expression argument:

   a. Positive: In this case, the exact value of the expression is relevant. RND(1) and RND(1234) behave identically. The sequence of numbers generated is always the same, starting with .185564016 after power-on, because RND uses a seed value set during the power-on reset sequence.

   b. Zero: The seed value is taken from VIA timers. The result is more truly random, although short ML loops may show repetitiveness.

   c. Negative: The random number generator is reseeded, with a value dependent on the argument. RND(−1) is always 2.99E−8, for example. Chapter 9 has information on RND and explains why negative integers yield very small seed values.

   During development of a program using random numbers, start with 0 X=RND(−1.23) to seed a fixed value, then use RND(1) during the program, which will always follow the same sequence. Thus, the same numbers will be generated each time the program is run, which will facilitate testing. In the final program, use 0 X=RND(0) to start seeding with a random value so that the sequence of numbers will be different each time the program is run.
2. A random number between A and B (excluding exact values of A and B) is given by A + RND(1)*(B−A). For example, RND(1)*2−1 generates numbers between −1 and +1. Integers are equally simple: 1 + INT(RND(1)*10) generates integers from 1 to 10 with equal probability.

Examples:

1. **FOR J=0 TO 3000*RND(1): NEXT**

   Gives a random delay of zero to three seconds.

2. **100 RESTORE: FOR J=0 TO 100*RND(1): READ X$: NEXT**
   Reads a random string, perhaps a word for a language test, from a list of 100 DATA items. X$ holds the random word.
3. **1000 IF RND(1)<.1 THEN PRINT "A VERY GOOD DAY TO YOU"**
   Has a one-in-ten chance of printing its message.
4. **500 INPUT N:DIM D$(N):FOR J=1 TO N:D$(J)=LEFT$("ABCDEFGHIJ", RND(1)*10+1) : NEXT**
   Useful in generating test data, this construction generates an array holding N strings, of random lengths between 1 and 10.
5. **ON RND(1)*4+1 GOSUB 200,300,400,500**
   Selects one of the four subroutines at random.

RUN

Type: Command

Syntax: RUN [linenumber]. The line number must be ASCII numerals; anything else is ignored.

Modes: Direct and program modes are both valid.

Token: $8A (138)

Abbreviated entry: R SHIFT-U

Purpose: Executes a BASIC program in memory, either from its beginning or from a line number. RUN in effect starts with CLR, so variable values are lost; GOTO [line number] has the same effect but retains variable values.

Notes:
1. On the VIC, RUN doesn't LOAD first. The program must be ready in memory.
2. ?SYNTAX ERROR on RUN means start-of-BASIC is not correct. See CLR.
3. Chapter 8 shows how to run BASIC with ML.

Examples:
1. **RUN**
   RUN 1000
   These are two straightforward direct-mode examples.
2. **IF LEFT$(YN$,1)="Y" THEN RUN**
   For use after INPUT "ANOTHER RUN";YN$, this restarts the program from the beginning after Y or YES is typed in.

SAVE

Type: Command

Syntax: Identical to that for LOAD. With tape, however, the interpretation of the final parameter is different: 0 allows a relocating load, so BASIC can work whatever its start address; 1 forces LOAD to put the program where it was saved from; 2 and 3 are like 0 and 1 but additionally write an end-of-tape marker. Chapter 14 has details concerning tape SAVEs, and Chapter 15 discusses disk SAVEs.

Modes: Direct and program modes are both valid.

Token: $94 (148)

Abbreviated entry: S SHIFT-A

Purpose: SAVE writes BASIC in memory to tape or disk, so the program is stored for future use. Programs must be saved to disk by name. Tape programs need not have names, but names can be useful in identifying tape contents.

Machine language, graphics characters, and other continuous blocks of RAM can also be saved. All that's needed is to change two pointers, effectively redefining the position of BASIC. The pointers are in locations 43 and 44 (start) and 45 and 46 (end). See BLOCK SAVE in Chapter 6.

Saving BASIC with its variables is also possible. For example, BASIC followed by integer arrays holds its data in a very compact form. Both variables and BASIC can be saved together, although this is tricky (and strings are best excluded). BASIC followed by graphics definitions can be saved in this way too (see Chapter 12). In each case, only the pointer in 45 and 46 needs to be altered before saving.

Note:

Messages: As with LOAD, standard messages prompt the user when saving to tape. PRESS PLAY AND RECORD ON TAPE is the first. The system can't distinguish these keys from PLAY on its own, so if you're careless you may find you've recorded nothing.

Examples:

1. **SAVE :REM SAVES BASIC TO TAPE WITH NO NAME**
   SAVE "PROG",1,2:REM SAVE TO TAPE WITH END-OF-TAPE MARKER
   Two BASIC SAVEs to tape. (SAVE with forced load address is generally used only with ML, where the correct position is essential.)
2. **SAVE "PROGRAM"+TI$,8**
   Sample SAVE to disk. This adds a clock, so that a version's sequence can be read (provided the clock isn't reset).

SGN

Type: Numeric function

Syntax: SGN (numeric expression)

Modes: Direct and program modes are both valid.

Token: $B4 (180)

Abbreviated entry: S SHIFT-G

Purpose: Computes the sign of a numeric expression and gives −1 if negative, 0 if zero, +1 if positive. This is related to logical expressions and to ABS and the comparison operators. For example, SGN(X−Y) is 0 if X=Y, 1 if X exceeds Y, and −1 if X is less than Y.

Examples:

1. **ON SGN(X)+2 GOTO 400,600,800**
   Branches to 400 if X is negative, 600 if X is 0, and 800 if X is positive. This is useful in FORTRAN conversions.

2. **FOR J=−5 TO 5: PRINT J;SGN(J); SGN(J)*J;SGN(J)*INT(ABS(J)):NEXT**
   Prints several results; the lattermost is like INT but rounds negative numbers up.

SIN

Type: Numeric function

Syntax: SIN (numeric expression)

Modes: Direct and program modes are both valid.

Token: $BF (191)

Abbreviated entry: S SHIFT-I

Purpose: Returns the sine of the numeric expression, which is assumed to be an angle in radians. (Multiply degrees by π/180 to convert.) See ATN for the converse function.

Examples:

1. **FOR J=0 TO 90: PRINT J SIN(J* π/180): NEXT**
   Prints sines of angles from 0 to 90 degrees in one-degree steps.
2. **120 X=A+SIN(A)/2:Y=A+SIN(A)*3/2**
   This line calculates X and Y coordinates of a geometrical shape.

SPC

Type: Output format function

Syntax: SPC (numeric expression). SPC appears only in PRINT and PRINT# statements.

Modes: Direct and program modes are both valid.

Token: $A6 (166)

Abbreviated entry: S SHIFT-P (this includes the left parenthesis).

Purpose: Helps format output to the screen, printer, etc., though the name is misleading. With output to the screen, it is actually the cursor right character rather than the space which is printed (try PRINT SPC(200)"HI!"). However, with any other device, spaces are output since the cursor right character is nonstandard. TAB is similar, except that it tabulates to a particular position, rather than moving a fixed number of positions.

Examples:

1. **100 PRINT "{HOME}": FOR J=0 TO 21: PRINT "X" SPC(20) "X": NEXT**
   Prints a border down each side of the screen, without disturbing any other screen characters.
2. **90 OPEN 1,3: CMD 3**
   Add this line to the previous one; note how an open file causes spaces, not cursor rights, to be output.

ST

Type: Reserved variable

Syntax: ST is treated like a numeric variable, except that no value can be assigned to ST. (For example, X=ST is correct, but ST=X is not allowed.)

Modes: Direct and program modes are both valid.

Token: Not applicable

Abbreviated entry: Not applicable

Purpose: Indicates the status of the system after any input or output operation to tape, disk, or other peripheral. ST is set to zero before GET, INPUT, PRINT and CMD, GET#, INPUT# and PRINT#, so ST is rather ephemeral. Where it is used, it should be used after every command. Note that VIC doesn't handle ST correctly with modems; use PEEK(663) in place of ST in such cases.

ST is a compromise method of signaling errors to BASIC without stopping it. It can often be ignored. Table 3-1 shows the meaning of different values of ST for different devices. Where more than one error occurs, they are ORed together; ST=66 combines 64 and 2. Chapters 14, 15, and 17 provide details on ST with tape, disk, and modems.

Table 3-1. ST Values

ST	Tape Read	Write	Modem	Serial Bus Read	(Disk) Write
1			Parity error		Print time out
2			Frame error	Input time out	
4	Short block on input		RX buffer full		
8	Long block on input				
16	Mismatch on checking	None	CTS missing		
32	Checksum error				
64	End of file on input		DSR missing	End of file (EOI)	
−128	End of tape marker		Break detected	Device Not present	

Note:
ST for tape and disk operations is stored in location 144; RS-232's ST is stored in 663. ST (like TI and TI$) is checked for when a variable is set up; normally, no ST variable exists in RAM, and ST is processed by special routines. Thus ST isn't a tokenized keyword or even a normal variable, and that is why BEST=2 is accepted to mean BE=2, despite the apparent presence of keyword ST.

ST can be used from machine language. See Chapter 8, which deals with Kernal routines, for this and for the associated methods of reading errors from the disk drive.

Examples:

1. **OPEN 11,11: PRINT#11,X$**

   Opens a file to a nonexistent device. This sets ST=−128.

2. **150 INPUT#8,X$: IF ST=64 GOT 1000**
This is a typical end-of-file check when reading data from disk or tape. Line 1000 might be an exit routine to print totals of all the data, then finish.

STOP

Type: Statement

Syntax: STOP

Modes: Direct and program modes are both valid.

Token: $90 (144)

Abbreviated entry: S SHIFT-T

Purpose: Like the STOP key, this command returns the program to READY mode and prints a BREAK message showing the line number at which the program stopped. Like END, STOP can also set breakpoints in BASIC. STOP is generally better than END because the line numbers allow you to insert as many STOPs as you want. See CONT (and GOTO if CONT can't continue) for information on using breakpoints.

Example:
```
80 GET X$: IF X$="" GOTO 100
90 IF X$="*" THEN STOP :REM STOP IF ASTERISK PRESSED
```
Typical of a test for keypress which allows a program to be stopped at a particular point.

SQR

Type: Numeric function

Syntax: SQR (numeric expression)

Modes: Direct and program modes are both valid.

Token: $BA (186)

Abbreviated entry: S SHIFT-Q

Purpose: Calculates the square root of the argument, which must not be negative. This is a special case of the power (up arrow) function. SQR actually works faster than X↑0.5. In addition, it is more familiar to many people.

Examples:
1. **PRINT SQR(2) :REM PRINTS 1.41421356**
2. **X1=(−B +SQR(B*B−4*A*C))/(2*A)**
   X2=(−B −SQR(B*B−4*A*C))/(2*A)
   Both are solutions of the equation $AX^2+BX+C=0$.

STR$

Type: String function

Syntax: STR$ (numeric expression)

Modes: Direct and program modes are both valid.

Token: $C4 (196)

Abbreviated entry: ST SHIFT-R (this includes $)

Purpose: Converts any floating-point number into a string. It formats numbers as PRINT does, so STR$ (10.0) is " 10", with a leading space, and STR$(−123) is "−123".

Examples:

1. **FOR J=1 TO 100: PRINT STR$(J)+".0": NEXT**

   Prints 1 as 1.0, 2 as 2.0, etc.

2. **PRINT "0" + MID$(STR$(X),2)**

   Outputs X as "0.57", etc., where X is between .01 and 1.0. MID$ and STR$ together remove the leading space. Remember that numbers from 0 to .01 are output in exponential notation.

SYS

Type: Statement

Syntax: SYS numeric expression. The expression must evaluate to 0–65535.

Modes: Direct and program modes are both valid.

Token: $9E (158)

Abbreviated entry: S SHIFT-Y

Purpose: Transfers control to machine language at the address specified by the numeric expression. The ML is executed, and control will return to BASIC at the statement after SYS if an RTS instruction (or equivalent) is found. The registers A, X, Y, and SR are loaded from locations 780–783 by SYS, and the contents of these registers are replaced in 780–783 after the subroutine call. This offers a useful way to check short ML routines.

Notes:

1. Chapter 7 introduces machine language; Chapter 6 offers many examples which use SYS. Many of them end with a DATA value of 96, which is the decimal value of RTS. But a jump to a subroutine (DATA 76,xx,xx) also exits, and RTI is usable too (64).
2. Careless SYS calls may crash or corrupt BASIC, and perhaps cause odd anomalies sometime later. Try SYS 55367 as an example; it sets the decimal flag in the chip.
3. BASIC ROM in the VIC resides from 49152 on, so SYS calls to here always have repeatable effects in the VIC.

Examples:

1. **10 SYS PEEK(43)+256*PEEK(44)+30**

   Calls ML stored within BASIC; this form works regardless of what BASIC's starting address was. Chapter 9 explains such techniques in depth.

2. **SYS 64802**
   A ROM routine call which resets the computer as though it had just been switched on.
3. **POKE 780,ASC("$"): SYS 65490: REM KERNAL ROUTINE**
   Puts the dollar sign character in the storage location for A, then calls the Kernal output routine at $FFD2. Kernal routines are explained in depth in Chapter 8. The effect is to print $.

TAB

Type: Output format function

Syntax: TAB(numeric expression). TAB appears only in PRINT or PRINT# statements. There must be no space between the B in TAB and the left parenthesis, and the expression must evaluate to 0–255.

Modes.: Direct and program modes are both valid.

Abbreviated entry: T SHIFT-A (This includes the left parenthesis.)

Purpose: Tabulates PRINT expression to the position on the line (from 0 to 255) specified by the parameter, unless this position is left off an earlier TAB in the same PRINT statement. Like a typewriter's TAB, this function doesn't work backwards.

Notes:
1. TAB is nearly identical to SPC; the difference is that it subtracts its current position on the line from the TAB value, then issues that number of moves right. Its use of cursor-rights and spaces is the same as with SPC.
2. PRINT# has a bug with both TAB and SPC. PRINT#1,SPC(5), for example, gives an error. Use PRINT#1,""SPC(5) or avoid these commands.

Example:

FOR J=1 TO 10: PRINT J;TAB(4)J*J;TAB(10)J*J*J: NEXT
   Produces a tabbed (left-justified) table of squares and cubes.

TAN

Type: Numeric function

Syntax: TAN (numeric expression)

Modes: Direct and program modes are both valid.

Token: $C0 (192)

Abbreviated entry: None

Purpose: Calculates the tangent of any numeric expression, which is assumed to be an angle in radians. The values $\pi/2$ (90 degrees) and other equivalent values cause ?DIVISION BY ZERO and should be tested for to avoid program crashes. TAN divides SIN by COS; it is slower and less accurate than either.

Example:
90 A=ATN(TAN(A))*180/ π
Converts any radian measurement into its equivalent angle from −90 to +90 degrees.

TI and TI$

Type: Reserved variables

Syntax: TI is treated like a numeric variable, and TI$ like a string variable, but TI=X is never allowed. TI$=string expression of length 6 is allowed.

Modes: Direct and program modes are both valid.

Token: Not applicable

Abbreviated entry: Not applicable

Purpose: Contain the numeric and character values of the VIC's realtime clock. The clock is kept running by BASIC in its normal operation. A feature known as an interrupt operates this clock; about every 1/60 second, locations 160–162 are incremented, and their collective value is interpreted for TI and TI$. See Chapter 5 for a discussion of the hardware aspects of this; Chapter 8 discusses its use in programming.

The clock isn't particularly reliable. Tape use makes it run much faster than usual, and programs which disable (turn off) interrupts stop the clock altogether.

TI's maximum value is 518400, the number of sixtieth-seconds in a day. TI is equal to 65536*PEEK(160)+256*PEEK(161)+PEEK(162), where the latter location changes fastest. (Try PEEK(162) in a loop.) The easiest way to change the clock setting is with TI$="101500" (fifteen minutes after ten) or whatever. Note that ML can be used to set and print TI$; Chapter 7 gives full information.

Notes:
1. Like ST, these variables are intercepted by BASIC, not set up in the normal variables space. TIME and TIME$ are treated like TI and TI$, but ANTIC is treated as AN, and its enclosed TI doesn't matter.
2. The interrupt rate can be changed by POKEing different values into locations 37158 and 37159.

Examples:
1. **50 TI$=HH$+MM$+SS$**
   Combines three two-digit strings into TI$.
2. **T$=TI$: PRINT MID$(T$,1,2)+":"+MID$(T$,3,2)+":"+MID$(T$,5,2)**
   Formats TI$ as HH:MM:SS. Note that T$ stores the value in case TI$ changes while the strings are being calculated (for example, from 11:59:59 to 12:00:00).

USR

Type: Statement

Syntax: Numeric variable = USR (numeric expression)

Modes: Direct and program modes are both valid.

Token: $B7 (183)

Abbreviated entry: U SHIFT-S

Purpose: Allows the user to call a function in machine language. This requires a lot of ML knowledge; in BASIC it's nearly always easier to use SYS or a DEF FN expression, and not much slower. Chapter 8's section on calculations has a complete explanation of this function, along with examples.

VAL

Type: Numeric function

Syntax: VAL (string expression)

Modes: Direct and program modes are both valid.

Token: $C5 (197)

Abbreviated entry: V SHIFT-A

Purpose: Converts a string into a number, so calculations can be performed on it. If the conversion does not yield a valid number, as much as possible is converted, and the remainder is ignored with no error message. Valid characters include spaces, signs, numerals, unSHIFTed E, and periods in some combinations. VAL is the converse of STR$.

Examples:

```
PRINT VAL("0.77")              :REM PRINTS .77
PRINT VAL("1.72E3")            :REM PRINTS 1720
PRINT VAL(" +773 DOLLARS"):REM PRINTS 773
IN$="1.2.3": PRINT VAL(IN$)    :REM PRINTS 1.2
PRINT VAL("12"+"."+"01")       :REM PRINTS 12.01
IF VAL(IN$)<0 OR VAL(IN$)>10 THEN PRINT "ERROR"
```

These should be self-explanatory. Note that the last of these tests an input number, avoiding bugs caused by comparing strings with each other.

VERIFY

Type: Command

Syntax: Identical to LOAD

Modes: Direct and program modes are both valid.

Token: $95 (149)

Abbreviated entry: V SHIFT-E

Purpose: Reads and compares a BASIC program (or ML) from disk or tape with the program currently in memory, to verify that a program has been saved correctly. If the two versions are not identical, ?VERIFY ERROR is reported. It can be used in program mode, so you can have a program verify itself.

Because programs may load into different addresses depending on LOAD's parameters, VERIFY should match the parameters of LOAD. Even so, BASIC can generate spurious ?VERIFY ERROR messages, as explained in the notes at the end of this chapter.

VERIFY cannot be used with data files, since these cannot be loaded like programs.

Examples:

1. **SAVE "NEWPROG",8**
   VERIFY "NEWPROG",8

   Saves a program to disk, then verifies it.
2. **10 PRINT "REWIND TO VERIFY"**
   20 GET X$: IF X$="" GOTO 20: REM PRESS KEY
   30 VERIFY

   At the start of a tape program, this verifies in program mode.
3. **LOAD "VERSIONS6",8**
   VERIFY "VER 6",8

   If you have two programs, which you believe may be identical, VERIFY will compare them. A report of OK means that your two programs are indeed identical.

WAIT

Type: Statement

Syntax: WAIT numeric expression, numeric expression [,numeric expression]. The first parameter is an address (in the range 0–65535); the others must be in the range 0–255. The optional parameter defaults to zero.

Modes: Direct and program modes are both valid.

Token: $92 (146)

Abbreviated entry: W SHIFT-A

Purpose: Waits until one or more bits of the memory location are cleared or set in the way specified by the two parameters. The contents of the location are exclusive ORed with the third parameter, then ANDed with the second parameters. If all bits are still zero, the comparison is repeated; otherwise, BASIC continues with the next instruction.

Notes:

1. The location read by WAIT must be one whose contents can change, or the program will wait indefinitely. Chapter 11 has a list of locations which WAIT might use. Note, however, that WAIT commands don't usually transfer to other computers. In fact, they're better replaced, as they always can be, by an equivalent statement using PEEK.

   2. The operation of WAIT can be hard to explain. First, consider exclusive OR. Its truth table is:

```
0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0
```

XOR with 1 flips a bit, while XOR with 0 leaves it unchanged. WAIT address,a,b first XORs the byte in "address" with b. This allows any bit(s) to be flipped. The result is ANDed with *a*, which allows any bit to be turned off. Since a zero result makes WAIT continue, we can select *a* and *b* to respond so that any single bit, changing either to on or off, can exit from WAIT. In the special case WAIT address,a there is no exclusive OR; thus, WAIT address,16 waits until bit 4 is set on. If it never is, WAIT continues forever. This is why WAIT addresses should only be in RAM or in a hardware register which can change.

Examples:

1. **POKE 162,0: WAIT 162,16**
   Waits until the jiffy clock TI counts to 16 (about 1/4 second).
2. **100 POKE 198,0: WAIT 198,1**
   Waits for a keypress (that is, until a character is in the keyboard buffer).
3. **WAIT 37153,2,255**
   Waits for the left SHIFT key to be pressed.

VIC-20 Error Messages

The following list describes VIC-20 BASIC error messages. Disk error messages are handled separately from BASIC; for a list of disk error messages, see Chapter 15.

?BAD SUBSCRIPT

The value given an array subscript is larger than that in the corresponding DIM statement, larger than 10 if the array had not been explicitly dimensioned, or negative. Also given if the wrong number of subscripts is used.

?BREAK ERROR

STOP key pressed before SAVE or LOAD complete.

?CAN'T CONTINUE

The program cannot be continued using CONT because one of the following conditions occurs:

1. It halted through a syntax error, instead of STOP key, STOP, or END.
2. CLR has erased its variables.
3. It was edited after it stopped, erasing variables.
4. A direct-mode error occurred, which the system cannot distinguish from a program error.
5. It has not been run.

?DEVICE NOT PRESENT

One of the following occurs:

1. Printer, disk, or other device does not respond, typically on GET# or INPUT#, because it is unplugged, switched off, nonstandard, unresponsive, or addressed by a wrong device number.
2. An end-of-tape marker was found.

?DIVISION BY ZERO
An attempt has been made to divide by zero, which BASIC does not allow, generally when a denominator underflows to zero. For example, TAN($\pi/2$) contains an implicit division by zero.

?EXTRA IGNORED
Given when the response to INPUT contains more items than asked for by INPUT's parameter list. The extra items are lost. INPUT# behaves identically, but doesn't print the error message. Often caused by inclusion of commas or colons in an input string; avoid this with leading quotes.

?FILE DATA ERROR
The type of data in a file doesn't match the variables to which it is assigned by GET# or INPUT#. Usually happens when INPUT# tries to read a string into a numeric variable.

?FILE NOT FOUND
Disk file or program not present on current disk, or name misspelled. (Tape gives ?DEVICE NOT PRESENT.)

?FILE NOT OPEN
Indicates that the logical file number referred to in a statement has not been opened.

?FILE OPEN
Means that a logical file number referred to in an OPEN statement has already been opened.

?FORMULA TOO COMPLEX
This is given if a string expression contains parenthesized subexpressions and the string descriptor stack (locations \$19–\$21) is exhausted. For example, PRINT "A"+("A"+("A"+"Q")) will give this error.

?ILLEGAL DEVICE NUMBER
One of the following has occurred:
1. A command has been issued to an unacceptable device (for example, SAVE to keyboard or LOAD from screen).
2. Tape buffer has been moved below \$0200.

?ILLEGAL DIRECT
A command requiring the input buffer has been entered in direct mode, typically GET or INPUT, or DEF FN was entered in direct mode.

?ILLEGAL QUANTITY
An expression used as the argument of a function or in a BASIC command is outside the legal range. Examples include attempting a POKE with either parameter neg-

ative, using a logical file number greater than 255, or asking for the square root of a negative number.

I/O ERROR#1 through 9

These are Kernal errors, seen in BASIC only after POKE 157,64. See Chapter 15 for a further discussion.

?MISSING FILE NAME

LOAD from, or SAVE to, disk must have a program name.

?NEXT WITHOUT FOR

Given when the interpreter cannot find a FOR entry on the stack corresponding to the NEXT it has just encountered. This may happen under several conditions:

1. The stack has no FOR entries on it at all, because more NEXTs than FORs have been encountered.
2. The variable in the NEXT statement is misspelled and doesn't match any FOR entries on the stack.
3. The required FOR entry has been flushed from the stack by an incorrectly ordered NEXT in a nested loop.
4. An active GOSUB exists, as in 10 FOR J=1 TO 20: GOSUB 100, followed by 100 NEXT.

?NOT INPUT FILE

Appears when an attempt is made to INPUT or GET from a file that was opened to be written to. For example, files in the write mode cannot be read from.

?NOT OUTPUT FILE

An attempt has been made to write to a disk file that was opened in read mode. Such a file cannot be written to, and the attempt will give this error. A file to the keyboard may be OPENed, and read from, but ?NOT OUTPUT FILE will be given if the attempt is made to write to it, as the keyboard device cannot act as an output device.

?OUT OF DATA

There were no remaining unread DATA items when a READ statement was encountered. Pressing RETURN while READY is on the screen generates this message. RESTORE resets the data pointer.

?OUT OF MEMORY

This means either that the VIC hasn't enough RAM for the program and its variables (especially if dimensioning large arrays or inputting long strings) or that temporary storage on the stack has run out. In the latter case, the stack may be filled with GOSUBs (about 24 maximum), FOR loops (about 10 maximum), or intermediate calculation results, or some combination of the three. Typically, GOSUBs which are never RETURNed cause this problem; see the discussion of POP in Chapter 6.

This message also appears if the end-of-program pointer in 45 and 46 is set (perhaps by LOAD into high memory) higher than the end-of-BASIC pointer in 55 and 56. Reset the pointer or NEW to correct this problem.

?OVERFLOW
The value of a calculation is outside the valid range for floating-point numbers, approximately −1.37E38 to +1.37E38. If a result is actually within the valid range, this error may be avoided by restructuring the computation.

?REDIM'D ARRAY
An attempt has been made to dimension an array that has already been dimensioned. It may have been dimensioned automatically; for example, a reference to X(8) implicitly performs DIM X(10) if that array does not yet exist in memory.

?REDO FROM START
Given when the response to an INPUT statement (but not to INPUT#) contains items of the wrong type. The whole INPUT statement is executed again.

?RETURN WITHOUT GOSUB
A RETURN has been encountered without a GOSUB having first been executed.

?STRING TOO LONG
String expressions must have from 0 to 255 characters; this error is given if a string expression evaluates to a string longer than this. It will also be given if an attempt is made to read a string of 89 or more characters into the input buffer, typically by INPUT#.

?SYNTAX ERROR
Generally, this indicates that a BASIC statement is unacceptable. There are many causes; Chapter 8 includes a routine to help pinpoint them. The VIC-20 anticipates a sequence of statements. If a statement does not start with a keyword or the equivalent of LET, if a variable name isn't ASCII, if a statement isn't terminated with colon or null, or if parentheses, commas, and other symbols are misplaced, ?SYNTAX ERROR results.

This error message may be given after NEW if the first byte of BASIC is nonzero. POKE PEEK(44)*256,0: NEW typically works.

?TOO MANY FILES
Given in response to an OPEN statement, if ten logical files (the maximum permissible number) have already been opened.

?TYPE MISMATCH
Given if the interpreter detects a numeric expression where a string expression is expected, or vice versa.

?UNDEF'D FUNCTION
An undefined function has been used in an expression; it should first have been defined with DEF FN.

?UNDEF'D STATEMENT
The target line number of a GOTO, GOSUB, or RUN does not exist.

?VERIFY ERROR
The program in memory is not identical to the disk or tape file it is being compared against. Spurious VERIFY ERRORs occur if BASIC programs are loaded into VIC with memory expansion different from that which was present when the program was saved; in such cases the link pointers between lines are different, even though the BASIC may be the same.

Chapter 4

Effective Programming in BASIC

Chapter 4

Effective Programming in BASIC

Successful programming requires skill in problem solving and design, not just knowledge of BASIC keywords. This chapter covers the psychology and methodology of designing, writing, and debugging programs, with many useful examples.

Becoming Fluent in BASIC

BASIC is a language, with its own vocabulary and syntax. You need a certain amount of creativity to get good results, just as you need more than a basic knowledge of English words and syntax to write good novels.

But how can a novice programmer develop style and fluency in BASIC? Just as the best way to learn to write is to do a lot of reading and writing, so the best way to learn BASIC is to examine programs and adopt good techniques which suit you. Read keyword descriptions to get a feel for those which are appropriate in certain situations. If you're new to programming, you will certainly find some keywords very hard to grasp, but don't let that worry you. Just start writing without them, and introduce them later on.

When writing programs, you have an advantage over writers who use words. You can experiment freely, and find immediately whether your way of expressing your programming intentions was right or wrong. If your methods fail, no harm is done as long as the experiments are kept separate from serious work.

Another aspect of learning BASIC, in a sense the most important, is developing an appreciation of a machine's capabilities. This is largely a matter of experience. The remainder of this chapter will present you with advice and information, derived from computer industry practice over many years, that is essential if you are to write reliable, easy-to-use programs.

Programs and People

Before considering program design, it is helpful to get an overview of the software market and of attitudes of software producers and users.

There are three types of programs:

Stand-alone, or single, programs have no problems with compatibility or with sharing data. They simply carry out their function or functions. Most games are like this, and "Diet Analysis" (later in this chapter) is an example.

Program systems are less commonly seen associated with the VIC. Such systems generally rely on large numbers of programs and many files of data stored outside the computer. Interactive systems allow information to be put into or taken out of files directly; batch systems store new data on file, after which another program processes it, perhaps merging it into an already-existing file. "Wordscore," a version of Bingo given later in this chapter, is a fairly simple program system which VIC can run.

Programs resembling systems. Single programs with a family resemblance to each other might be classified as midway between single programs and program systems. For example, multiple-choice and other educational program types collectively can be regarded as systems; so can CBM's adventure games.

The important things here are the concepts, not the names. Program systems are likely to be more difficult to program than stand-alone, single programs, since they require validation and checks that are unnecessary in the other types. However, programs resembling systems are often easy to program, provided standardized methods have been developed.

Just as there are types of computer programs, there are types of computer users. Microcomputer owners are generally classifiable as business, science, educational, or personal users, and the VIC can be useful to any of them.

Business. A VIC-20, combined with a disk drive and a printer, is capable of handling reasonable quantities of business data. Mailing or telephone lists, fee schedules, specialized calculations, and word processing illustrate just a few of the business applications that can be managed by the VIC.

Science. The VIC can be used to control external hardware, for example, in process control or in experimental setups. Micros are also quite suitable for calculations and simulations. An equation solver using Newton's method, like the one described in a later chapter, is a typical calculation program; in fact, anything with a definite formula can be done on a computer. Possible subjects range from architectural stress calculations to zoo nutrition, provided the data will fit into the computer.

Education. The continued drop in the price of computing, plus skillful playing on parents' fears, has created something of a boom in computer education. But what makes good educational software?

Multiple-choice question-and-answer programs, categorized by year and subject, make a potentially attractive package; in principle, dozens of programs could be used as refreshers and tests in a range of subjects. Multiple-choice questions are easy to program, since the only reply needed is typically 1, 2, 3, or 4, with no need to interpret a verbal answer.

Single-concept programs are also good possibilities. Children's counting programs and alphabetic recognition programs represent the simplest of this type; good graphics can add a lot of appeal. More advanced examples include foreign language vocabulary and translation tests; economics concepts like price elasticity, supply and demand curves, and marginal costs; musical relationships between frequency and pitch; population simulations; and math techniques and ideas like graph plotting, limits and sums of series, calculus, and simulations of randomness with coins, roulette, and so on.

All of these, and more, are feasible on the VIC.

Personal use. Obviously, the personal category is as vague as it is large. Owners need not restrict themselves to any one category and in fact may use their computers for a variety of applications.

Program Design

This section describes the thought processes necessary in programming and design, with a concrete example of each to give substance to the generalities. Bear in mind that many computer hobbyists use rather ad hoc methods that don't really fit into tidy theoretical schemes, so if your programs are sometimes disjointed and patched up, don't worry about it too much. Most other programmers' are too.

To illustrate the programming process, this section will help you write a program which thinks of a number from 1 to 99, then accepts guesses typed from the keyboard. Where the guess is wrong, it prints TOO LARGE or TOO SMALL, as the case may be. Correct input is rewarded by an encouraging message plus the total number of guesses.

Putting this into BASIC requires four steps, which may be formally written down or simply carried out mentally. The steps are given below, in order.

Understanding the problem. The first step is always to understand the problem. In this example, the problem is quite simple; obviously, however, many computer problems are much more complex.

Expressing the problem in a computerizable way. For a computer to solve a problem, that problem must first be explained in a way that makes sense to the computer. This is where programming experience is essential. For example, if you haven't grasped the idea of computer files, you'll obviously not be able to appreciate their use in storing data. Similarly, if you haven't understood that the computer must count lines of print to know where it is on a page, you won't be able to print titles where you want them. The logical part of programming equips you with methods and tricks to process data, but experience is necessary to appreciate the physical limitations and capabilities of computer systems.

One good way to put a problem into computer terms is to use a flow chart. In this case, the flow chart shown in Figure 4-1 illustrates one approach to the game in a form which can be written as BASIC. Entries in the boxes are shorter than usual to avoid clutter. However, you should be able to trace how N records the number of guesses and how all three possible outcomes of the comparison between X and the current guess are processed.

Flow charts generally use diamonds to indicate options and rectangles to indicate operations. Many other, less common, symbols may also be used, but they are not necessary for this example.

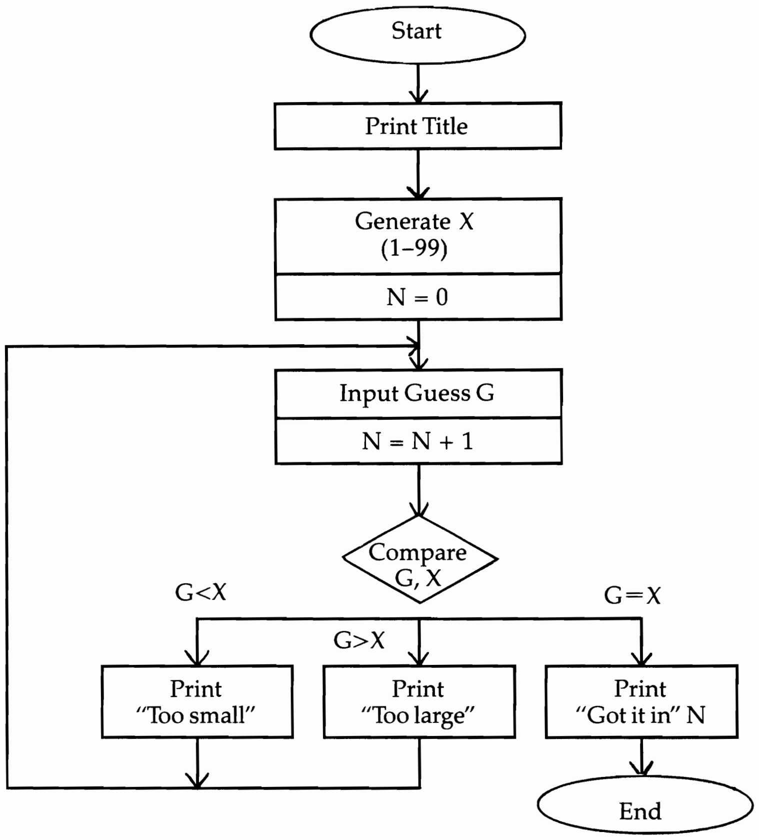
The direction of flow is usually down, with loops and branches arranged clockwise, as in this diagram. Virtually all programs have decision points and loops; if they didn't, they would finish their processing very quickly. Flow charts show these clearly. However, flow charts are hard to modify and take up space, so many people prefer quasi-computer languages (that is, stylized lines of English resembling BASIC programs). The sad fact is that any complex program remains complex regardless of how it is written down.

Writing the problem as BASIC. In some cases you may be able to write the whole BASIC routine at once; if it's long, you'll probably write parts of it and test them individually as modules. This is where past practice is invaluable, not only because of skill in BASIC per se, but because experience suggests efficient ways of getting results.

The specific way in which you solve a problem is called an algorithm. Algorithms are rules with explicit instructions and no exceptions which generate the desired correct results.

Math algorithms can be used by anyone, however little may be understood of the underlying theory. For example, linear programming (solving such problems as finding the cheapest combination of foods which supply all known nutrients) involves long calculations. At a simpler level, an algorithm for chronologically arranging dates in the format YYMMDD is simpler than one for arranging dates of the form MMDDYY. Similarly, a poker hand can be assessed by sorting five cards (ace high), evaluating four differences, and taking account only of zeros, ones, pairs, threes, fours, straights, and others.

Figure 4-1. Number Guessing Game Flow Chart



Other types of algorithms can be used to deal with very complex situations, where a particular rule is found to give good results and is used for lack of anything better. Complex chess openings can be generated with quite simple algorithms, moving to maximize the area under attack and minimize the opponent's range of replies.

Program 4-1. Guessing Game

```
Ø REM * GUESSING GAME *
1Ø PRINT "{CLR}GUESS MY NUMBER (1-99)": PRINT
2Ø X=INT(RND(1)*99)+1: N=Ø
3Ø INPUT "YOUR GUESS";G: N=N+1
```

```
40 IF G<X THEN PRINT "TOO SMALL": GOTO 30
50 IF G>X THEN PRINT "TOO LARGE": GOTO 30
60 PRINT "GOT IT! IN " N "TRIES"
```

The algorithm used in this guessing game is shown in Program 4-1. As you can see, it is a fairly simple one. Lines 0–30 correspond exactly to the first boxes of the flow chart; after this, because IF allows only two options, the lines cannot exactly match boxes but the logic is identical. Note that line 60 doesn't need to test IF G=X, since no other possibility exists.

Testing (and probably improving) the program. There are usually ways to improve a program's design. In this example, you could add line 70 FOR J=1 TO 3000: NEXT: GOTO 10, replacing the box END by a delay loop and a branch back to PRINT TITLE. Values could be checked to make sure they are in the correct range (and that they are integers), or the screen layout could use color.

Testing is difficult. In fact, commercial programmers spend most of their time removing bugs from programs. Ideally, with everything about a program planned in advance, bugs would not appear. But in practice it's impossible to plan for all eventualities.

An Example of a System

Later in this chapter is a program called "Wordscore" which assesses five-letter words on a score-per-letter basis. One form of this game scores words reading across a 5 × 5 grid, where the word BINGO must be included vertically in any one column or diagonally from top left to bottom right. Can VIC help?

The first step is to assess the potential data base of words. A typical dictionary lists about 2000 five-letter words, but not all of these will contain one or more letters from BINGO. VIC can handle this, and the demonstration shows how any values can be assigned to letters A–Z before checking the tape file. Keying in the data is not too long a job. Adding BASIC to select the highest scoring words of form Bxxxx, Ixxxx, and so on (29 relevant formats) isn't difficult, and the conclusion is that VIC could be valuable here.

In general, system planning requires four steps. They are given here in order.

Ask if it is feasible. Can a program system reasonably handle the job? Time may be a problem; sorts, searches, graphics, or tape processing may be too slow to meet your requirements. You may have to write a test program to check these things out. RAM space may also be a problem, particularly with the unexpanded VIC. Can the data coexist with BASIC? Would splitting the program into smaller subprograms help? The list goes on.

Less tangible problem areas might include reliability or recovery of lost data if a tape or disk is lost. A little time spent addressing these questions may save time later on.

Write a solution. Again, you will need to develop suitable algorithms to solve the problem at hand.

Write the program(s). These should be structured so they are easy to understand later. Programs written in modules, each having a single entry and exit point, are likely to be easier to maintain.

Figure 4-2 shows two charts which may be used; one shows a file's structure, while the other illustrates a modular program structured to read that file. Table 4-1, a condition table, lists alternative actions in tabular form and allows complex decisions to be checked more easily than spaghettis of IFs permit.

Figure 4-2. File Structure and Related Program

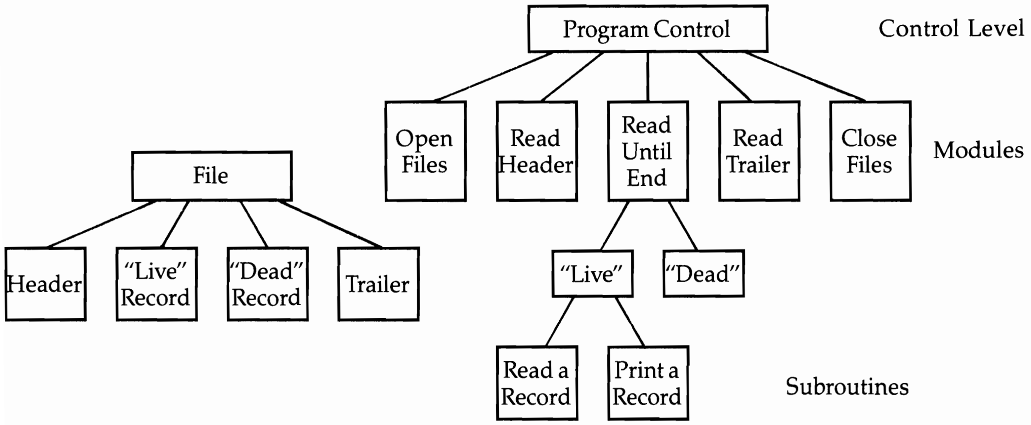


Table 4-1. Condition Table

Conditions:	Stock > reorder level?	Y	Y	Y	N	N	N
	Stock minus stock out > reorder level?	Y	N	N	N	N	N
	Stock out > stock?	N	N	Y	N	Y	N
Actions:	Issue stock	X	X	–	X	–	–
	Issue reorder request	–	X	X	–	–	–
	Part issue stock / increase commitments	–	–	X	–	X	–

Test the system. The final step is to test the system, to see if it functions as desired. See the next section for a detailed discussion of system testing.

Programming Considerations

Unless you're subject to external constraints (for instance, a house style), there is no correct way to program. If it's your computer, you can do what you like. However, this section examines a number of considerations which are important to a program's readability, ease of maintenance and modification, user-friendliness, and so on. In each case you must decide whether inclusion of the feature is worth the additional effort.

Conventions for line numbers, variable names, and REMs. You may choose to number active lines by fives or tens, putting REMs only on lines whose numbers end in 9. If standard subroutines are used, retain the same line numbers in different programs.

REMs help readability, but take up space; you may find it worthwhile to document standard routines for future reference, and delete REMs from programs.

As for variables, remember that BASIC uses global, not local, variables. Be sure that variables can't be accidentally changed. Either list every variable as it's used (and make variable names meaningful), or establish a convention that you will adhere to.

Documentation. Paper documentation could include an operator manual (explaining how to switch on and off, handle and copy disks, and so on); a user manual (explaining file structure, validation methods, and correct sequence of programs); and a system manual (providing a complete reference to the system programs, specifications, file structures, and so on).

Ease of modification. Hard-coded programs use large numbers of constants; soft-coded programs rely more heavily on variables. As a rule, soft coding is easier to modify but more trouble to write.

Error messages. These signal that a mistake has been made, and, more importantly, tell you what it was. Some are built into the computer; you can build others into your programs to make them easier to use.

Easy INPUT of data. A simple INPUT is fine in many cases, but it doesn't give full control. It is better to test for and disable all keys except those actually desired for input; integer input must accept only numerals and not cursor keys, colors, letters, and the like. STOP and RESTORE may need to be disabled (see Chapter 6), and the length of the integer checked. None of these is very difficult, but all take programming time and use up memory.

Prompts are also important. Instructions like ENTER THE AMOUNT or ENTER NEXT ADDRESS make programs much easier to use, but (again) take time to write and occupy memory.

Convenience is another aspect to consider; for instance, a HELP command might also be included. In addition, correction of data can be simplified by printing 10 or 20 values on the screen and asking the user to indicate any changes that might be needed; if so, a minimum number of keystrokes are necessary.

Menus are like elaborate prompts that help users find their own way through programs. A menu displays several options from which the user selects one, typically by pressing a single key. If a menu program stands alone, it has to load and run a new program; alternately, most or all of the options may exist together in memory.

Output formatting. Neat output, particularly of numbers, requires some work. There are a number of ways to format output; for instance, PRINT USING in Chapter 6 gives you a way to print numbers as, say, 2.00 instead of 2.

Subroutines. Standard subroutines allow programs to be developed and tested as modules. It is easier to check isolated parts of BASIC than entire programs; breaking a program into sections also makes it possible for several people to work simultaneously, provided the variables and line numbers are agreed on beforehand.

Subroutines often save space and should always improve clarity. In fact, if your aim is to enhance clarity, it may even make sense to write subroutines which are only called once.

Testing. Thorough testing ideally checks every possible combination of data. Generally, this is impossible. In practice, depending on the program or subroutine, a loop may be used to generate ascending values or check their effect, or RND may be used to make up strings or numerals of the right size. The program "Rounding" later in this chapter includes a loop demonstration; Chapter 6's sorting routines use random data to test sorting.

There are several potential problems that you should be aware of when testing. First, there may be extreme or boundary values which have strange effects; negative numbers, numbers below .01 (which are printed in exponential notation), and the double quotes key are likely to wreck input subroutines unless they're tested for and removed. Second, programming errors may show up only when several events occur at once, making bugs hard to trace because of their apparent random appearances. Third, unconscious bias may influence the choice of test data. For these reasons, commercial systems are tested with data supplied by the user, who also checks that the output is what it should be. However, this can cause problems since the user may not appreciate the importance of testing with obviously wrong data which the system ought to reject.

Validation. Validation is the process of checking to see that data is of the correct type, without necessarily guaranteeing the actual values. For example, although 19/19/86 is an invalid date which should be rejected, 9/9/86 is valid but may be wrong. In its simplest form, validation simply repeats the request for data. More sophisticated checking includes error messages.

Debugging BASIC

This section lists common faults in BASIC programs. While such a list cannot be exhaustive, it should help pinpoint errors. STOP allows you to set breakpoints at which you can check variables' values; PRINT allows you to check key variables while BASIC runs.

Syntax errors. These occur when VIC finds something which isn't BASIC, and therefore can't be understood. Chapter 8 contains a machine language routine that may help you find the mistake, but generally it's up to you to find where you went wrong.

Runtime errors. Runtime errors occur in BASIC which is syntactically correct, but which is trying to manipulate data that isn't valid. Chapter 3 lists all of these errors. Validation routines which only pass acceptable values are a solution.

Errors in program logic. These errors occur when the programmer makes a mistake; the program may run without errors but do the wrong thing. They can have several causes:

Keyword misunderstanding. Misunderstood keywords can produce statements that don't do what you expect them to do. This is common with logical expressions where parentheses have been omitted.

Variable's value altered by mistake. All BASIC variables are global, not local, and a subroutine which uses J can easily be called without its effect on J being noticed. The same thing will happen if an existing variable name is used over again by mistake; for instance, you may forget that D already means decimal position and use it for dollars instead.

Subroutine may be poorly structured. Errors in subroutine structure may cause program execution to drop through to the following lines.

BASIC pointers may be wrong. Placing graphics definitions and ML at the top of memory requires that certain pointers be changed. If not, the data will be corrupted by strings. BASIC may assume an arrangement of hardware or software (for example, a memory configuration) which may not apply.

FN may have been omitted. Omitting FN will cause a function to be read as an array; DEEK(X) without FN is interpreted DE(X).

There may be loop errors. Systematic errors are usually caused by errors in loops, particularly in the zeroth and final elements in buffers or POKEs, or in boundary values that are incorrectly processed.

Bugs in BASIC itself. BASIC has a number of peculiarities. For instance, ASC of a null character crashes. Several other peculiarities are listed below:

- CLOSE to printer or disk needs PRINT# first.
- FRE is slow if there are very many strings.
- INPUT# gives no error message if it finds extra data.
- PRINT attempts to print anything; a stray period appears as 0, for instance.
- TAB and SPC have a few quirks when used with PRINT# (see Chapter 3).

In addition, numbers are not held with absolute accuracy, as Chapter 6 explains. For example, as Program 4-2 shows, FOR X=0 TO 40 STEP .2 never reaches X=20. After 19.8 the value of X has accumulated enough rounding error to throw the value off by a tiny percentage.

Program 4-2. Rounding Error

```
10 FOR X=0TO40 STEP.2:PRINT X
20 IF X=20 THEN PRINT"X IS NOW EQUAL TO 20"
30 NEXT
```

Hardware bugs. Serious hardware bugs are rare. However, trivial ones may hold you up; having the SHIFT/LOCK key down may cause inputs to be rejected or appear very unusual; SHIFT-C may alter the entire screen appearance. Additionally, a printer may jam, a cassette recorder may need to be demagnetized, or a disk drive may be disconnected or switched off.

Examples in BASIC

The remainder of this chapter presents several examples of BASIC programming. Not only are they useful in themselves, but they also illustrate practical BASIC programming techniques.

Input

Program 4-3 and Program 4-4 illustrate ways to input data. Both examples use GET to build a string IN$. In Program 4-3, line 150 defines the range of acceptable characters; for integer input, change the range to 0–9. Line 140 allows the DELete key to operate; all other special keys are disallowed, except STOP and STOP-RESTORE (which can be disabled if you wish [see Chapter 6]). The POKEs simulate the flashing cursor seen during BASIC's INPUT.

Program 4-3. String and Integer Input

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
Ø REM ***** STRING (AND INTEGER) INPUT{6 SPACES}**
  ***                                        :rem 81
1 REM ***** ROUTINE TO INPUT DATA UNDER YOUR FULL
  {SPACE}CONTROL *****                       :rem 77
2 REM * INDIVIDUAL CHARACTERS ARE X$;       :rem 152
3 REM * STRING BUILDS UP INTO IN$;          :rem 176
4 REM * UPPER/LOWER CASE AND ALL USUAL SYMBOLS;
                                             :rem 6Ø
5 REM * THE DELETE KEY OPERATES, ALLOWING CORRECTI
  ONS                                       :rem 247
6 REM                                        :rem 26
7 REM * SET CURSOR POSITION AT START IF YOU WISH
                                            :rem 11Ø
8 REM TO INPUT NUMERALS,{2 SPACES}LINE 15Ø LIMITS
  {SPACE}ARE "Ø" AND "9"                     :rem 14
9 REM                                        :rem 29
1Ø GOSUB 1ØØ                                :rem 114
11 PRINT: PRINT IN$                         :rem 113
12 GOTO 1Ø                                  :rem 253
1ØØ IN$="":{33 SPACES}:REM SET STRING NULL :rem 35
11Ø POKE 2Ø4,Ø: POKE 2Ø7,Ø{18 SPACES}:REM FLASH CU
    RSOR                                    :rem 123
12Ø GET X$: IF X$="" GOTO 12Ø{15 SPACES}:REM FETCH
     CHARACTER                              :rem 15Ø
13Ø IF X$=CHR$(13) THEN PRINT " ";: POKE 2Ø4,1: RE
    TURN: REM EXIT; NORMAL CURSOR EFFECTS :rem 125
14Ø IF ASC(X$)=2Ø THEN IF LEN(IN$)>Ø THEN IN$=LEFT
    $(IN$,LEN(IN$)-1):GOTO 17Ø:REM DELETE  :rem 81
15Ø IF NOT (X$>=" " AND X$<="Z") GOTO 11Ø
    {3 SPACES}:REM RANGE IS UPPER & LOWER CASES &
    {SPACE}SYMBOLS                          :rem 115
16Ø IN$=IN$+X${3Ø SPACES}:REM ADD CHARACTER TO END
     OF STRING                               :rem 75
17Ø PRINT X$;: GOTO 11Ø:REM ECHO CHARACTER TO SCRE
    EN, THEN CONTINUE                       :rem 2Ø6
```

Decimal input is a bit more complicated, as Program 4-4 illustrates, and extra programming is needed to insure that only one decimal point can be entered. This version disallows any more than two figures after the point. All these features can, of course, be changed, but be sure to test the results carefully.

Program 4-4. Decimal Input

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
Ø REM **** DECIMAL INPUT (INPUTS DECIMALS UNDER FU
  LL CONTROL) ****                           :rem 43
1 REM                                        :rem 21
```

```
2 REM * SET FOR 2 DECIMAL PLACES AT MOST  :rem 196
3 REM                                      :rem 23
10 GOSUB 100                              :rem 114
11 PRINT: PRINT D$                         :rem 30
12 GOTO 10                                :rem 253
100 D$="": D=-1                           :rem 147
110 POKE 204,0: POKE 207,0                 :rem 17
120 GET X$: IF X$="" GOTO 120             :rem 129
130 IF X$=CHR$(13) THEN PRINT " ";: POKE 204,1: RE
    TURN                                   :rem 67
140 IF ASC(X$)=20 THEN IF LEN(D$)>0 THEN D$=LEFT$(
    D$,LEN(D$)-1):D=D-1:{2 SPACES}GOTO 170:rem 145
142 IFASC(X$)=20 GOTO110                   :rem 52
144 IFX$="." THEN FORJ=0 TO LEN(D$):IFMID$(D$,J+1,
    1)<>"."THEN NEXT                      :rem 100
146 IF X$="." AND(J=LEN(D$)+1 ) THEN D=0: GOTO 160
                                          :rem 222
150 IF NOT (X$>="0" AND X$<="9") GOTO 110 :rem 226
152 IF D>=2 GOTO 110                      :rem 227
154 IF D>-1 THEN D=D+1                     :rem 89
160 D$=D$+X$                               :rem 75
170 PRINT X$;: GOTO 110                   :rem 225
```

GET can be used to build strings in any format. Equipment part numbers might be of the form 999XXX (three digits followed by three letters), and a routine to input these simply needs to test for the correct input and ignore anything else. Where a DELete key is allowed, this is a little more difficult. Alternatively, an input string might be accepted and then tested for correct format; if an error is found, the program would loop back for reinput, perhaps with an error message.

INPUT in Chapter 3 explains a few tricks, like forcing quotes after the prompt. Pressing only RETURN in response to the INPUT prompt leaves everything unchanged; so 100 X=50: INPUT "NEW X (OR RETURN=50)"; X allows easy entry where default values exist.

Output

Program 4-5 formats output. Line 20 demonstrates the routine. Lines 105–115 test for an E in the string equivalent of V, and line 120 retains the minus sign so that every eventuality is accounted for. Line 150 controls the length of the string V$ in its processed form.

Program 4-5. Rounding

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
1 REM ***{14 SPACES}ROUNDING SUBROUTINE{22 SPACES}
  ***                                     :rem 135
2 REM * WHEN +VE OR -VE V IS INPUT; V$ RETURNS WIT
  H LENGTH 12,{4 SPACES}*                 :rem 194
3 REM * AND TRUNCATED TO 2 DEC. PLACES (USER CAN C
  HANGE THESE).{3 SPACES}*                 :rem 30
```

```
4 REM * EXPONENT E: BELOW .Ø1 GET Ø.ØØ; HUGE NUMBE
  RS GET OVERFLOW *                              :rem 214
5 REM *{6 SPACES}.ØØ2 IS{2 SPACES}FORMATTED AS "
  {8 SPACES}Ø.ØØ"{17 SPACES}*                     :rem 5
6 REM *{7 SPACES}-.2 IS{2 SPACES}FORMATTED AS "
  {8 SPACES}-.2Ø"{17 SPACES}*                   :rem 21Ø
7 REM *{2 SPACES}1234.567 IS{2 SPACES}FORMATTED AS
   "{5 SPACES}1234.56"{17 SPACES}*              :rem 134
8 REM *{7 SPACES}9E9 IS{2 SPACES}FORMATTED AS "***
   OVERFLOW"{17 SPACES}*                         :rem 51
2Ø FOR V=-2Ø TO 1ØØ: GOSUB 1ØØ: PRINTV; V$: NEXT:
   {SPACE}END:REM DEMONSTRATION                  :rem 75
1ØØ T9$=STR$(V):REM WE'LL USE STRINGS           :rem 211
1Ø5 E9=Ø:FOR J9=1 TO LEN(T9$):IF MID$(T9$,J9,1)="E
        "THEN E9=J9                              :rem 31
11Ø NEXT:IFE9>Ø AND MID$(T9$,E9+1,1)="-"THENT9$="Ø
    .ØØ":GOTO 15Ø:REM UNDERFLOW                 :rem 1Ø2
115 IFE9>Ø AND MID$(T9$,E9+1,1)="+"THENT9$="***OVE
    RFLOW":GOTO 15Ø:REM OVERFLOW                :rem 226
12Ø IF MID$(T9$,2,1)="." THEN T9$=LEFT$(T9$,1)+"Ø"
    +MID$(T9$,2)                                :rem 127
121 REM ADD LEADING ZERO TO (E.G.) .5 OR .123, RET
    AINING '-' WHERE PRESENT                    :rem 253
125 D9=Ø:FORJ9=1TOLEN(T9$):IFMID$(T9$,J9,1)="."THE
    N D9=J9:REM DECIMAL POINT IS                 :rem 59
13Ø NEXT:REM AT POSITION D9,                    :rem 164
135 IF D9=Ø THEN D9=LEN(T9$)+1: T9$=T9$+".":REM OR
     ADD D.P. ON END                            :rem 154
14Ø T9$=T9$+"ØØ":REM ALWAYS ADD ØØ,BUT           :rem 5Ø
145 T9$=LEFT$(T9$,D9+2):REM ONLY TAKE 2 D.P.
                                                :rem 134
15Ø V$= RIGHT$("{12 SPACES}"+T9$,12):REM ADD LEADI
    NG SPACES                                   :rem 137
155 RETURN                                      :rem 123
```

Routines like this are valuable for such purposes as printing invoices, receipts, and reports. Chapter 6 contains a machine language version.

Note that X=INT(100*X+.5)/100 is a quick method for rounding to two decimal places.

Calculations

Diet Analysis. Program 4-6 is an example of a calculation-and-report program, predicting weight change by sex, calorie intake, and activity. Lines 400–450 calculate weight change per week; 16 weeks' results are printed out. The algorithm makes standard assumptions that one pound of fat is equivalent to 3500 calories, and that a fairly constant ratio exists between total weight and static-weight calorie intake.

Program 4-6. Diet Analysis

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 REM ******************************************
   ************                            :rem 75
20 REM *******{2 SPACES}WEIGHT{2 SPACES}LOSS
   {2 SPACES}PREDICTOR{2 SPACES}FOR{2 SPACES}DIETE
   RS{2 SPACES}*******                     :rem 62
30 REM ******************************************
   ************                            :rem 77
40 REM * ASSUMES SPARE FAT/ NOT APPLICABLE DURING
   {SPACE}PREGNANCY **                    :rem 154
100 PRINT "{CLR}"                         :rem 245
110 INPUT "WEIGHT (POUNDS)";P             :rem 227
120 PRINT " INTENDED DAILY"                :rem 34
130 INPUT "CALORIE INTAKE";C              :rem 161
140 PRINT "INACTIVE, FAIRLY, OR"          :rem 121
150 INPUT " VERY ACTIVE-0,1,2";A            :rem 0
160 INPUT "MALE, FEMALE-M,F";S$: S$=LEFT$(S$,1)
                                           :rem 96
170 S=1: IF S$="F" THEN S=.9              :rem 134
200 PRINT "{CLR}" S$ "{2 SPACES}WEIGHT NOW:" P
                                          :rem 247
210 PRINT "CALORIE INTAKE:" C             :rem 156
220 PRINT                                  :rem 33
300 FOR W=0 TO 16                          :rem 72
310 PRINT "WEEK" W INT(P*10)/10           :rem 143
400 FOR J=1 TO 7                           :rem 13
410 M=P*(14.3+A)*S + C/10{6 SPACES}:REM M=CALORIES
     TO MAINTAIN WAIT                     :rem 186
420 D=M-C{22 SPACES}:REM DIFFERENCE BETWEEN M & IN
    TAKE                                  :rem 246
430 DW=D/3500{18 SPACES}:REM DW=WEIGHT CHANGE; FAT
    =3500 CALS/POUND                        :rem 1
440 P=P-DW{21 SPACES}:REM P=PREDICTED WEIGHT AFTER
     1 DAY                                :rem 197
450 NEXT J{21 SPACES}:REM LOOP FOR 1 WEEK'S LOSS/G
    AIN                                   :rem 199
500 NEXT W                                 :rem 43
```

Bills and Coins. Program 4-7 works out the smallest note/coin combination, in dollars and cents, to equal any given amount. Line 60 holds DATA; it can be changed to eliminate $100 bills or to convert to other currencies. Line 130 adds a small correction to each figure so there is no chance of rounding errors.

Program 4-7. Bills and Coins

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
0 REM ** NOTE/COIN ANALYSIS **            :rem 174
1 :                                       :rem 107
```

```
50 REM * READ DATA INTO ARRAY NC(), AND SET UP ARR
   AY FOR QUANTITIES *                            :rem 107
60 DATA 11,100,50,10,5,2,1,.5,.25,.1,.05,.01:REM A
   LL 11 US$ DENOMINATIONS                        :rem 43
70 READ NUMBER OF DENOMS: DIM NC(NU),QU(NU)  :rem 6
80 FOR J=1 TO NU: READ NC(J): NEXT            :rem 72
100 REM * INPUT SALARY DATA *                 :rem 63
110 INPUT "{CLR}# OF EMPLOYEES"; EMPLOYEES: DIM SA
    LARIES OF (EMPLOYEES)                     :rem 83
120 FOR J=1 TO EM: PRINT "EMPLOYEE #"J;     :rem 122
130 ::INPUT SALARY OF (J): SA(J)=SA(J) + NC(NU)/2
                                             :rem 122
140 NEXT                                     :rem 212
200 REM * DETERMINE REQUIRED COMBINATION; START HI
    GH *                                     :rem 228
210 FOR J=1 TO EMPLOYEES                     :rem 136
220 ::FOR K=1 TO NUMBER                       :rem 20
230 :::X=INT(SAL(J)/NC(K)): SAL(J)=SAL(J)-X*NC(K):
     QU(K)=QU(K)+X                           :rem 178
240 ::NEXT K                                 :rem 148
250 NEXT J                                    :rem 32
300 REM * PRINT RESULTS *                    :rem 138
310 PRINT "{CLR} ANALYSIS:"                  :rem 150
320 FOR J=1 TO NU: IF QU(J)=0 THEN 340       :rem 183
330 PRINT QU(J) "OF $" NC(J)                 :rem 141
340 NEXT                                     :rem 214
```

Solving Equations. Program 4-8 employs a math technique to find solutions to equations. Lines 2 and 3 have examples, and the program is set up to perform an interest calculation. It will tell you, for example, that ten payments of $135 to clear a loan for $1000 implies 5.865 percent interest per payment period, a calculation which is ordinarily difficult because the formula assumes the interest rate is known.

Program 4-8. Solving Equations

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
0 REM *** SOLVER                             :rem 109
1 :                                          :rem 107
2 REM *** EXAMPLE:{2 SPACES}DEF FN Y(X)=X*X - 2 SO
  LVES SQR(2)                                 :rem 10
3 REM *** EXAMPLE:{2 SPACES}DEF FN Y(X)=X↑3 + 5*X↑
  2 - 3 SOLVES X↑3+5X↑2=3                    :rem 181
4 REM *** EXAMPLE:{2 SPACES}DEF FN Y(X)=EXP(X*X+X)
  -2 SOLVES EXP(X*X+X)=2                      :rem 94
5 REM *** EXAMPLE HERE NEEDS EXTRA VARIABLES N,S,
  {SPACE}AND P, INPUT BY LINES 11-13.         :rem 98
6 :                                          :rem 112
7 REM *** USES AN 'ITERATIVE' (REPEATING) PROCESS T
  O CONVERGE ONTO RESULT                     :rem 141
8 :                                          :rem 114
```

```
10 DEF FN Y(X) = P*(1-1/(1+X)↑N)/ X - S     :rem 68
11 INPUT "NO. OF PAYMENTS";N                :rem 144
12 INPUT "TOTAL SUM";S                       :rem 62
13 INPUT "EACH PAYMENT IS";P                :rem 142
20 GUESS=.1{2 SPACES}:REM SET GUESS AT 10% PER PAY
   MENT INTERVAL, SAY                       :rem 180
30 DX=1/1024 :REM SMALL INCREMENT WITH NO ROUNDING
    ERROR                                   :rem 104
40 GRADIENT = (FN Y(GUESS+DX) - FN Y(GUESS))/DX
                                            :rem 137
50 GUESS=GUESS - FN Y(GUESS)/GRADIENT        :rem 31
60 IF ABS(GUESS-G1)<.00001 THEN PRINT"SOLUTION=" G
   UESS: END                                :rem 245
70 G1=GUESS: GOTO 40:{2 SPACES}REM PRINT G1 TO WAT
   CH CONSECUTIVE GUESSES                    :rem 30
```

The program lets you input guesses. This can sometimes be important, particularly if a problem has more than one solution. Line 60 controls the precision of the answer; greater precision takes longer.

Approximating Fractions. Program 4-9 calculates fractional approximations of decimal entries. It will tell you, for instance, that π is about 22/7, and that 355/113 is much closer. It also gives approximations of any constants, perhaps for overseas currency conversions.

Program 4-9. Approximating Fractions

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
0 REM ************************************:rem 252
1 REM * FRACTIONS APPROXIMATING A NUMBER *:rem 255
100 REM * INPUT NUMBER; CONVERT IT TO FRACTION *
                                           :rem 20
110 INPUT A: T=A:{2 SPACES}B=1             :rem 89
120 IF ABS(T-INT(T+.0001))>.001 THEN T=T*10: B=INT
    (B*10+.1): GOTO 120                    :rem 98
130 T=INT(T+.1)                            :rem 63
140 DIM A(50),T(50),B(50)                 :rem 192
150 A(1)=INT(T/B): T=T-INT(T/B)*B          :rem 80
200 REM * CALCULATE CONTINUED FRACTION IN A() *
                                          :rem 128
210 X=1                                    :rem 89
220 X=X+1: A(X)=INT(B/T)                   :rem 63
230 B1=T: T=B-A(X)*T: B=B1                :rem 103
240 IF B<>1 AND T<>0 GOTO 220             :rem 176
250 IF X>16 THEN X=16{11 SPACES}:REM FIGURES TOO L
    ARGE AROUND HERE                      :rem 203
300 REM * CALCULATE TOP/BOTTOM APPROXNS * :rem 203
310 T(1)=A(1): B(1)=1                     :rem 214
320 T(2)=A(1)*A(2)+1: B(2)=A(2): REM CALCULATE FIR
    ST 2,                                  :rem 72
```

```
330 FOR J=3 TO X{15 SPACES}:REM USE THEM TO CONTIN
    UE                                      :rem 115
340 T(J)=A(J)*T(J-1) + T(J-2)               :rem 143
350 B(J)=A(J)*B(J-1) + B(J-2)                :rem 90
360 NEXT                                    :rem 216
400 REM * PRINT RESULTS *                   :rem 139
410 FOR J=1 TO X: PRINT T(J)"/"B(J)          :rem 53
420 NEXT                                    :rem 213
```

String Handling

Words are handled by BASIC as strings. This allows constructions like 10 INPUT "NAME";N$: PRINT "HELLO, "N$. Typing tutor programs use the same principle.

At a more advanced level, string handling will let you select any individual characters, using MID$, LEFT$, or RIGHT$ (actually, MID$ is enough). In addition, any combination of characters can be generated with the aid of the string concatenation operator (+). An earlier program (Program 4-5) shows how you can scan a string for the character E; Program 4-10 shows how a numeral can be scanned to replace 0 symbols with O, which many people prefer.

Program 4-10. Replacing Zeros

```
0 REM *** REPLACE ZEROS IN STRING WITH LETTERS O *
  **
10 INPUT N$
20 L=LEN(N$)
30 FOR J=1 TO L: IF MID$(N$,J,1)="0" THEN N$=LEFT$
   (N$,J-1) + "O" + RIGHT$(N$,L-J)
40 NEXT
50 PRINT N$
```

When storage space is short, data compression may be worthwhile. Program 4-11 illustrates how long numerals can be packed into roughly half their normal length, using string-handling techniques.

Program 4-11. Packing Numbers

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 INPUT "NUMBER";NS$                       :rem 254
99 REM ** PACK NUMBER STRING NS$ INTO NP$--------
   {SPACE}***                                :rem 16
100 IF LEN(NS$)<>INT(LEN(NS$)/2) *2 THEN NS$="0"+N
    S$                                       :rem 18
110 NP$="": FOR J=1 TO LEN(NS$) STEP 2      :rem 180
120 NP$ = NP$ + CHR$(VAL(MID$(NS$,J,2))+33): NEXT
                                            :rem 247
199 REM *** UNPACK NP$ INTO NUMBER STRING NS$----
    {SPACE}***                               :rem 90
200 NS$="": FOR J=1 TO LEN(NP$): NI$=STR$(ASC(MID$
    (NP$,J))-33)                             :rem 40
```

```
210 NI$=RIGHT$(NI$,LEN(NI$)-1){14 SPACES}:
    {2 SPACES}REMOVE LEADING SPACE          :rem 22
220 NI$=RIGHT$("00"+NI$,2): NS$=NS$+NI$: NEXT: REM
     MAY ADD LEADING 0(S)                   :rem 93
300 PRINT NS$ " " NP$: GOTO 10              :rem 191
```

Analogous tricks include collecting heterogeneous characters together and selecting from them with MID$. For example, there's no simple connection between color keys and their ASCII values, but C$="{BLK}{WHT}{RED}{CYN}{PUR}{GRN}{BLU}{YEL}" is an eight-character string holding them all. PRINT MID$(C$,J,1) prints the Jth color, where J is 1–8.

Program 4-12, a version of the game Bingo, illustrates a small system that evaluates five-letter words on the basis of the point value of each letter (which may vary between runs).

Program 4-12. Wordscore

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
0 REM **********************                :rem 176
1 REM **{7 SPACES}BINGO *{4 SPACES}**        :rem 86
97 REM ---------------------------           :rem 19
98 REM BUILD FILE OF WORDS ON TAPE          :rem 208
99 REM ---------------------------           :rem 21
100 OPEN 1,1,1,"5-LETTER WORDS"             :rem 223
110 INPUT W$                                :rem 157
120 IF LEN(W$)<>5 GOTO 110                   :rem 71
130 PRINT#1,W$                               :rem 28
140 IF W$<>"END**" GOTO 110                  :rem 83
150 CLOSE 1: END                             :rem 78
197 REM --------------------                  :rem 9
198 REM PUT IN LETTER VALUES                :rem 182
199 REM --------------------                 :rem 11
200 DIM V(26)                               :rem 123
210 FOR J=1 TO 26                            :rem 61
220 PRINT CHR$(64+J);                       :rem 141
230 INPUT " VALUE";V(J)                      :rem 18
240 NEXT                                    :rem 213
297 REM ------------------------------      :rem 159
298 REM READ TAPE & PRINT WORD VALUES       :rem 140
299 REM ------------------------------      :rem 161
300 OPEN 1,1,0,"5-LETTER WORDS"             :rem 224
310 INPUT#1,W$                               :rem 31
320 IF W$="END**" THEN CLOSE1: END           :rem 50
330 PRINT W$;                               :rem 217
340 S=0: FOR J=1 TO 5                         :rem 8
350 L$=MID$(W$,J)                           :rem 133
360 A=ASC(L$) - 64                           :rem 70
```

```
370 S=S+V(A): NEXT                              :rem 9
380 PRINT S                                    :rem 123
390 GOTO 310                                   :rem 105
```

Line 360 converts each letter into a number from 1 (for A) to 26 (for Z), and S is the total for the word which was just read from the file. This shows how MID$ can analyze the individual letters in a word. The first part of the program INPUTs five-letter words (note line 120's check) and writes them to tape, stopping when END** is typed in.

RUN 200 then runs the second phase, in which 26 values corresponding to A–Z are entered. The computer then goes on to read back all the words on file and print values. (DATA statements can be used instead. Just use 230 READ V(J) and add 10000 DATA 20,3,5... or whatever numbers are appropriate.) The program can be refined by categorizing the words in W$ (for example, into those beginning B, I, and so on).

Sorting, Searching, Shuffling

Sorting is commercially important in applications like check processing. Chapter 6 has examples, in both BASIC and ML, of sorting routines for VIC. The main restriction is likely to be VIC's small memory. But whenever some sort of sequential or alphabetical list would be useful, bear in mind that a sort may be valuable.

Searching is necessary when you have a lot of data in memory or on a file, but have no index to directly locate a record. For example, an address-book program might store names and addresses, surname first, so that a printout of all names and addresses isn't a problem. But there's no way to instantly access a given name.

Rather than read through all the names, a typical search method (the binary search) is likely to be fast and effective. It assumes the data is already sorted, hence its inclusion here. The idea is analogous to finding a name in a phone book by opening the book exactly in the middle, comparing the name you want with the name at the center of the open page, and repeating the process with earlier or later halves, depending on whether the target name is before or after the current position.

This is a binary search algorithm:

```
X  Input and validate item to be searched for (NA$ = name item)
   Set N1 and N2 to lowest and highest record numbers
Y  R=INT ((N1+N2)/2): REM CALCULATE NEW MIDPOINT
   Read the appropriate field of record number R, for instance R$
   IF R$=NA$ THEN Z :REM FOUND IT
   IF N1>=N2 THEN PRINT "RECORD NOT ON FILE":GOTO X: REM
   NONEXISTENT
   IF R$>NA$ THEN N2=R-1: GOTO Y: REM REVISE UPPER LIMIT DOWN, OR
   IF N1=R+1: GOTO Y: REM REVISE LOWER LIMIT UP
Z  Continue processing of found record
```

N1 and N2 will converge, sandwiching the correct record number R between them. This is easy to program and converges quite rapidly. In the worst cases, the item at the first or last position takes the most tests to find; Table 4-2 gives approximate average numbers of searches to find an item, for various numbers of data items.

Table 4-2. Average Binary Searches to Locate Data Item

Number of Data Items:	50	100	200	500	1000	2000	4000	9000
Average Number of Searches:	5	6	7	8	9	10	11	12

Shuffling is the converse of sorting. Program 4-13 prints a random card deal to show how shuffling programs can work.

Program 4-13. Random Card Deal

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
99 REM *** SHUFFLE CARDS.{2 SPACES}(REPRESENTED BY
    NUMBERS 1-52){2 SPACES}***                :rem 8
100 DIM S(52): FOR J=1 TO 52: S(J)=J: NEXT:rem 129
110 FOR J=1 TO 51                            :rem 58
120 J%=J+INT(RND(1)*(53-J))                 :rem 176
130 TEMP=S(J): S(J)=S(J%): S(J%)=TEMP        :rem 45
140 NEXT                                    :rem 212
299 REM *** LOOP PRINTS OUT CARDS FROM ARRAY S ***
                                              :rem 9
300 FOR J=1 TO 52                            :rem 60
310 N=S(J)-1                                :rem 107
399 REM{2 SPACES}** CONVERT 0-51Q INTO SUIT; PRINT
     IT WITH COLOR:                         :rem 235
400 S=INT(N/13)                              :rem 65
410 PRINT MID$("{BLK}A{RED}S{RED}Z{BLK}X",S*2+1,2)
    ;                                        :rem 35
499 REM{2 SPACES}** THEN PRINT VALUE OF CARD IN SA
    ME COLOR:                                :rem 60
500 V=N - INT(N/13)*13                       :rem 78
510 IF V=1 THEN PRINT "A, ";:{3 SPACES}GOTO 600
                                            :rem 154
520 IF V=11 THEN PRINT "J, ";: GOTO 600     :rem 213
530 IF V=12 THEN PRINT "Q, ";:GOTO 600      :rem 222
540 IF V=0 THEN PRINT "K, ";:{2 SPACES}GOTO 600
                                            :rem 166
550 PRINT MID$(STR$(V),2,LEN(STR$(V))-1)"{BLK}, ";
                                             :rem 81
600 NEXT                                    :rem 213
```

As you'll see, this program uses a fast-shuffling algorithm. Somewhat slower, but easier to understand, is the routine given in Program 4-14. It puts 1, 2, 3, and so on (to 52) into a random position in an array, printing out each number as it does. If the array position is already occupied, it recalculates and tries again.

Program 4-14. Simple Shuffle

```
100 DIM S(52):FORJ=1TO52
110 C=INT(RND(1)*52)+1:REM RANDOM NUMBER 1 TO 52
120 IF S(C)>0GOTO110:REM IF ALREADY USED, TRY AGAI
    N
130 S(C)=J:PRINTC:NEXT
```

Another shuffling program, "Queens," is listed as Program 4-15. It positions queens on an N × N chessboard so that no queen attacks another queen. Rather than test completely random boards, it retains most of an unsuccessful test and switches an attacking queen at random, producing results quite rapidly. The speed tapers off with larger boards; solving the problem of a 20 × 20 board may take many hours.

Program 4-15. Queens

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
9 REM *** GENERATE RANDOM STARTING POSITIONS FOR B
  OARD                                       :rem 42
10 INPUT "SIZE OF BOARD";N                  :rem 246
20 DIM Q(N)                                  :rem 44
30 FOR J=1 TO N: Q(J)=0: NEXT               :rem 255
40 FOR J=1 TO N                             :rem 244
50 R=1 + INT(RND(1)*N)                      :rem 106
60 IF Q(R)>0 GOTO 50                        :rem 245
70 Q(R)=J                                   :rem 226
80 NEXT                                     :rem 167
99 REM *** TEST BOARD; EXCHANGE IF NOT SUCCESSFUL
   {SPACE}***                               :rem 248
100 FOR J=1 TO N-1                          :rem 127
110 FOR K=J+1 TO N                          :rem 152
120 IF ABS(Q(J)-Q(K)) <> K-J THEN NEXT: NEXT: GOTO
     200                                    :rem 119
130 R=1 + INT(RND(1)*N)                     :rem 153
140 IF R=J OR R=K GOTO 130                   :rem 69
150 TEMP=Q(R): Q(R)=Q(K): Q(K)=TEMP: GOTO 100
                                            :rem 243
199 REM *** PRINT DIAGRAM; GO ON FOR MORE ***
                                            :rem 141
200 FOR J=1 TO N                             :rem 34
210 FOR K=1 TO N                             :rem 36
```

```
220 IF K<>Q(J) THEN PRINT "W";                  :rem 230
230 IF K= Q(J) THEN PRINT "Q̲";                  :rem 164
240 NEXT: PRINT                                 :rem 156
250 NEXT: PRINT                                 :rem 157
300 GOTO 30                                      :rem 47
```

It's sometimes handy to use random numbers to help solve simulation problems. At a simple level, Program 4-16 prints the total of two dice, and also keeps a running score so it can print the average number of throws between 7's. This takes some math skill to do analytically; the answer is that it takes six throws on average to score 7.

Program 4-16. Dice

```
0 REM ** DICE SIMULATION **
10 S=S+1
20 D1%=1 + 6*RND(1)
30 D2%=1 + 6*RND(1)
40 PRINT D1% D2%
50 IF D1%+D2%=7 THEN PRINT "SEVEN";: N=N+1: T=T+S:
   S=0: PRINT T/N
60 GOTO 10
```

Data Structures

BASIC's data structures include files, DATA statements, variables, and RAM storage. Files, which store data externally on tape or disk, aren't limited by available RAM, and are necessary for handling large amounts of data. Disk files are generally more versatile than tape, since several disk files can be accessed at once and search time is far less. Chapters 14 and 15 give details of tape and disk programming respectively.

You've seen program examples using DATA. Obviously DATA cannot be changed in the way variables can.

Simple variables are widely used in BASIC. Chapter 6 explains exactly where they are placed in memory and how their values are stored.

Arrays (subscripted variables) offer a powerful extension to the concept of variables and are worth mastering for many serious applications. They provide a single name for a whole series of strings or numbers, using one or more subscripts to distinguish the separate items (called *elements*).

One-dimensional arrays have a single subscript which may take any value from 0 to the value used in the DIM statement which defined the array (or to 10, if DIM wasn't used). DIM A$(50), N%(100), SX(12) defines three arrays (string, integer, and real number respectively), and space is allotted in memory for them (except for the strings).

These arrays can be visualized as a set of consecutively numbered pigeonholes, each capable of storing one value, and initialized with contents zero. A string array might hold values like this: A$(0)="ZERO", A$(1)="ONE", and so on, so PRINT

A$(J) would print the string at pigeonhole J, provided J fell into the correct range. It might hold strings ready for sorting, so that A$(0), A$(1), and so on would be arranged alphabetically after sorting.

Numeric arrays can store the results of calculations; many of the examples of this section use such arrays. For example, numbers 1 to 52, stored in an array, can represent playing card values; similarly, 1 to 8 in an array can represent the position of queens on a chessboard, where 1–8 denotes the file in which that column's queen is placed. It's often worth setting up and saving tables of calculation results, which can be looked up rather than recalculated. Chapter 13's sound calculations illustrate how this is done.

Array variables are slower than simple variables, because of the processing overhead, but they are more versatile. DIMA$(50) gives control over 51 variables and assigns each a unique name. Without this facility, you'd have to define 51 individual names, and the resulting loss of speed would be significant.

Two-dimensional arrays have two subscripts. DIM C(8,8) defines a number array with room for 81 numbers, which might be used to record a chess position, with positive and negative numbers, in which sign represents color and magnitude represents the type of piece.

Two-dimensional arrays are also valuable for storing data for business reports. For example, sales figures may be available for ten types of items, in 12 different outlets. An array can keep the sets of data distinct; subtotals and overall totals can be conveniently stored in zeroth elements.

Integer arrays which store numbers from −32768 to 32767 are particularly efficient in storing data and can be loaded from disk as a block. It's possible to telescope surprisingly large amounts of data into memory like this, although the programming is likely to be difficult.

Arrays with more than two dimensions can be created, but they aren't used much, probably because of the difficulty of visualizing the data's storage pattern. DIM X(2,2,2) can be pictured as three dimensional tic-tac-toe, with element X(1,1,1) in the center position of the center plane. After this, depiction becomes progressively more complicated. In practice, large numbers of dimensions soon exhaust VIC's memory.

Matrix arithmetic, which manipulates arrays with rules for addition and multiplication, is perfectly feasible on the VIC. Briefly, a matrix is an array of form A(R,C), where R and C indicate the numbers of rows and columns. Its arithmetic rules are important to the solution of simultaneous equations. Complex simultaneous equations, and analogous applications, include predictions in biology and economics. Matrix multiplication is essential to the solution of such problems.

Program 4-17. Matrix Inversion

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
Ø REM ******************************************
                                           :rem 248
1 REM *{8 SPACES}MATRIX INVERSION PROGRAM
  {8 SPACES}*                               :rem 19
2 REM ******************************************
                                           :rem 25Ø
3 :                                        :rem 1Ø9
4 REM * EXAMPLE: MULTIPLY INVERSE BY COLUMN ARRAY
  {SPACE}TO SOLVE SIM. EQUATIONS             :rem 8
5 :                                        :rem 111
1Ø DATA 4: DATA 1,2,3,4,Ø,12,4,Ø,-1,-5,-7,12,Ø,Ø,5
   ,8{2 SPACES}:REM SPECIMEN 4 BY 4 MATRIX:rem 246
2Ø READ N: DIM M(N,N),I(N,N){26 SPACES}:REM SET UP
    MATRIX AND INVERSE                      :rem 71
1ØØ REM ** READ MATRIX VALUES (YOU COULD INPUT THE
    SE); SET UP IDENTITY MATRIX             :rem 99
11Ø FOR Y=1 TO N: FOR X=1 TO N: READ M(X,Y): NEXT
    {SPACE}X,Y :REM READ VALUES             :rem 58
12Ø FOR Y=1 TO N: FOR X=1 TO N: PRINT M(X,Y);: NEX
    T: PRINT: NEXT :REM PRINT MATRIX       :rem 192
13Ø FOR Y=1 TO N: FOR X=1 TO N: I(X,Y)=-(X=Y): NEX
    T X,Y                                  :rem 187
2ØØ REM ** FIRST STAGE: DIAGONAL AND BELOW ALL ONE
    S                                      :rem 153
21Ø FOR X=1 TO N                            :rem 49
215 :FOR Y=X TO N                          :rem 152
22Ø ::D=M(X,Y): IF D=Ø OR D=1 GOTO 255     :rem 166
225 :::FOR K=X TO N                        :rem 255
23Ø ::::M(K,Y)=M(K,Y)/D                      :rem 9
235 :::NEXT K                              :rem 21Ø
24Ø :::FOR K=1 TO N                        :rem 213
245 ::::I(K,Y)=I(K,Y)/D                      :rem 7
25Ø :::NEXT K                              :rem 2Ø7
255 :NEXT Y                                :rem 11Ø
26Ø IF X=N GOTO 4ØØ                        :rem 215
3ØØ REM *** NOW PUT ZEROS BELOW DIAGONAL    :rem 45
31Ø :FOR Y=X+1 TO N                        :rem 24Ø
315 ::IF M(X,Y)=Ø GOTO 35Ø                  :rem 85
32Ø :::FOR K=X TO N                        :rem 251
325 M(K,Y)=M(K,Y)-M(K,X)                    :rem 77
33Ø :::NEXT K                              :rem 2Ø6
335 :::FOR K=1 TO N                        :rem 218
34Ø ::::I(K,Y)=I(K,Y)-I(K,X)                :rem 38
345 :::NEXT K                              :rem 212
35Ø :NEXT Y                                :rem 1Ø6
355 NEXT X                                  :rem 52
```

```
400 REM *** TEST DIAGONAL ELEMENTS ARE ALL 1
                                          :rem 180
410 FOR X=1 TO N: IF M(X,X)=1 THEN NEXT    :rem 82
420 IF X<>N+1 THEN PRINT "NO INVERSE": END:rem 107
500 REM *** FINAL STAGE                   :rem 213
505 FOR X=N TO 2 STEP -1                  :rem 211
510 :FOR Y=X-1 TO 1 STEP -1               :rem 113
515 ::D=M(X,Y)                             :rem 11
520 :::FOR K=X TO N                       :rem 253
525 ::::M(K,Y)=M(K,Y)-M(K,X)*D            :rem 165
530 :::NEXT K                             :rem 208
535 :::FOR K=1 TO N                       :rem 220
540 ::::I(K,Y)=I(K,Y)-I(K,X)*D            :rem 150
545 :::NEXT K                             :rem 214
550 :NEXT Y                               :rem 108
555 NEXT X                                 :rem 54
600 REM ** PRINT INVERSE OF MATRIX        :rem 225
610 FOR Y=1 TO N: FOR X=1 TO N: PRINT I(X,Y);: NEX
    T X: PRINT: NEXT Y                    :rem 241
700 REM ** TEST BY REINVERTING            :rem 247
710 FOR Y=1 TO N: FOR X=1 TO N: M(X,Y)=I(X,Y): NEX
    T X,Y: GOTO 130                       :rem 213
```

There is not sufficient space here for a complete discussion of matrix mathematics, but Program 4-17 illustrates one VIC application. It inverts square matrices fairly rapidly (10 × 10 in four minutes). It transforms matrix M into the identity matrix in four stages; repeating the identical operations on the identity matrix generates the inverse. Sample data is included, and (as a check) the program loops back to re-invert its output.

RAM storage. Data may be POKEd into RAM for future use, or loaded into memory from disk or tape, although this is not (strictly speaking) BASIC. Chapter 6 has many examples.

BASIC data can be treated in the same way, although generally this is only worth doing when integer arrays store data and are to be saved directly to disk or tape (which is far more efficient than writing to a file). Chapter 6 explains block saves, the relevant area being that from PEEK(47)+256*PEEK(48) and PEEK(49)+256*PEEK(50).

Control Structures

Some computer languages offer forms like REPEAT...UNTIL and DO...WHILE, and these can be simulated in BASIC. For instance, routines of the form

```
100 FOR J=0 TO -1 STEP 0
110 ...
120 J= (A=B)
130 NEXT
```

have the same effect as REPEAT...UNTIL A=B, since J becomes −1 only when the logical expression in line 120 sets J true.

IF...THEN...ELSE... is another structure missing from VIC BASIC. ON-GOTO or GOSUB is the closest approximation available. (See ON in Chapter 3.) Where ON isn't suitable, because an expression evaluating to 1, 2, 3... doesn't exist, GOTOs will probably be necessary to process both the THEN and ELSE parts of the program.

Processing Dates

Dates are fairly difficult to handle, but this section offers three routines to help validate them, to compute the day of the week given the date, and to calculate the number of days between two dates. Note that leap years are allowed for, but the years 2000 and 1600 (which don't count as leap years) have not been corrected.

Validation. When D, M, and Y (day, month, and year) are input as one- or two-digit numbers, you can use Program 4-18 to check that the day, month, and year combination is valid. If OK is true, D, M, and Y are acceptable. Line 1000 expects Y to be 85 or 86; you can modify the limits for your own purposes. Line 1010 checks that the day does not exceed 28, 29, 30, or 31, whichever applies to the given month and year.

Program 4-18. Validating Dates

```
1000 OK=M>0 AND M<13 AND D>0 AND Y>84 AND Y<87
1010 OK=OK AND D<32+(M=4 OR M=6 OR M=9 OR M=11)+(M
     =2)*(3+(INT(Y/4)*4=Y))
```

Program 4-19 gives the day of the week, for any date, and Program 4-20 tells you how many days are between two dates. The weekday is found by an algorithm usually called "Zeller's Congruence." Days between dates are calculated by taking the difference between days elapsed from an arbitrary early date to the two requested dates.

Program 4-19. Day of the Week

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
0 REM ********************************************
  ******                                  :rem 72
1 REM *{10 SPACES}FIND DAY OF WEEK FOR ANY DATE
  {9 SPACES}*                             :rem 22
2 REM ********************************************
  ******                                  :rem 74
3 REM * EG. MONTH=3, DAY=12, YEAR=86 FOR MARCH 12
  {SPACE}1986 *                           :rem 90
4 REM ********************************************
  ******                                  :rem 76
10 DATA SUN,MON,TUE,WED,THU,FRI,SAT :C=19  :rem 15
11 REM * C=19 MEANS THIS CENTURY; PUT C=18 FOR 180
   0S *                                  :rem 122
20 FOR J=0 TO 6: READ D$(J): NEXT        :rem 171
30 INPUT "MONTH,DAY,YEAR"; M,D,Y         :rem 161
40 M = M-2: IF M<1 THEN M=M+12: Y=Y-1     :rem 56
50 J = INT(2.6*M - .19) + D + Y + INT(Y/4) + INT(C
   /4) - 2*C                             :rem 233
```

```
60 J = J - INT(J/7)*7                          :rem 177
70 PRINTD$(J)                                  :rem 247
```

Program 4-20. Number of Days Between Two Dates

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
0 REM *** DAYS BETWEEN DATES **                :rem 146
10 DATA 0,31,59,90,120,151,181,212,243,273,304,334
                                               :rem 137
20 DIM D(12): FOR J=1 TO 12: READ D(J): NEXT
                                               :rem 193
100 INPUT "DATE 1 (M,D,Y)"; M,D,Y              :rem 196
110 GOSUB 1000: DX=DE                          :rem 111
120 INPUT "DATE 2 (M,D,Y)"; M,D,Y              :rem 199
130 GOSUB 1000: DY=DE                          :rem 114
200 PRINT DY-DX "DAYS": END                     :rem 11
1000 DE = D + D(M) + 365*Y + INT((Y-1)/4) - ((INT(
     Y/4)*4=Y) AND (M>2))                      :rem 146
1010 RETURN                                    :rem 162
```

Chapter 5

VIC-20 Architecture

Chapter 5

VIC-20 Architecture

Successful use of the VIC-20 demands some knowledge of the machine's hardware and of the way software fits into and uses it. The first half of this chapter deals with memory and the effects of memory expansion—important with the VIC-20 because of the many configurations memory can take. The expansion port is covered thoroughly; topics include the uses of expansion boards and program recovery with a reset switch. The second half details input/output, including the VIC chip and the VIAs. By the end of this chapter you should have an appreciation of the different memory configurations plus a grasp of the VIC chip, and you will be able to start analyzing and manipulating VIC programs.

The chapter includes eleven sections:

Some Basics. Covers bits, bytes, hexadecimal notation, RAM, ROM, and so on.

The Unexpanded VIC. Discusses the hardware memory map and includes a short introduction to the VIC's software features. Also includes a short program to look around in your VIC's memory.

The Expanded VIC. Notes on memory expansion and changes in the memory maps when RAM expansion is added.

ROM Expanders. What they are and how they work.

Using the RESTORE Key.

Power Up. What happens when you turn on your VIC-20.

The Expansion Memory Port. Covers expansion boards, reset switches, battery backup of RAM, using write protection to simulate ROM, and hardware addressing.

The VIC Chip. Discusses interfacing to a TV, and international compatibility of programs.

VIA (Versatile Interface Adapter) Chips. What they are and how they work.

IEEE, RS-232, and Other Standard Ports.

How Commercial Software Is Stored in VIC Memory.

Some Basics

This section is intended for beginners, to show the logic behind bits and bytes and to introduce the related subject of hexadecimal arithmetic.

Bits, Binary Numbers, Bytes

A bit, or *binary digit,* is a single, tiny electronic switch which can be either on or off. It can be pictured as an ordinary switch, which either permits current to pass or doesn't. Or it can be visualized as a spot that holds either a small electrical charge or no charge. Hundreds of thousands of these switches are contained in a VIC-20, inside integrated circuit chips, which are mounted inside the black plastic rectangles with metal legs which you can see inside the VIC-20 and all other microcomputers.

Why does each bit have a choice of only two values? Because electronics technology can reliably detect the difference, even if any given machine has quite large variations from ideal voltages. In principle, three values could be used, making trinary arithmetic relevant, but binary hardware is by now so firmly established that this possibility can be ignored for all practical purposes. There would be little economic sense in introducing hardware based on such novel processes.

All VIC-20 operations use binary processes. In hardware terms this is reflected in the large number of tracks needed to carry data within the VIC-20. The expansion memory port, for example, has 44 separate tracks. Every track, apart from those which supply power or are grounded, is treated by the system as carrying only high or low voltage values. Each additional track roughly doubles the system's potential for information handling.

According to convention, the binary values are assigned numbers (0 and 1), and the systems are then structured so that ordinary arithmetic works correctly. The values aren't actually 0 and 1, but this provides a convenient way of talking about them.

A full understanding of software requires a grasp of the relation between binary numbers and ordinary numbers. Fortunately this is not difficult, although it can look rather forbidding. Binary arithmetic uses a notation of 0's and 1's only. However, it represents ordinary numbers and is merely a different way of writing them, just as MCMLXI is a different way of writing 1961.

Just as a digit's position within a decimal number determines the magnitude of that digit, so that 123 and 1230 mean different things, so the position of 0's and 1's within a binary number determines the value of that number. 10101100 is different from 00101011, with the leftmost number being worth the most. And just as decimal digits increase in value by 10, 100, 1000, 10000 (in other words, by powers of ten) as the digit's position moves left, so binary digits increase in value by 2, 4, 8, 16, 32 (powers of two).

To avoid confusion, binary numbers will be written as a series of 0's and 1's prefaced by a percentage sign (%). This lets us be sure, for example, that the decimal number 10 is not confused with %10, which represents 2 in binary notation.

The word *byte* is a derivation of the word bit. It is supposed to imply a larger, scaled-up version of a bit, and that is more or less what it is: A byte is a collection of eight bits which are dealt with by the system as though they were a single unit. This is a purely hardware matter: IBM invented eight-bit bytes, but other numbers such as four or six bits are in use too.

The 6502 microprocessor which operates your VIC-20 is called an eight-bit chip because it is designed to deal with data eight bits at a time. Its addressing uses sixteen bits, but only by dividing up each address into two sets of eight bits each.

Since each bit can be either on or off, you have 2*2*2*2*2*2*2*2 = 256 potential combinations for each byte. That is the reason that PEEKing any address with PRINT PEEK(*address*) always yields a value between 0 and 255. It also explains why the 16-bit address cannot exceed 65535, which is 256*256−1. There are a total of 65536 memory locations, with addresses ranging from 0 to 65535.

Since binary and decimal are only different notations for the same thing, they must be interchangeable. How can you translate binary into decimal? Consider only eight-bit numbers now, since they are very common in programming. The convention is to number the bits in order of importance, reading from the left, as 7, 6, 5, 4, 3, 2, 1, 0. Figure 5-1 may make this easier to understand.

Figure 5-1. Binary-Decimal Interconversion

Binary-Decimal Number Interconversion

Bit Number:	7	6	5	4	3	2	1	0	
Decimal Value:	128	64	32	16	8	4	2	1(	=2↑ Bit Number)
Sample Bytes:									
0000 0000 =	0	0	0	0	0	0	0	0=	Decimal 0
0000 1111 =	0	0	0	0	1	1	1	1=	Decimal 15
1000 0001 =	1	0	0	0	0	0	0	1=	Decimal 129
1111 1111 =	1	1	1	1	1	1	1	1=	Decimal 255

The diagram shows some binary numbers and their decimal equivalents. Notice that it also demonstrates certain features of the bit pattern of binary numbers. For example, if bit 7 is 1, the number must have a decimal value of 128 or more. If bit 0 is 0, the number must be even because only bit 0 can take the value 1 and make a number odd. If a bit pattern is moved bodily right one position, its value is exactly halved, provided the smallest bit isn't a 1 and therefore lost when the shift takes place. Finally, if a bit pattern is completely inverted (so that all 1's become 0's and all 0's become 1's), the two individual values will always add to 255 (because 255 is %11111111). Observations like these, which are exactly analogous to ordinary base 10 arithmetic, are crucial to the understanding of ML programs.

Hexadecimal Notation

Hexadecimal, or base 16—hex for short—is another notation that is useful when programming. BASIC programmers usually ignore hex, but ML programmers will find it quite useful. It relates directly to the internal design of the computer, and it helps ML programmers and hardware designers because of its direct connections with system design.

Hex uses the ordinary numeral symbols 0 through 9 to represent quantities 0 through 9, and the letters A through F to represent quantities 10 through 15. To avoid confusion with ordinary numbers, hex numbers are preceded by the dollar sign ($) or some other signal such as H. This book uses $, which is by far the most popular. Thus, $1 is a valid hex number; so are $A000, $1234, $21, $ACE, and $BEEF. The decimal equivalents of these numbers are 40960, 4660, 33, 2766, 48879. These can be looked up in tables or calculated with a programmer's calculator or conversion program.

Because the VIC has eight-bit bytes, which can only have values from 0 to 255, it is usual to write bytes in hex using two digits, even if the leading digit is 0. Thus, the range is $00 to $FF. That makes for tidy programs, because the numbers line up neatly. Similarly, since memory addresses can only range from 0 to 65535, they are written as four-digit hex numbers. It is not *necessary* that leading zeros be included; $033C and $33C mean the same thing. It is just that many ML programs are written to expect such an arrangement.

Hex and Decimal Arithmetic

If you are programming in BASIC, you are likely to use decimal rather than hex. But when conversion is needed—for example, to find a SYS address—you must make the conversion. It is usually easiest to convert using a program, a programmer's calculator, or a set of conversion tables. But with practice, transition between decimal and hex as well as addition and subtraction in hex become fairly straightforward.

Decimal numbers use a position convention; the further left the digit, the greater its value. The numeral 1 can mean 10, a 100, or a 1000, depending on where it is located within the number. Similarly, a 1 in hex can mean 16, 256, or 4096.

Using the analogy between base 16 and base 10, we need to remember that hex numerals count up to multiples of 16. Thus $19 plus $1 is $1A, not $20. Similarly, $1234 plus $0F is $1243. Figure 5-2 illustrates some hexadecimal numbers and their decimal equivalents.

Figure 5-2. Hexadecimal-Decimal Number Conversion

Hexadecimal-Decimal Number Interconversion

Decimal Value of Each Unit:	4096	256	16	1		
Sample Hexadecimal Numbers:						
$A0	0	0	A	0	= Decimal	160
$11	0	0	1	1	= Decimal	17
$1000	1	0	0	0	= Decimal	4096
$033C	0	3	3	C	= Decimal	828
$FFFF	F	F	F	F	= Decimal	65535

Memory capacity is rated in K (for *kilobytes*); 1K of memory is 1024 bytes. 32K is 32768 bytes, for example. Note that $1000 is more than 1K, and in fact is exactly 4K. A 4K section of memory is often called a block, and your VIC's entire memory can be regarded as a collection of 16 blocks of 4K each.

In hex notation each block will start with a different hex digit. $0000–$0FFF is the first block, $1000–$1FFF is the second, and so on. $F000–$FFFF is the sixteenth and final block.

It is perfectly possible to program without ever using hex arithmetic. All ML aids could be written to use decimal. But because of its relevance to hardware, hex is almost universally used by ML programmers.

RAM, ROM, and Registers

Each memory address of the unexpanded VIC-20 can correspond to one of three types of hardware: An address can be RAM (Random Access Memory), ROM (Read Only Memory), or a programmable location called a *register* (part of another chip). Also, there may be nothing connected at a particular address. These distinctions are important and deserve thorough explanation.

RAM exists in blocks of consecutive addresses. It is not cost-effective to have a single RAM address, and it is convenient to have consecutive storage for almost all applications. RAM can be written to, not just read. It can hold programs or data, which disappear when the power is turned off. It can be overwritten with new data; if this happens in error, the data is said to have been corrupted, while meaningless data left over from earlier programs is called garbage. All VICs are fitted with RAM from $1000 to $1FFF (4096–8191). Try, for example, POKE 4096,123: PRINT PEEK(4096). It will return 123.

ROM also exists in blocks. Your VIC-20 has ROM in all locations from $C000 to $FFFF (49152–65535). PRINT PEEK(49152) gives a fixed value; a POKE to this location (for example, POKE 49152,0) has no effect. ROM cannot be overwritten, modified, or corrupted.

PROM (Programmable Read Only Memory) resembles ROM. ROMs are manufactured with their programs masked in, and are more economical in very large production runs (as for the VIC's BASIC ROMs). PROMs can be individually programmed by using a higher-than-normal voltage to make the byte pattern permanent. An EPROM (Erasable Programmable Read Only Memory) has a window in the top of the chip's case, allowing the memory contents to be erased by ultraviolet light and then reprogrammed. Most non-Commodore ROM products are produced as EPROMs. The window is covered by a sticky label, and often the package is mounted inside a cartridge, since exposure to sunlight over a long period could partly erase such chips.

There is an intermediate form of memory, as we shall see later, in which data written as RAM can be in effect converted to ROM. This involves use of the RAM chip's write-enable line. Battery backup can be provided so the package can store data when disconnected from the VIC-20, providing an alternative to ROM and EPROM storage.

More than half the memory locations in an unexpanded VIC are unused. An unexpanded VIC has nothing from $0400 to $0FFF, from $2000 to $7FFF, and from $A000 to $BFFF. Try POKEing location 1024 with any value; PRINT PEEK(1024) will return the same value regardless of what value you POKE. Note, however, this is not ROM, but a collection of wires which end at the expansion port and can be converted into RAM or ROM. The PEEK value is always the high byte of the address; for example, PEEKing locations 1024–1279 ($0400–$04FF) returns 4, but PEEKing location 1280 ($0500) returns 5.

VIC memory addresses can also refer to another chip. The VIC-20 has three subsidiary chips: the VIC (Video Interface Chip), which serves primarily to produce a signal, including sound, for a TV or video monitor; and two VIAs (Versatile Interface Adapters), which handle the keyboard, tape, disk, joystick, and the rest of the computer's communications with the outside world. POKEing these locations, and sometimes just PEEKing them, can produce dramatic effects which are impossible with ordinary RAM. The reason, of course, is that these chips have their own functions and don't simply store values like RAM. Programming them is more complicated than storing data in RAM; for example, changing one location's value can alter that of a different location.

Incomplete Addressing

There are a few anomalies in VIC addressing; for instance, we have seen how an address with nothing connected to it gives a PEEK value related to its position in memory. The I/O chips show a related oddity: The 16 bytes of each chip are repeated. The VIC chip registers at $9000–$900F appear to repeat at $9040, $9080, and $90C0. Both VIA chips appear to repeat 12 times; for instance, VIA 1 repeats at $9110, $9150, $9190, $91D0, $9210, and so forth. This is caused by incomplete address decoding. It is assumed that the part of the memory map with the I/O chips won't be used for programming, and it is cheaper to simply omit some address lines. Thus $9000, $9040, $9080, and $90C0 can be regarded as the same address.

In BASIC, this can be useful because some decimal numbers are easier to remember than others. Locations 36879 and 37007 both control the border and background colors, and one is definitely easier to remember. In addition, 37060 can be used for 36868, 37070 for 36878, and 37000 in place of 36872. The VIAs have two alternative starting addresses, 37200 and 37600, which are far easier to recall than 37136 and 37152.

Something resembling incomplete decoding occurs in VIC-20's color RAM, where only four bits (not the usual eight) are distinguished. Location $9400 contains values which change erratically—try PEEK(37888)—but if the part of the byte greater than 16 is ignored, the value is stable, as it has to be to properly represent a color.

Memory in the Unexpanded VIC

What Is a Memory Map?

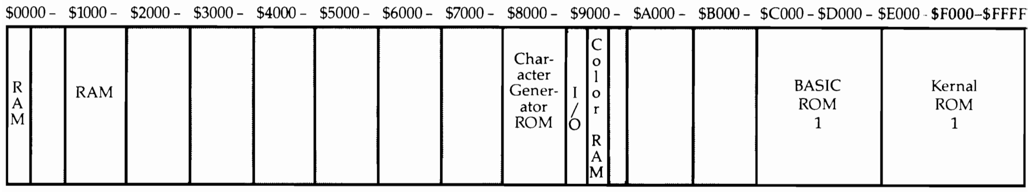
The VIC-20's memory map represents the arrangement of memory as it is seen by—in other words, connected electrically to—the 6502 chip which performs most of VIC-20's computing. Generally, this book uses the phrase memory map to refer to the 6502's map. However, note that every chip which can address memory has its own individual memory map. The VIC chip itself has another less complex memory map which is crucial in understanding graphics, and it is dealt with mainly in Chapter 12.

The 6502 uses 16-bit addressing and can access 2↑16 = 65536 memory locations. As already noted, they are conventionally numbered 0 to 65535 (or, in hexadecimal notation, from $0 to $FFFF).

A Hardware Memory Map for the Unexpanded VIC-20

The memory map shown in Figure 5-3 is divided into 16 equal 4K blocks (4K is 4*1024 = 4096 bytes; this is $1000 in hex). This is a convenient subdivision. Conceptually, each block can be further subdivided into pages of 256 bytes each, with the zero page at $0–$FF, page 1 at $0100–$01FF, page 2 at $0200–$02FF, and so on. This is important because the 6502 treats pages 0 and 1 as special cases.

Figure 5-3. A Hardware Memory Map for the Unexpanded VIC


$0000 - $1000 - $2000 - $3000 - $4000 - $5000 - $6000 - $7000 - $8000 - $9000 - $A000 - $B000 - $C000 - $D000 - $E000 - $F000–$FFFF
RAM
RAM
Character Generator ROM
I/O
Color RAM
BASIC ROM 1
Kernal ROM 1

Here's an explanation of the map:

RAM from $0000 to $03FF (0–1023). This 1K block of RAM is necessary to run BASIC and has many functions, described later.

RAM from $1000 to $1FFF (4096–8191). BASIC programs have to be stored in a single continuous area of memory, so BASIC is stored here in the unexpanded VIC. The screen is also stored in 512 bytes in this region. The screen normally starts at $1E00 in this configuration (unless moved), so BASIC has only 4096−512=3584 bytes available.

ROM from $8000 to $8FFF (32768–36863). The VIC character sets are stored here. There are 128 characters in the uppercase/graphics set and 128 characters in the lower-/uppercase set; moreover, each is duplicated in reverse. Each character definition requires eight bytes (64 bits); so the total amount of memory required for character definitions is 128*2*2*8 = 4096 bytes.

Input/Output chips occupy a total of 48 bytes. The VIC chip uses $9000–$900F, VIA 1 uses $9110–$911F, and VIA 2 uses $9120–$912F.

RAM from $9400 to $97FF (37888–38911). This is color RAM, determining the color and type of character on the screen. Only the lower four bits of each byte in this area are significant.

ROM from $C000 to $FFFF (49152–65535). BASIC and the computer's operating system are stored here. The Kernal ROM holds all the information needed for the computer to interface with the outside world, for instance through the screen and keyboard.

Filling the Gaps in the Memory Map

One drawback of the unexpanded VIC-20 is its tiny memory. VIC comes with RAM from 0 to $03FF and with 4K of RAM for BASIC from $1000 to $1FFF. Because of this, the unexpanded VIC is sometimes called the 5K VIC. However, when the screen has taken its 512 bytes from BASIC's 4096 bytes, only about 3500 bytes remain.

Memory expansion is essential for serious programming with the VIC-20. However, the VIC cannot be expanded simply by inserting RAM chips into sockets inside the machine. Cartridges have to be used; these are simple to use but far more expensive than their component chips.

There are four missing sections of the memory map, each with particular characteristics:

$0400–$0FFF. This 3K space can be occupied by RAM or ROM; RAM is usual, and both the 3K RAM expander and the Super Expander cartridge put RAM here. BASIC can now occupy an extra 3K.

$2000–$7FFF. This large area can be occupied by RAM, ROM, or by a combination of the two. Commodore's 8K RAM expander can be set to occupy any of three blocks; the 16K expander normally puts RAM from $2000 to $5FFF. BASIC can fill whatever extra RAM follows from $1FFF on. Thus, an 8K expander set to occupy $6000–$7FFF adds 8K for ML or data storage, but doesn't normally add to BASIC free memory.

$9800–$9FFF. This 2K area is the odd one out: No Commodore RAM expander fits it, and generally the area is left unused. It is not contiguous to BASIC and is therefore always an isolated area separate from BASIC. Some commercial utilities have ROM here; initialization by SYS 40000 is typical. You'll see later how a 3K RAM expander can be modified to put RAM here, and ML enthusiasts may find this area convenient for storage of routines (like Chapter 6's OLD) which they wish to be protected from BASIC.

$A000–$BFFF. This is the most common location for ROM cartridges; almost all cartridge games load here. In addition, if the proper bytes are present starting at $A000, programs here will automatically begin to run when the VIC is turned on.

As its memory is expanded, the VIC-20 reconfigures itself in several ways. Thus, programs written for the unexpanded VIC may not work when memory is added, even though there is clearly no lack of memory space available. For the moment, attention will focus on the unexpanded VIC, returning later to examine memory expansion in depth.

Looking Inside the VIC-20's Memory

It is possible to use BASIC to PEEK all locations from 0 to 65535. In a sense, this would provide you with a memory map. But it would be relatively meaningless without some further distinctions.

Program 5-1, which works with any VIC-20 regardless of memory configuration, shows you the contents of any section of VIC-20's memory up to 255 bytes long. With it you can select a portion of memory and display its contents in black on top of the screen. Set the display to lowercase mode by pressing SHIFT and the Commodore logo key. This program is a useful investigative tool, worth typing into your VIC and saving for later use. It redisplays the selected portion of memory 60 times each second, giving for all practical purposes a continuous picture of VIC's memory. Use RUN/STOP–RESTORE to turn it off. Note that it will not operate while the cassette is in use, or with some cartridges.

Program 5-1. Looking at Memory in the VIC-20

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
Ø DATA32,115,Ø,24Ø,37,32,138,2Ø5,32,2,215,132,25
  1,133                                   :rem 169
1 DATA252,32,155,215,142,254,2,169,Ø,133,253,173,1
  36,2                                    :rem 132
2 DATA133,254,12Ø,169,1Ø2,141,2Ø,3,169,3,141,21,3,
  96,172                                  :rem 223
3 DATA254,2,136,177,251,145,253,169,Ø,153,Ø,148,15
  3,Ø                                      :rem 9Ø
```

```
4 DATA150,192,0,208,239,76,191,234          :rem 189
10 FOR J=828TO892:READX:POKEJ,X:NEXT         :rem 21
```

Activate Program 5-1 with a statement of the form SYS 828,X,Y where X is the starting address and Y is the number of bytes, like this:

SYS 828,512,88	(displays the input buffer, showing line input)
SYS 828,256,20	(displays numerals as they are formatted)
SYS 828,217,24	(shows the screen link table)
SYS 828,49310,255	(shows some ROM keywords)
SYS 828,631,10	(shows the keyboard buffer)
SYS 828,211,1	(shows the horizontal position of the cursor)
SYS 828,0,255	(displays the entire zero page)
SYS 828,36864,15	(shows the VIC chip contents)
SYS 828,37200,15	(shows the contents of a VIA chip)
SYS 828,160,3	(shows the locations which make up the VIC's clock)
SYS 828,PEEK(43)+256*PEEK(44),80	(shows how part of a BASIC program in memory is stored)

The first example shows how the input buffer operates; nothing happens until RETURN is pressed, then an entire line is input and searched for reserved words. You can see BASIC being converted into one-byte tokens, and you can watch as invalid variable names like CONVERSION have the reserved word found.

The next example shows numbers as they are stored before being printed. It is possible to write an ML program to reformat numbers however you like, for example with a zero before the decimal point (0.5) rather than .5.

The third example shows a table which keeps track of the way VIC's 22-character lines are linked. Let the space bar repeat past the end of a few lines to watch the effect, then scroll the screen and see the links move to match.

Type in SYS 828,631,10 and press RETURN. A series of ten @ symbols should appear at the top left of the screen. Since this program POKEs values directly to the screen, and since @ is the screen representation of a zero byte, this means the keyboard buffer is empty. To show how it operates, type this short program in—0 GET X$: FOR J = 0 TO 1000: NEXT: GOTO 0—and RUN it. The delay loop allows you to queue keypresses in the buffer, and you can see the later keypresses lining up after the earlier ones.

Don't worry if you don't completely understand the examples yet; all will become clear in due time.

Software Landmarks in VIC-20's Memory

To see how VIC's RAM is prepared for BASIC, you need to understand certain software structures. "Looking at Memory," the program you just typed in, can help, and later on we'll present a full memory map showing how software fits into the hardware framework of the VIC-20.

There are several types of software landmarks:

Tables. These contain data, not programs, and have innumerable uses. The screen link table and ROM keywords, revealed by the previous program, are typical. The screen table is in RAM, because it has to be able to change to reflect the screen's organization; the high bit of the last character of each ROM keyword is on, making the word appear in reverse on the screen. File tables, which hold details about each currently open file, are another example.

Buffers. A buffer is a section of RAM reserved for input or output. Buffers include the input buffer, the keyboard buffer, and the 192-byte tape buffer at $033C–$03FB (828–1019), which is important when reading from and writing to tape.

Pointers. Zero page (locations 0–255) contains many pointers in the form of a pair of adjacent bytes. Information about the storage of BASIC is held in this manner. The pair of bytes forms an address in standard low-byte/high-byte format. For example, locations 43 and 44 are the pointer to the beginning of BASIC program storage. On the unexpanded VIC, the normal values held in these locations are 1 ($01) and 16 ($10), indicating that program storage starts at location 1+(16*256)=4097 ($1001).

Vectors. These resemble pointers, as they are also pairs of bytes that constitute addresses. However, while pointers merely hold address information, vectors are used to tell the computer where to find routines to perform certain important operations. Each vector is set up to point to a routine within BASIC or the Kernal operating system when the system is turned on or reset. Altering these values enables many functions of VIC-20 to be modified. The memory examination program described earlier changes the vector to the routine which looks at the keyboard every sixtieth of a second. Sometimes ROM contains vectors; the Kernal itself is a good illustration. A jump table uses a similar approach, except that each address is preceded by an ML JMP instruction and therefore occupies three bytes instead of two.

Flags and temporary storage. These are programming equivalents of a jotted-down note, invariably in RAM. They keep track of a wide variety of events while programs run, ranging from whether the machine is in immediate mode to the position of the cursor on the screen.

Programs. Most of ROM is subdivided into the BASIC interpreter and the Kernal, a collection of many interrelated machine language routines. The only substantial program outside ROM is CHRGET, a routine at locations $73–$8A (115–138) which fetches individual BASIC characters. CHRGET is copied out of ROM into RAM when the system is turned on or reset. Having the routine in RAM is marginally faster than using a ROM routine. It also permits new BASIC keywords to be added to the original stock using a program called a *wedge*, which will be explained later.

Accumulators. Several number storage areas exist in RAM: two floating-point accumulators, where numbers are added, multiplied, and so on ($61–$66 and $69–$6E); a pseudorandom number storage area ($88–$8F); and the realtime clock ($A0–$A2). The memory examination program shows the three bytes of the clock changing, and PRINT PEEK(160)*65536+PEEK(161)*256+PEEK(162) is identical to PRINT TI.

The stack. The stack can't really be understood without knowing machine language, so it is dealt with thoroughly in later chapters. Essentially, it is 256 bytes of RAM from $100 to $1FF (256–511) that are used by the 6502 microprocessor to store temporary information, particularly information relating to subroutines. It is normally best left alone. Short machine language routines can be stored in the lower portion of the stack; if tape is in use, a safe starting location is $0140.

The Expanded VIC-20

Memory Expanders

Commercial RAM expanders are cased in plastic, either as a sealed unit or (as with Commodore's) with a screw fitting enabling the two halves to be separated. Such packaging protects against wear and tear and reduces the risk of static electricity damage, though that is no longer the hazard it was when chip technology was new. It also makes a neater and more robust device. It is not necessary to the proper functioning of the expansion devices.

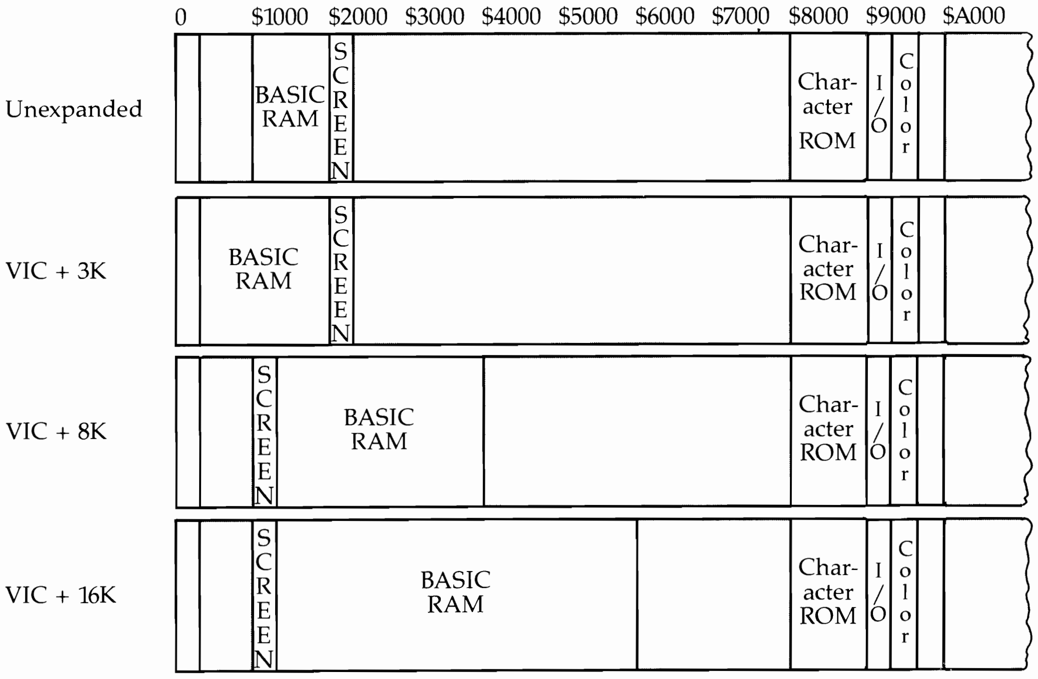
RAM expanders are assembled on printed circuit boards, typically made of fiberglass, with a pattern of metal tracks etched on each side to provide electrical connections. The circuits may also include copyright notices, dates, names of components, pin numbers, ground and power lines, part numbers, and other notes.

ROM cartridges, which use the same address lines that RAM may use, are constructed identically. However, since RAM boards usually contain more chips, ROM cartridges are apt to appear a little disappointing when you open them up.

Understanding Commodore's 3K, 8K, and 16K Expanders

The BASIC memory maps for the unexpanded VIC, and for the VIC with 3K, 8K, and 16K expansion, are shown in Figure 5-4.

Figure 5-4. BASIC Memory Maps for Unexpanded and Expanded VIC



It is helpful to look at the programming side of these expanders before seeing what can be done with simple hardware modifications. RAM expander combinations are listed in Table 5-1. They show start and end of RAM, start and end of BASIC program storage, bytes free, the start of the screen, and the start of color RAM. Color RAM position is related to screen position.

Note that a 3K expander, if used together with another expander, is not used by BASIC (although it can be used to store machine language or data). This is the best the VIC-20 can do, since its chip design prevents it from putting the screen below location $1000. Thus, the absolute maximum length of the BASIC program storage area is from $1200 to $7FFF, about 28,000 bytes. Also note that the *Super Expander* cartridge includes an additional 3K built-in RAM, which extends the low part of memory, so a 3K expander used with the *Super Expander* adds nothing extra.

Table 5-1. Memory Expander Configurations

Cartridge Only	RAM	BASIC	Bytes Free	Screen Start	Color RAM Start
Unexpanded VIC-20	$1000-$1FFF	$1000-$1DFF	3583	$1E00	$9600
VIC-20 + 3K	$0400-$1FFF	$0400-$1DFF	6655	$1E00	$9600
VIC-20 + 8K*	$1000-$3FFF	$1200-$3FFF	11775	$1000	$9400
VIC-20 + 16K	$1000-$5FFF	$1200-$5FFF	19967	$1000	$9400
Multiple Cartridges, with Expansion Board					
VIC-20 + 3K + 8K*	$0400-$3FFF	$1200-$3FFF	11775	$1000	$9400
VIC-20 + 3K + 16K	$0400-$5FFF	$1200-$5FFF	19967	$1000	$9400
VIC-20 + 8K† + 16K	$1000-$7FFF	$1200-$7FFF	28159	$1000	$9400
VIC-20 + 3K + 8K† + 16K	$0400-$7FFF	$1200-$7FFF	28159	$1000	$9400

*8K pack set to $2000-$3FFF
†8K pack set to $6000-$7FFF
Note: "Bytes free" = total RAM usable by BASIC less 512 screen bytes less 1 zero byte at the very start of BASIC.

These are the most useful combinations for BASIC. The last two configurations represent fully expanded VIC-20s. The version which includes the 3K expander can store more machine language routines or other data, but the extra 3K isn't of much use for BASIC except in unusual situations requiring a short BASIC program with a huge number of variables.

The combinations in the table aren't exhaustive, because one or more 8K expanders can be switched to create an area of RAM which is separate from the rest of RAM. Note that 3K and 16K expanders don't have this possibility unless they're modified. For example, an 8K expander set to start at $2000 adds 8K to an unexpanded VIC-20's BASIC storage area, but the same expander set to start at $A000 is independent of BASIC and never corrupted by it. It is perfectly acceptable for a RAM expander at $A000 to be loaded with a program from a ROM cartridge, which it can then run. Alternately, VICMON can be used to write ML programs direct into RAM at $A000, although the contents of this area cannot be directly saved to tape.

If you are mixing BASIC and machine language, it is useful to have reserved RAM unaffected by BASIC, either in a 3K expander with other expansion or in an 8K expander switched to select RAM in a disconnected block.

In practice, strange things may occur when you use a fully expanded VIC, because it is easy to forget that a certain RAM or ROM device is on. For example, programs written for the unexpanded VIC often will not run on expanded VICs, so some expansion memory may need to be turned off and the equipment reset before those programs can be run.

Reconfiguring and Downgrading VIC's Memory

Generally, any program written in VIC BASIC can run in any VIC-20 with enough memory. But it often happens that a VIC won't run a program unless its memory is reconfigured, because some element (such as BASIC's starting address) is in the wrong place. Some tape games, for example, are labeled "No RAM Expansion Necessary" when a more accurate description would be "Will Not Work With Expansion."

The problem arises because such programs assume that one particular configuration is in use, without allowing for possible differences. Often the programmer has not understood that there could be a problem. Any program which POKEs characters to the screen, changes some of the pointers to BASIC, stores its own graphics characters in some fixed location, or relies on the use of supposedly fixed locations within BASIC RAM, is liable not to run in a differently expanded VIC.

There are two distinct problems here. The first is that the VIC-20 may have expansion fitted so that a program won't run, even though the total memory includes, as a subset, what is needed to run the program. The extra memory has caused the difficulty; for example, the screen may be in the wrong place. Utility programs which autostart when the computer is turned on can also cause this sort of interference with memory. To solve this problem, either the extra RAM must be removed or the VIC must be downgraded by software.

In the second case, a program written for a 3K-expanded VIC may not run on one with 8K or 16K expansion, or vice versa. Again, the memory is actually in a different position from what is desired. In such cases you could acquire a 3K expander; alternately, software reconfiguration is worth a try.

Program 5-2 reconfigures the VIC in one of five ways. Downgrading is generally successful, but reconfiguring 8K or 16K expansion to run programs written for 3K expansion is problematical, as the program is simply moved to a new area. For example, BASIC POKEs into locations 55 and 56, which set the top of BASIC, must be removed.

The three reset routines are each useful under different conditions.

Normal reset (as though switching on): SYS 64802
Reset which preserves nonstandard BASIC: SYS 64818
Reset which ignores ROM at $A000 (for example, *Super Expander*), giving normal full memory expansion: POKE 783,181: SYS 64815

Program 5-2. Memory Reconfiguration

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
Ø PRINT "1. UNEXPANDED VIC": PRINT"2. VIC+3K"
                                          :rem 36
1 PRINT "3. VIC+8K":PRINT "4. VIC+16K"    :rem 31
```

```
2 PRINT "5. TRY 3K ON 8K PACK":INPUT V      :rem 66
3 FOR J=1 TO V:READ B,T,S:NEXT             :rem 219
4 POKE 641,Ø:POKE 642,B:POKE 643,Ø:POKE 644,T:REM
  {SPACE}BASIC                              :rem 54
5 POKE648,S:REM START OF SCREEN            :rem 134
7 SYS 64818:REM RESET BASIC WITHOUT TESTING RAM
                                           :rem 118
8 DATA16,3Ø,3Ø,4,3Ø,3Ø:REM DATA HOLDS START OF BAS
  IC                                       :rem 14Ø
9 DATA18,64,16,18,96,16:TOP OF BASIC       :rem 2Ø4
1Ø DATA 32,64,3Ø:REM SCREEN START          :rem 113
```

This program mimics most of the features of VIC's power-on sequence, except the search for an autostart cartridge and for the limits of RAM, which are POKEd in instead.

Modifications (to make the screen start in nonstandard positions, for example) are simple. The program doesn't check to see that the RAM corresponding to its setup actually exists; remember that you can't get something for nothing. In such a case, the hole in the middle will usually keep programs from running. For further information, refer to the section on reconfiguring BASIC in Chapter 6.

ROM Cartridges and How They Work

Programs in ROM are instantly available without the need for loading from tape or disk, hence the value of ROM cartridges. Autostart ROMs are designed so that turning on your VIC automatically runs the program. There's no way to prevent a plugged-in autostart cartridge from running, except for hardware modifications (for example, some expansion boards have switches to disconnect power to the ROM). However, ROM need not be autostarting. *VICMON* (occupying \$6000–\$6FFF) and *Programmer's Aid* (\$7000–\$7FFF), as well as Commodore's *Adventure* programs (\$4000–\$7FFF), print READY when the computer is turned on and need a SYS call before the system takes note of their presence.

Autostart ROMs must start at address \$A000 (40960). The first nine bytes control the working of the ROM:

A000 and A001: Jump address within ROM taken when the computer is turned on.
A002 and A003: Jump address taken when the RESTORE key is pressed.
A004 through A008: Standard sequence of five bytes, \$41, \$30, \$C3, \$C2, and \$CD (65, 48, 195, 194, and 205), representing the characters a0CBM.

It follows that SYS 64802 (which has the same effect as turning the machine on) almost invariably runs an autostart program which wasn't present when the computer was turned on. So does a SYS to the address at the start of ROM, SYS PEEK(40960)+256*PEEK(40961).

Autostart ROM cartridges may contain several ROMs—filling \$A000–\$BFFF, for example—or only a single ROM, generally \$A000–\$AFFF. Obviously, longer programs need more space; most games occupy the full 8K from \$A000 to \$BFFF.

ROM cartridges generally contain machine language programs, which cannot be listed like BASIC and are generally more impenetrable. It is actually possible to store

BASIC in an autostart ROM, using VIC's RAM lower in memory for variables, by manipulating BASIC's pointers.

Some expanders, rather than carrying soldered-in ROMs or EPROMs, have sockets (typically for 24-pin 2K or 4K EPROMs) so the user can select his own ROM software and update or change it as desired. This is primarily of interest to people using utilities, for in addition to storing programs with an obvious single use (such as a game) ROM cartridges can contain programs which supplement VIC in various ways. Commodore's trio of *VICMON, Programmer's Aid,* and *Super Expander* are examples. Others include fast tape operating systems, which add tape commands to decrease program loading and saving times; IEEE adapters, enabling the use of CBM or technical and scientific peripherals; new languages, like Forth; ML assemblers; and graphics packages. Commodore also has a diagnostic ROM board which is used to test the VIC and which some dealers have. The most elaborate devices add a 40- or 80-column display; these add RAM, too, but ignore the VIC chip, generate their own display, and require their own TV connection.

The RESTORE Key

RESTORE is a panic button. When the RUN/STOP key and RESTORE are pressed simultaneously, the system usually returns to the preRUN condition of whatever program was in memory. It is therefore very useful.

There are, however, a few complications. Note that RESTORE doesn't always work. The reason is that it relies on the Non-Maskable Interrupt (NMI) line of the 6502 processor, which is intended to cause the processor to stop its work at any time and attend to the interrupt. However, NMI is not always successful; in particular, if the chip is in an internal loop (called an X2 crash), no interrupt is triggered. Try SYS 49808 when RUN/STOP–RESTORE is ineffective. Later, you'll see how to implement a RESET switch that always works.

Another potential disadvantage is that RESTORE can be disabled. Many programs disable RESTORE as a security device. POKE 37150,2 is one easy way to do so, but note that it is ineffective until the program is actually RUN.

Before summarizing the uses of this key, it is helpful to look at the way it works and what it does. Unlike the rest of the keys, RESTORE is connected directly to one of the VIA interface chips. The VIA is in turn connected to the 6502's NMI line; however, if the VIA is set so that no interrupt occurs, the RESTORE key won't get that far. The POKE just mentioned turns off the interrupt (POKE 37150,130 turns it back on). When an NMI is received by the processor, the 6502 jumps to the address contained in locations $FFFA and $FFFB. In the VIC, this vector points to location $FEA9, the Kernal NMI handler routine. This routine makes an indirect jump through the vector held in locations 792 and 793 ($0318 and $0319). Normally this points straight back to $FEAD, but if the values in 792 and 793 have been corrupted, the system will crash.

Upon return to the interrupt handler routine, the 6502 sorts out whether the interrupt was RESTORE or an RS-232 or tape interrupt. Assuming RESTORE was pressed, the computer checks for the presence of an autostart ROM; if one is found, the computer begins executing the program at the address specified in locations $A002 and $A003. Otherwise, the STOP key is checked and, if pressed, three subroutines are called which reinitialize all indirect vectors, all VIAs, and the VIC chip, before entering BASIC.

You can investigate some of these effects by POKEing 792 and 793 with 60 and 3, then POKEing 828 through 835 with 169, 65, 32, 210, 255, 76, 173, and 255. That redirects RESTORE to PRINT A and then behaves as normal. Pressing RESTORE causes two outputs of A, showing that an interrupt is triggered by a transition in the VIA. Note that STOP–RESTORE terminates this process by resetting locations 792 and 293 to their normal value.

With ordinary BASIC, STOP–RESTORE is a convenient way to return to the normal READY state, especially when the VIC chip or VIAs are being programmed. It is also useful with machine language, but in neither case (especially with ML) is it foolproof. When it does work, everything is left intact except 16 of BASIC's vectors and a few other things (open files are aborted, for example). With autostart ROMs it can be used to reset a game. It could also be used with autostart programs in RAM to interrupt a program while it runs, by changing the contents of $A002 and $A003 so that current variable values can be dumped, the background color changed, a new program loaded, or whatever.

What Happens When VIC-20 Is Turned On

VIC's BASIC requires a fair amount of initial setting up. Therefore, the machine is designed to run through an initialization routine when it is turned on. A timer counts a fixed short delay before grounding a line to the 6502 (the RESET line). This causes the 6502 to start executing whatever program exists at the address specified in bytes $FFFC and $FFFD. That, of course, is ROM in the VIC-20, so the same program is invariably run. What happens is this:

1. The stack pointer is set to $FF, interrupts are disabled, and the decimal flag is cleared.
2. A test is made for an autostart cartridge at $A000; if one is found, the program jumps to the address in $A000 and $A001 without further initialization. A subroutine at $FD3F looks for the five standard bytes. If it finds them, ROM is deemed to be present. It follows that RAM with these bytes will be executed, too, and that ROM without these bytes will be ignored. This test can be imitated from BASIC by SYS 64831. If PEEK(783) AND 2 is 0, there is no ROM; if it's 2, there is.
3. A subroutine at $FD8D (64909) is called, which puts 0's into RAM from $0 to $03FF, excluding the stack ($0100–$01FF), which is not usually considered part of normal RAM. It stores a few values in RAM and then performs an important nondestructive RAM test to find the maximum number of consecutive bytes which it can allocate to BASIC. This test starts at $0400. An unexpanded VIC has no memory there; its RAM starts at $1000. Only a 3K RAM expander can fill the 3K of RAM between $0400 and $0FFF.

   Either way, the start of RAM is noted. The test continues from $1000 to $1FFF, which contains the standard 4K of an unexpanded VIC. (If this RAM is defective, the program enters an infinite loop, leaving a static pattern of meaningless characters on the screen.) The top of RAM is noted, allowing for any 8K or 16K expanders which may be present. At that point, the screen position is determined; unlike the CBM, where the screen always starts at $8000, the VIC chip's flexibility allows the screen to start in a variety of places. Only two of them are used by VIC-20 itself; if RAM is not present at $2000 and above, as with the unexpanded VIC, the screen starts at $1E00. Otherwise, it starts at $1000. Finally, the starting and ending positions of BASIC program storage are fixed to use as much RAM as possible while avoiding the screen.

4. A subroutine at $FD52 (64850) moves a table from ROM into the third page of RAM, from $0314 to $0333. This is a table of two-byte vectors; we'll see later how to make use of them.
5. A subroutine at $FDF9 (65017) initializes both VIA chips.
6. Another subroutine at $E518 (58648) initializes the VIC chip with the 16 values stored in $EDE4–$EDF3 (60900–60915), setting (among other things) the border color to cyan and the background color to white. Next the VIC chip registers pointing to the start of the screen are made to conform to the required screen starting address. The screen start byte usually contains only $1E or $10, but the routine is compatible with other values. Then, keyboard and screen characteristics (keyboard repeat rates, cursor flash rates, a screen link table to keep track of connected lines) are set. The cursor is moved to the home position.
7. Finally, the BASIC interpreter is entered via the address in $C000 and $C001. Interrupts are enabled, some specifically BASIC pointers are set, and the familiar BYTES FREE message appears, followed by READY.

   The main BASIC pointers are important and worth knowing:

   BASIC program storage starts at the address pointed to by $2B and $2C (43 and 44).

   BASIC program storage ends and variables begin at the address pointed to by $2D and $2E (45 and 46).

   Variables end and arrays begin at the address pointed to by $2F and $30 (47 and 48).

   Arrays end at the address pointed to by $31 and $32 (49 and 50).

   BASIC working memory ends at the address pointed to by $37 and $38 (55 and 56).

The contents of $2B–$2C and $37–$38 match the addresses in Table 5-1; for example, PEEK(43)+256*PEEK(44) is the start of BASIC program storage in decimal. BYTES FREE is the difference between the start and end of BASIC. There are also two string pointers which can be ignored; they take care of themselves or can be reset by NEW or CLR.

These pointers are used too:

MEMBOT is $0281 and $0282 (641 and 642).
MEMTOP is $0283 and $0284 (643 and 644).
Screen start page is $0288 (648).

MEMBOT and MEMTOP are stored when the system has found the extent of contiguous RAM. These values aren't used except to decide where to put BASIC and the screen. If there is noncontiguous RAM, the system has no record of it, and it is irrelevant to normal BASIC. By itself, the VIC-20 will set the screen start at one of just two locations, allowing 512 bytes for the screen RAM. (Since a normal screen is

22 × 23, or 506 bytes, there are 6 spare bytes after the screen). However, as you'll see in Chapter 6, the screen position can be set to other values, and the screen size can also be changed within wide limits.

To review, when power is turned on, almost all BASIC RAM is left untouched. Only RAM between $0 and $03FF—excluding the first three bytes of BASIC (which are set to zero) and the stack—is changed. (Consequently, BASIC RAM is left with the garbage present at power-up. After turning on your computer, try PEEKing locations 4099 and above to verify that RAM hasn't been zeroed.)

This has two consequences. First, SYS 64802, which imitates the reset function, acts as though VIC had been switched on but leaves any RAM program unchanged, except that BASIC NEW is in effect performed. See OLD in Chapter 6 for a method to recover BASIC after SYS 64802 has reset it. Second, a hardware reset switch—see next section—can be used to investigate any program above $0400 in memory or within the stack area ($0100–$01FF).

The Expansion Port

This is the large port at the left of the VIC, as seen from the back. It has 44 connections—22 on each side—two of which are unused. One track (pin 21, located second from the right on the top row) carries the +5 volt power supply from VIC to the additional RAM or ROM; such asymmetry, plus the fact that the edge connections are rather close, makes it risky to insert or remove these when the power is on.

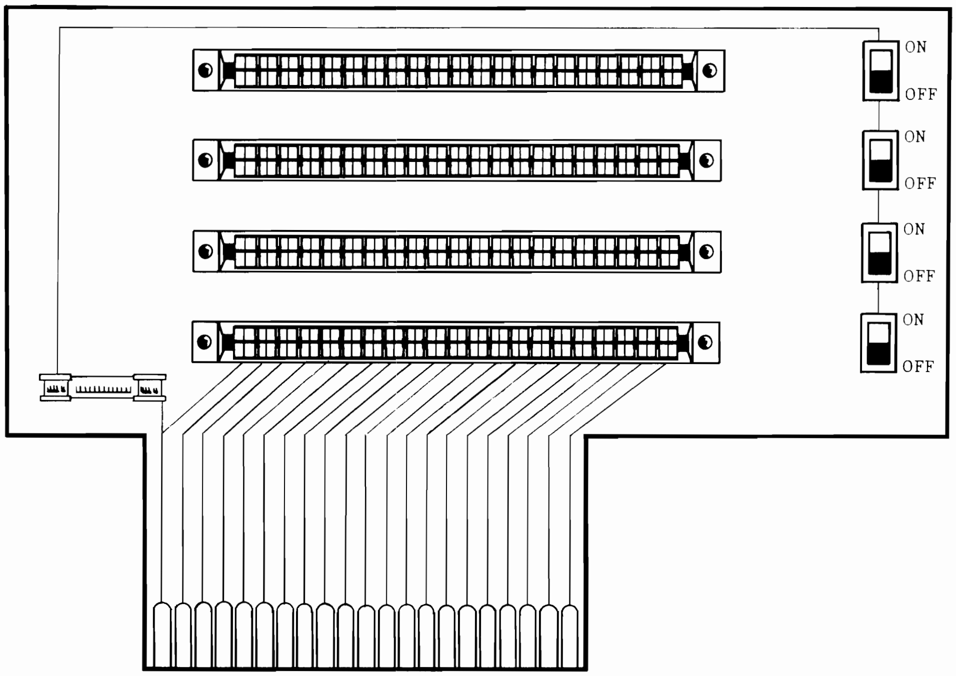
The edge connector system is ideal for the occasional replacement of faulty computer cards during maintenance, but it is not really designed for repeated insertion and removal of boards. That can damage both the plating on the contacts and sometimes the spring connectors in the edge connector sockets.

VIC's original design assumed that only one ROM or RAM cartridge would be put directly in the port, and some of Commodore's cartridges reflect this. For example, *VICMON* assumes the screen starts at $1E00, which is true for the unexpanded VIC.

Expansion boards (a typical board is shown in Figure 5-5) have been produced almost as long as the VIC has. A typical expansion board has three or four slots into which RAM or ROM devices can be inserted, with their labels facing the user. Several cartridges can be used at once provided their memory requirements don't overlap; for instance, *Super Expander, Programmer's Aid,* and *VICMON* can all be plugged in and used at one time. So can several RAM expanders, *VICMON* and ROM cartridges, and RAM with ROM. If each slot has a switch to turn its power on or off, several cartridges designed for the same area of memory can be present simultaneously, although only one of them can be in operation at any one time.

An expansion board is a virtual necessity for serious programming on VIC-20. However, if more than four additional devices are used (particularly with the cassette recorder, which is a heavy power user) the VIC's built-in power supply may not be adequate. Thus, some expansion boards include their own power supply and must be plugged into a wall outlet. That increases the quantity of cable spaghetti, as well as the cost of the expanders. Other boards may rely on battery power, and rechargeable batteries can make for a tidy board.

Figure 5-5. Typical Four-Slot Expansion Board



Commodore's own expansion board, the 1010, seems to have been withdrawn, possibly because cheaper alternatives were available (or perhaps because it was realized that such boards make it easier to copy software). The 1010 had six slots and its own power supply, so concern for the VIC's power supply could not have been a factor.

Expansion boards are relatively simple pieces of equipment, which hardware people with printed circuit board equipment can easily make. The simplest type just extends the 44 connections from the expansion port and puts several edge connectors across them. The edge connectors exactly duplicate the configuration of the expansion port. Ribbon connectors, rather than rigid boards, can also be used.

Hardware Aspects of the Expansion Port, Chips, and Addressing

For a more complete understanding of the expansion port, it is helpful to take a look at the connections provided by the port, and also at the pinout of a chip and the internal layout of typical cartridges. You'll see how to figure out the purpose of most of the expansion port's lines.

Figure 5-6. The VIC-20 Expansion Port

		Bottom	Top		
GND	Ground	Z	22	Ground	GND
	[Not used]	Y	21	+ 5 Volts	+ 5V
$\overline{\text{RESET}}$	6502 $\overline{\text{RESET}}$	X	20	[Not used]	
$\overline{\text{NMI}}$	6502 $\overline{\text{NMI}}$	W	19	$\overline{\text{IRQ}}$ to 6502	$\overline{\text{IRQ}}$
S/02	Phase 2 Clock	V	18	READ/$\overline{\text{WRITE}}$ (6502)	CPU/R/$\overline{\text{W}}$
$\overline{\text{I/03}}$	I/O Block 3 $9C00	U	17	READ/$\overline{\text{WRITE}}$ (VIC Chip)	VIC/R/$\overline{\text{W}}$
$\overline{\text{I/02}}$	I/O Block 2 $9800	T	16	RAM3 $0C00	$\overline{\text{RAM3}}$
CA 13	Address Bit 13	S	15	RAM2 $0800	$\overline{\text{RAM2}}$
CA 12	Address Bit 12	R	14	RAM1 $0400	$\overline{\text{RAM1}}$
CA 11	Address Bit 11	P	13	Block 5 $A000	$\overline{\text{BLK 5}}$
CA 10	Address Bit 10	N	12	Block 3 $6000	$\overline{\text{BLK 3}}$
CA 9	Address Bit 9	M	11	Block 2 $4000	$\overline{\text{BLK 2}}$
CA 8	Address Bit 8	L	10	Block 1 $2000	$\overline{\text{BLK 1}}$
CA 7	Address Bit 7	K	9	Data Bit 7	CD 7
CA 6	Address Bit 6	J	8	Data Bit 6	CD 6
CA 5	Address Bit 5	H	7	Data Bit 5	CD 5
CA 4	Address Bit 4	F	6	Data Bit 4	CD 4
CA 3	Address Bit 3	E	5	Data Bit 3	CD 3
CA 2	Address Bit 2	D	4	Data Bit 2	CD 2
CA 1	Address Bit 1	C	3	Data Bit 1	CD 1
CA 0	Address Bit 0	B	2	Data Bit 0	CD 0
GND	Ground	A	1	System Ground	GND

Note: Diagram shows pins in the sequence they appear on a *board*. Thus, a RAM pack face up with pins toward you has the 5-volt power line second from left on top.

Figure 5-6 shows how the port is configured. It is possible to trace the connections with the aid of the schematic layout printed in the *VIC-20 Programmer's Reference Guide*. The diagram shows the layout as it appears on an expansion board or a plug-in cartridge, which is the way it's usually seen. The numbering and lettering of the pins are standard; however, some VICs are numbered back-to-front. Note the position of the 5-volt power line; this is always connected and often has a wider track than the other lines.

Most of the other connections select addresses. There are 13 bits common to all addresses, 8K block selection lines (for $2000, $4000, $6000, or $A000), 1K selection lines (for the 3K RAM expander—$0400, $0800, or $0C00), and 2K selection lines for $9800 or $9C00. The other lines can be used in hardware control; we'll see how RESET can provide a useful reset switch.

The horizontal line above some of these abbreviations is a notation derived from logic, meaning "not." Thus, IRQ can be read "not IRQ," or, more accurately, as "IRQ active low." That means that the line is normally high (5 volts) and will generate an interrupt when brought low (grounded).

Expansion cartridges use the following lines:

Line 17. VIC chip READ/WRITE
Lines 1–9. Data bits 0–7

Lines B–P. Address bits 0–11
Lines 10–13. Some are used to select 2K blocks
+5 volt and ground lines

To see how these lines function, consider, for example, POKE 8192,12. This puts 12 (%00001100) on the data bus, so the eight bits of the data line take corresponding values of 0 or +5 volts. The address 8192 is $2000 in hex; all address bits 0–11 are low, and block $2000–$3FFF is selected. When the READ/WRITE line is brought low, the data is written to RAM.

All this, of course, happens in a fraction of a second. The actual order of events is shown by timing diagrams, and the precise detail is complex.

Commodore's RAM and ROM Cartridges

Each of Commodore's RAM and ROM cartridges has interesting features, so we'll look at each one in turn. The boards don't contain many components. A typical board may include capacitors from the 5-volt line to ground, RAM or ROM chip, and perhaps a multiplexer which picks the correct chip to read from or write to on the basis of address information from the VIC-20. The circuit boards usually have optional connectors which allow the addressing to be changed, so the basic board is potentially usable in the manufacture of other products. Incidentally, most components can be identified, and their functions checked, by reading the numbers stamped on them and referring to a parts catalog.

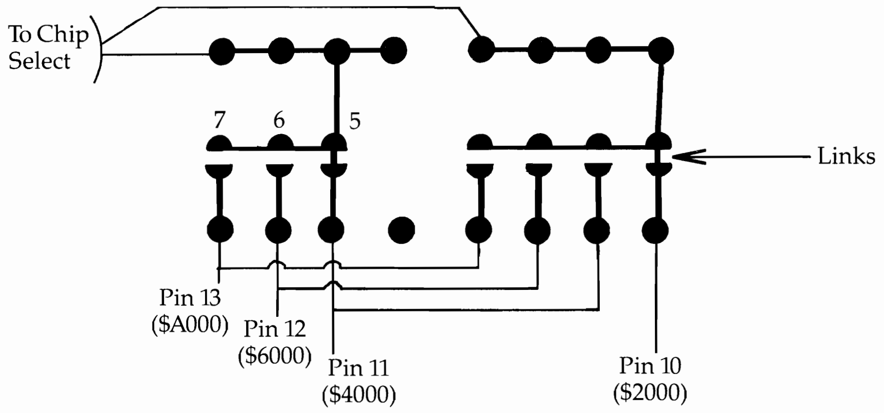
The 16K RAM expander (the VIC 1111) has eight chips of 2K each, mounted on a board identical to the 8K expander. Capacitors are included to provide protection against power supply irregularities. An LS139 dual two- to four-line multiplexer selects one of the eight RAM chips. Input from pin 10 or 11 selects block $2000 or $4000; input from pins P and R, corresponding to address bits 11 and 12, selects one of the four relevant chips.

Selection of blocks $2000 or $4000 is done by the circuitry to the right of the multiplexer chip, as shown in Figure 5-7. It is a simple matter to reconfigure the addressing. For example, cutting the connection marked 5, and making a new connection with a small blob of solder on 7, gives you one 8K block at $2000–$3FFF and another 8K RAM block at $A000–$BFFF. (Note that this will probably invalidate any warranty.) A switch can be fitted to have the same effect.

The 8K RAM expander (VIC 1110) is half a 16K expander. It contains four 2K RAM chips. It has an additional interesting feature, a small four-section switch, which can be accessed only by opening the case of the cartridge. When one of these sections is on, contact is made to one of lines 10–13 on the expansion port, thus selecting an 8K block starting at $A000, $6000, $4000, or $2000 (in order, left to right). That makes the 8K expander perhaps the most valuable of the Commodore expanders. Up to four can be used at once, but this is a costly way to provide memory. The expander can be upgraded to 16K by adding four more RAM chips and four capacitors.

The 3K RAM expander (VIC 1210) contains a smaller circuit board, which is identical to that of the *Super Expander*. Unlike the other expanders, pins 14, 15, and 16 are connected, because the 1210's memory does not start at a block boundary but at $0400. The board has six chips, each with 512 bytes of RAM. They are low power

Figure 5-7. Address Selection of 8K and 16K RAM Expanders

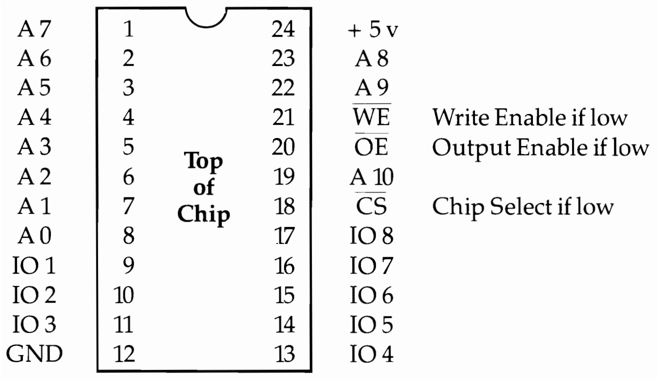


Note: With 16K RAM, the left circuit selects \$4000 – \$5FFF; the right selects \$2000 – \$3FFF. Resoldering the links enables other combinations to be selected.
With 8K RAM, only the right circuit operates; it is switchable to start at one of \$A000, \$6000, \$4000, or \$2000.

chips and can operate by drawing power down address and data lines, so that expansion boards with switches may not be able to turn these expanders off. The board is wired to accommodate ROM (or EPROM) at \$A000, such as the *Super Expander.* But the circuitry permits any other block to be used with a little modification.

As an example of a memory chip's pinout, consider the 2K RAM chips in the 8K and 16K RAM expanders. The pinout and numbering convention is illustrated in Figure 5-8. This particular chip is CMOS static RAM, but all RAM and ROM chips are basically similar.

Figure 5-8. RAM Chip Pinout


A 7 1 24 + 5 v
A 6 2 23 A 8
A 5 3 22 A 9
A 4 4 21 $\overline{\text{WE}}$ Write Enable if low
A 3 5 20 $\overline{\text{OE}}$ Output Enable if low
A 2 6 19 A 10
A 1 7 18 $\overline{\text{CS}}$ Chip Select if low
A 0 8 17 IO 8
IO 1 9 16 IO 7
IO 2 10 15 IO 6
IO 3 11 14 IO 5
GND 12 13 IO 4
Top of Chip

It's sometimes helpful to trace the circuit paths within the VIC or within a cartridge. Note that there are 11 address lines, labeled A0 to A10, permitting the chip to distinguish 2↑11 = 2048 bytes, which of course is 2K. There are eight data lines, labeled IO1–IO8, allowing eight-bit bytes, each with 256 possible values. IO means Input/Output, and this usage distinguishes RAM from ROM, which is output only.

The 5-volt power supply and ground pins are at opposite corners of the chip. The three remaining lines are WE (Write Enable if low), OE (Output Enable if low), and CS (Chip Select if low). When CS is brought low, that particular chip, wired to its own 2K of memory area, can be written to or read from, depending on whether write enable or output enable is low or high. If write enable is held high, the chip effectively becomes ROM, since it cannot be written to.

Modifying Expansion RAM Expanders and Expansion Boards

It is possible to make some useful modifications based on what has been covered so far. None of them require much hardware expertise, and I have personally found all to work perfectly. However, I can accept no responsibility for failure or error.

Reset switch. Pin X on the expansion port resets the 6502 chip when grounded. The effect is similar to switching off and then on again, except that most RAM is retained. BASIC can be recovered intact, including variables, using OLD, a short ML routine in Chapter 6. The reset switch is identical in effect to SYS 64802, but its reset action can be carried out even with protected programs, where it is impossible to exit with the STOP key.

The simplest approach is to attach wires to pin Z (ground) and pin X (6502 RESET). On a plug-in board, these pins are at the bottom left, if the board's contacts are toward you. Momentarily touching the two wires together causes reset. A more elegant approach is to install a momentary-contact switch. A .01μF capacitor, placed across the switch contacts, is desirable.

Note that pin X is not connected in RAM or ROM cartridges, as it is in expansion boards, so if you don't have an expansion board you'll have to connect the RESET pins at the port. The proper pins are the two bottom right ones inside the slot.

Is it possible to use the NMI line for a reset switch which leaves memory in the lowest parts of RAM untouched? This is potentially useful in investigating programs, but the answer generally is no. NMI processing is more conditional than RESET, as you saw when using the RESTORE key, so software which disables the NMI or changes its vector will be unaffected by NMI resetting. In any case, the X2 crash can't be interrupted by NMI. For these reasons, reset switches usually rely on pin X.

Battery backup of RAM. This modification powers RAM (or ROM, for that matter), using a supply external to the VIC. This has two advantages. First, it removes all external drain from the VIC's own power supply; second, it maintains data in RAM after the VIC has been switched off, so you don't have to reload it later. Note, however, that if the battery runs down or is disconnected, the program is lost.

As an example, consider an 8K RAM expander. If it is opened, the top left track is ground, and the one next to it carries power to the board. If the power is off, the RAM is inactive. A piece of masking tape over the power track connection will isolate it from the connector in the VIC; to make the change permanent, use a sharp tool to score through and break the circuit at the base of the power track connection, where it passes onto the main portion of the circuit board. (Cover or break the +5 volt line only; do not disconnect the ground connection or the expander will not work.)

At that point, you're ready to connect the battery backup. Connect the positive lead to the power line and the negative lead to ground. Both connections are easily made at the base of the electrolytic capacitor in the lower left corner of the top of the board. Ideally, you should use a 5- or 6-volt battery pack, although many experimenters use small 9-volt transistor radio batteries.

When plugged into the VIC, the expander draws all its power from the battery. It retains its addressing and data line connections with the VIC, and any data stored in it will remain after VIC itself is off. In effect, the expander acts as a sort of programmable ROM.

Making RAM appear like ROM. Pin 17 (READ/WRITE) signals readiness to write to RAM. If the line is not connected, RAM cannot be written to and behaves like ROM. You can verify this by covering this track on a RAM expander with masking tape; the contents of RAM remain what they were when the computer was turned on but cannot be changed. Obviously, this is not of much value as it stands; a switch is needed to select RAM (when a program is being entered or loaded) or ROM (to preserve the program, which will be immune from overwriting and corruption). Expansion boards are easily modified in this way.

2K RAM for $9800–$9FFF. This is a relatively simple conversion operation. 2K of the 3K expander is used, by readdressing $0800–$0FFF so that it appears at $9800–$9FFF. 1K of the expander is left disconnected. (Alternatively, if $0400–$0BFF is readdressed as $9800–$9FFF, 1K expansion is left with BASIC, giving 4607 bytes free.) Note that 8K and 16K RAM expanders aren't quite so easily modified.

Three tracks (14, 15, and 16) control the 3K expander's addressing; these are left of center on the top of the 3K board. If these are masked off, $0400–$0FFF is no longer accessible as RAM. Pins T and U on the bottom of the board select $9800 and $9C00, respectively; if they are connected to the tracks where $0800 and $0C00 were selected, all the internal wiring of the expander applies, but the actual address is shifted to $9800–$9FFF. As it happens, those tracks are close together on the bottom of the board, and a small modification (two jumpers) is all that's needed.

Changing block addresses. Blocks $A000 and $6000 are selected by pins 12 and 13 of the expansion port. A simple interchange of these pins, or substitution of pin 12 for pin 13, allows RAM or ROM to appear at either address at will. This is significant because VIC's tape system won't allow a program to be saved beyond address $8000. However, with perhaps a switch on an expansion board, saving of ROM software to tape is possible. The following subsection deals with this topic, which is fairly tricky, in detail.

Address changing is easy to demonstrate on an 8K expander, provided it's mounted away from VIC so its miniature switch is accessible. Suppose VICMON is also connected. If the 8K expander's switch is set to $A000, for instance, fill RAM up to $C000 with $FF. Then, changing the switch to select block $2000 causes the pattern of 8192 $FFs to appear at $2000–$3FFF. They could also be moved to start at $4000 or (if VICMON is off) at $6000.

Saving Programs at $8000 and Above to Tape

There are only three areas above $8000 that you are likely to want to save to tape: the color RAM, the VIC registers, and the ROM expansion area from $A000 to $AFFF or to $BFFF. The first two are necessary if VIC's TV picture is to be saved; the third allows the ROM area to be loaded back into an 8K RAM expander adjusted to start at block $A000. The programs are detailed in Chapter 14; here we'll explain three approaches:

Saving the area as a file. This is straightforward but fairly slow (more than five mintues for 8K). Each address is PEEKed and the resulting value written to tape. No special hardware is needed.

Moving ROM down. When using an expansion board, an 8K RAM expander and a ROM cartridge can be present at the same time. Chapter 14 includes a program which moves ROM down, then saves it to tape with a forced load header so that it will automatically load into the correct memory area.

Using a block switch. With an expansion board and a switch from block $6000 to block $A000, SAVEing is faster than with the first approach. LOADing is easier, too, but still may not justify the extra work. However, the technique does illustrate address line changes. Typically, this sequence might be used:

1. Set switch to connect $A000 to $A000.
2. Load or plug program into $A000.
3. Switch so program appears at $6000.
4. SYS 64802 or NEW to reset pointers.
5. POKE 43,0: POKE 44,96: POKE 45,0: POKE 46,128 to alter BASIC pointers to cover $6000 to $8000.
6. SAVE "PROGRAM NAME",1,1 which stores memory from $6000 to $7FFF on tape, with a forced load header to insure reload into the identical area.

To reverse the process, use this sequence:

1. Set switch to connect $6000 to $A000.
2. POKE 183,0: SYS 62937 to load from tape without resetting as though the program were BASIC.
3. Switch $A000 to $A000.
4. RUN the program. Typically, SYS 64802 simulates switch-on.

The VIC (Video Interface) Chip

TVs

Commodore's designers had the problem of interfacing the VIC with TVs, which aren't directly controlled by the computer. Their solution was effective and relied on isolating the TV-specific parts of the computer's operations as much as possible.

TV sets in the United States, Canada, Japan, and much of South America use the NTSC (National Television Standards Committee) standard of 525 lines per screen. The screen is refreshed 60 times per second. Most of Europe (excluding France), Australia, and some other countries use PAL (Phase Alternation by Line), which uses a 625-line picture and refreshes it 50 times per second. France and the USSR use SECAM, a system resembling PAL but with certain parameters changed.

All sets use interlace, which means that only half the picture is traced in each top-to-bottom scan. On American sets, 262 lines (every other one) are drawn each 1/60 second; thus, the full picture needs 1/30 second. This allows for the appearance of motion without flicker.

Not all TVs produce an identical picture. Some, particularly those with a squarish screen, lose some at the sides and top. In any case, there is a guard band of lines which are not intended to be displayed, and many sets are adjusted with overscan, meaning that the edges are removed. Since VIC-20's display size can be varied from 22 × 23 characters, this can be significant.

Some TVs do not work well with personal computers. In general, newer TVs are better than old, because the manufacturers now consider computers when designing their sets. Automatic tuning circuitry and field synchronization (without which the picture flutters up) have been two difficult areas, but neither square-wave sound nor static charge noise (when a screen suddenly blanks) has posed major problems. However, it is still desirable to keep the brightness level below its maximum.

Interfacing VIC-20 to a TV

TV output is generated by the video interface chip, or VIC, which gives the VIC-20 its name. These chips are designed and manufactured by MOS Technology, a subsidiary of Commodore. The VIC chip is a 40-pin integrated circuit positioned near the middle of the printed circuit board; some models have shielding around this area, so the chip may not be very easily accessible. The American version (for NTSC TVs) is numbered 6560; the PAL equivalent is the 6561. Improvements are continually being made in the VIC chip's design, so it may be worthwhile exchanging an older VIC chip for its most recent revision.

VIC produces a composite output (not a separate signal for red, green, and blue) which makes it difficult to produce a high-quality picture. Channels 3 or 4 are normally used in the U.S.; in the U.K., a videotape channel (36) is generally chosen.

In both cases an external modulator is used between the VIC and the TV. If it is handled carelessly, so that its connecting leads are weakened or broken, the picture may deteriorate or disappear. This fault is relatively easy to correct, by resoldering. Poor color can be improved either by tweaking the potentiometers in the output circuits (R7 or R32 on the VIC schematic diagram) or by adjusting the video or audio coils in the modulator, if they exist. The cores are easily damaged, however, and a nonmagnetic alignment tool should be used. Ordinarily, such adjustments shouldn't be necessary, and in any case they are best performed by experienced dealers.

Note that VIC-20s and their modulators are matched, to some extent, so one VIC-20 may not work well with another VIC-20's modulator. However, Commodore's color monitor, which needs no modulator, works with any VIC and produces more stable pictures than an ordinary TV.

What restrictions does the need to conform to TV standards impose on the computer? The output to the TV must be correctly synchronized, so the electronic clock which times all the VIC-20's operations must be properly set. In practice, a high-frequency clock is subdivided to drive both the VIC chip and the 6502 processor; in the U.S., the 6502 runs at 1.02 megahertz (MHz or million cycles per second), while in the U.K. its frequency is set to 1.10 MHz. As a result, programs run slightly faster on PAL sets than on NTSC sets. Other differences concern the position of the screen generated by the VIC-20 and the maximum size of the screen—24 rows by 28 columns in the U.S. but 26 rows by 32 columns (which still leaves space) on PAL sets.

Fortunately, international variations in the VIC-20's design are usually irrelevant to the individual programmer. However, there is sufficient traffic in programs to justify some discussion of the topic. Most differences are confined to the Kernal and do not affect BASIC, so exchanging the Kernal ROM (one of the two ROMs on the circuit board) and using a different VIC chip are the major changes needed to internationalize the VIC-20. In addition, the timing for the interrupt which controls the BASIC TI$ clock and the timing for tape and RS-232 operations require different Kernal constants. Other necessary modifications may involve characters, keyboard, and power supply.

Using the VIC Chip

The following is a detailed discussion of what the VIC chip can do. Sample programs are given to illustrate the function(s) of each register. For reference purposes, the appendices contain two diagrams which summarize the main features of the VIC. Chapter 12 (Graphics) has many programs using the VIC chip.

$9000 (36864)

Bit 7 selects the interlace mode. Normally, half the lines that make up a TV picture are scanned every sixtieth of a second, scanning every other line. The next sixtieth scans the remaining lines, with the idea being to avoid flicker.

Interlace mode turns off one of these scans. Another TV picture can then be superimposed onto the VIC-20's output. This may improve some TV pictures, so it is provided as an option on some software. Usually this bit is off.

To see the effect, if any, just add 128 to the POKE value to set the bit if it's off, or subtract 128 to turn it off if it's on. If you're not sure of its status, POKE 36864, PEEK(36864) OR 128 turns interlace on, while POKE 36864, PEEK(36864) AND 127 turns interlace off.

Bits 0–6 determine the distance of the picture from the left side of the screen. This value is normally 5 (U.S.) or 12 (U.K.). Some commercial programs give the option of controlling this feature. Each change in value of 1 moves the screen by four dots, half a character's width.

This is not very high resolution. The range 0–127 corresponds to several screen widths, so the screen can be moved right off the TV. Chapter 12 describes how horizontal positioning can help you scroll horizontally.

To watch the effect, if interlace is off, POKE 36864 with your chosen value. If you want to be sure that interlace is unchanged, use POKE 36864, PEEK(36864) AND 128 OR X, where X is 0 to 127.

$9001 (36865)

This register selects the distance of the picture from the top of the screen. Normally that is 25 (U.S.) or 38 (U.K.). A few commercial programs let you change it. Each change in value of 1 moves the screen by two dots, one-quarter of a character's height. This is good resolution; Chapter 12 shows you how to use it to achieve smooth upward scrolling.

To watch the effect, POKE 36865,X where X is 0 to 255.

$9002 (36866)

Bit 7 controls the start of screen position jointly with $9005. This bit is treated as bit 9 of the screen start address and is the lowest bit of the screen under software control. A screen address is usually of the form %0001 XXY0 0000 0000, where bits 1XX are controlled by $9005, and bit Y is controlled by $9002. For example, a screen start of $1000 means Y is off, while a screen start of $1E00 means Y is on.

Bit 7 also controls the color RAM address. When it is set to 0, color RAM starts at $9400 (37888); when it is 1, color RAM starts at $9600 (38400).

Because of this dual function, this bit is sometimes said to control the alternate screen mode. As an example, when the screen is at $1E00, its colors are controlled by RAM starting at $9600, and this bit is set. POKE 36866, PEEK(36866) AND 127 turns off this bit, making the screen start at $1C00 and color RAM start at $9400, so an entirely new complete screen is available.

Why should color RAM move like this? If the VIC's screen size were fixed at 22 × 23, there would be no need for it. However, the screen dimensions can be enlarged (for example, to 24 × 26). In that case, 512 bytes from $9600 to $97FF would be insufficient. Thus, it's necessary to have an option of selecting color RAM from $9400, which covers all possible screen row and column combinations.

Bits 0–6 determine the number of columns on the screen. These bits are normally set to 22; PEEK(36866) is therefore either 22 or 150, depending on screen position. To change the number of columns, leaving the screen position bit unchanged, use POKE 36866, PEEK(36866) AND 128 OR X where X is 0 to 127. X has an effective maximum of 238, beyond which it has no further effect, but in practice no more is likely to be required since that more than fills the screen horizontally. VIC-20 screen editing is not "soft," so screens of widths other than 22 columns (with the exception of 11) are relatively difficult to edit.

If screen rows multiplied by screen columns exceed 512, the usual 512 bytes of screen RAM are insufficient and more must be allocated. Chapter 12 explains how.

$9003 (36867)

Bit 7 belongs with register $9004; it is the smallest bit of the current line being scanned. Because the raster scan line changes so fast, PRINT PEEK(36867) normally alternates between 46 and 174 (128=46).

Bits 1–6 determine the number of rows on the screen. To change rows while in the normal 8 × 8 character mode, POKE 36867,2*R, where R is the required number of rows. For example, POKE 36867,2 gives a single-line display. If you are unsure of the mode, use POKE 36867, PEEK(36867) AND 1 OR 2*R, which retains the value of the smallest bit. 2*R (twice the number of rows) is POKEd in because doubling shifts the bit pattern left to coincide with the VIC's register.

Besides changing the number of rows with ordinary graphics, this register enables double-sized character graphics to neatly fill the screen. Typically, 20 columns by 10 rows, in double-height mode, will give the same screen shape as a 20 × 20 screen made up of ordinary characters.

Bit 0 selects normal (8 × 8) characters or double-sized (8 × 16) characters. Any VIC-20 screen is regarded by VIC as having a fixed number of locations; normally, with a 22 × 23 screen, that number is 506. Each of these locations contains a byte from 0 to 255, corresponding to a pattern stored in the character-generating memory.

In normal mode (8 dots × 8 dots), 64 bits are needed to define the whole pattern; if the bit is on, the dot is on, and vice versa. That means that eight bytes are required for each character, so (for example) the tenth character's pattern begins 72 bytes after the start of the character generator.

In double-size mode, twice as much memory (16 bytes) is required to define each character. Try POKE 36867,47 to set this mode; note how the screen extends down. (POKE 36867,23 corrects this by reducing the number of rows to 11.) Clear the screen; type @,a,b,c,d, and so on. Each character is twice the normal size and uses twice as much information from the character generator ROM. This mode is useful because it lets 22 columns by 11 rows fill the screen. That is less than 256 bytes, which means that every dot on the screen can be separately plotted to produce truly high-resolution graphics (except for colors, which are variable only inside their 8 × 16 area).

$9004 (36868)

This register identifies the TV screen line being scanned. It changes rapidly and is only usable with machine language. Since this is a nine-bit register (the smallest bit is stored in $9003), its maximum range is 512. The actual range is roughly 260 (310 in U.K.) which suggests that the VIC doesn't distinguish between alternate screen scans. Chapter 12 illustrates a split screen program which uses this register. Light pens generate coordinates which have the same vertical range of values.

$9005 (36869)

Since this register is the most difficult VIC register to understand, a table in the appendices lists all the usable values along with their meanings. However, the following will give you a quick overview.

Bits 4–7 control the starting address of the screen, along with one bit of $9002. The start of screen is under software control by five bits, suggesting that there are 2↑5 (32) possible screen positions. If left to itself, the VIC-20 will only select two of them; however, any of the others can be selected by software.

The most useful are the following: $1000, $1200, $1400, $1600, $1800, $1A00, $1C00, and $1E00. All can exist on any VIC-20 without extra memory. If you represent the five relevant bits as PWXYxxxx in $9005, and Zxxxxxx in $9002, the derived screen address will be %Q00W XYZ0 0000 0000, where Q is the reverse of P. All the usual addresses have bit P=1 and bit W=1 and can all be written as %0001 XYZ0 0000 0000 in binary; in this representation 1, X, and Y (bits 12, 11, and 10 of the address) come from $9005, while Z (bit 9) is from $9002.

Another, less useful set of screen addresses is selected when bit 6 is 0. They are all of form %0000 XYZ0 0000 0000 in binary, and therefore include, in principle, $0, $0200, $0400, $0600, and so on. The first cannot be used with BASIC and can only be partially used, with difficulty, in machine language. Address $0200 cannot be used with BASIC but can be used with machine language. The others, from $0400 to $0E00, cannot be used even with 3K RAM expansion, because of the way VIC-20 is designed. A snowy effect results, with the actual appearance controlled by the character set; try POKE 36869,64 with the normal character set. POKE 36869,119 is a typical variation.

Bit 7 of $9005 is almost always set to 1, since setting the screen to look at ROM or the I/O chips is pretty useless. (This addressing is actually used by the character generator, bits 0–3 in this register.) Given that bits 7 and 6 are usually 1, there are in effect only three other bits, giving you the eight possible normal screen positions.

The starting position of the screen can be calculated from this formula:

4*(PEEK(36866) AND 128) + 64*(PEEK(36869) AND 112)

which allows for locations $9002 and $9005. This will normally be identical to 256*PEEK(648), because the system stores the high byte of the screen position in 648. Thus, PRINT PEEK(648) is usually 30 (with the screen at $1E00, on an unexpanded or 3K expanded VIC) or 16 (with the screen at $1000, on a VIC with 8K or more expansion).

If those values are not equal, screen editing may not work. SYS 58648 will make them match; so will SYS 64818, which also cold starts BASIC. POKEing an even value from 16 to 30 into 648, then entering SYS 64828, lets you set the screen to any of the eight positions at will.

Each screen needs a color RAM area. You saw how bit 7 of register $9002 selects such an area and how it also controls bit Z of the screen address. It follows that adjacent screens use different color RAM. To better grasp the ways that the screen can move, enter POKE 648,18: SYS 64818. Now the screen starts at $1200 (or, in decimal, 18*256=4608). POKE 4608,1 will confirm this by putting the letter A in the top left of the screen, provided a character is already there so the color is set.

This is a nonstandard position and could not be selected by VIC-20 on its own. Some knowledge of BASIC is needed when using nonstandard screen positions to insure that the program won't corrupt the screen. Shifting the screen like this can be useful as a security device; a program can be developed in which both normal screen areas contain vital setting-up routines, so if the program is interrupted, parts of it will be corrupted as the screen overwrites 512 bytes of program.

Program 5-3. Shifting the Screen

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 POKE 36879,8:REM CHANGE BACKGROUND COLOR:rem 40
20 FOR J=12 TO 15:REM BITS P AND W BOTH ARE 1 IN T
   HIS RANGE                                :rem 120
30 POKE 36869,16*J:REM ONLY HIGH NYBBLE NEEDS TO B
   E CHANGED                                :rem 104
40 FOR K=0 TO 1:REM ALLOWS FOR BIT Z        :rem 231
50 POKE 36866,128*K+22:REM SET BIT 7 AND RETAIN 22
    COLUMNS                                 :rem 167
60 P=(PEEK(36869) AND 112)/4:REM BITS W,X,Y:rem 68
70 P=P+(PEEK(36866) AND 128)/64:REM BIT Z :rem 161
80 POKE 648,P:REM SCREEN POINTER NOW CORRECT
                                            :rem 186
90 Q=Q+1.7:GET X$:IF X$="" GOTO 90:REM AWAIT KEYPR
   ESS                                       :rem 17
100 NEXT K,J:REM CYCLE THROUGH EIGHT VALUES
                                            :rem 129
110 POKE36879,27: REM SCREEN BACK TO NORMAL
                                            :rem 194
```

Program 5-3 demonstrates this, using POKEs into the relevant registers to move the screen. Each of the eight normal screens is displayed when a key is pressed. Try this program without 3K expansion, so BASIC itself is stored in one of the screens. Line 90 includes a calculation involving a BASIC variable Q, so the portion of BASIC which stores variables will be easy to find.

Note the alternation in color patterns caused by alternate selection of color RAM. You can STOP the program, write to one of the screens or clear it, then RUN the program again provided you haven't overwritten BASIC and watch the effect. If you print to one of the screens in color, the color RAM for that set of four screens will change. Note also that the screen editing won't be retained between screens; there's only one link table, so you may no longer be able to modify an existing line of BASIC.

Bits 0–3 control the starting address of the character generator bytes. As is the case with the screen position, there is no direct correspondence with the character generator. The highest bit is wired differently from the other three, so that if $9005 holds xxxxVWXY, the character generator table starts at %C00W XY00 0000 0000. Then, to confuse things a little more, when V is 1, C is 0, and vice versa. Basically, this allows ROM to store characters for normal BASIC, while retaining the option of user-defined characters which naturally must be placed in RAM.

For example, suppose $9005 holds xxxx0000. Then the character generator table starts at %1000 0000 0000 0000, or $8000. This is the uppercase and graphics mode which VIC goes into when first turned on. If $9005 is altered to xxxx0001 (by POKE 36869,PEEK(36869)+1, for instance), the character generator starts at %1000 0100 0000 0000, or $8400. These are reversed uppercase and graphics characters. $8800 and $8C00 hold the lowercase equivalents, which you can see by adding an additional 1 to 36869. There are four of these tables.

Note that PEEK(36869) AND 2 shows which mode the keyboard is in—0 for uppercase and 2 for lowercase. The unexpanded VIC-20 generally has 240 or 242 in this location, setting upper- or lowercase respectively. POKEing 241 or 243, which the VIC-20 would never set on its own, gives you unshifted characters that are printed in reverse. Setting $9005 to xxxx01xx tries to put the character generator tables into $9000, $9400, $9800, or $9C00; the first area is I/O chips, the second color RAM, and the final two are nonexistent. As a result, you will not get stable characters.

When $9005 holds xxxx1xxx, the range of values for the character generator is zero to %0001 1100 0000 0000 ($1C00), in steps of $0400. The usable range is $0000, $1000, $1400, $1800, and $1C00. One of those last four values is the normal choice for user-defined graphics work.

$9006 (36870)

This register and the three that follow are read-only registers that you cannot write to. This one yields the horizontal reading from a light pen. It is a latched value, since grounding a line to the VIC chip reads a value into both this and the following register. Both remain fixed until a new light pen reading is obtained. The resolution is not fantastic: Only two dots (a quarter of a standard character) can be resolved, and in practice resolution may be even less because of screen instability and inadequacies in the pen. The horizontal values in a 22-column screen therefore have a maximum range of about 88; typically this register holds a value ranging from 48 to 135. See Chapter 16 for full programming information.

$9007 (36871)
Contains the vertical position of the light pen and is similar to $9006. Its typical range is approximately 24 to 105 (38 to 129 in the U.K.), corresponding to quarter-character resolution with a 23-row screen.

$9008 (36872)
This and the following register are analog-to-digital converters and can be used with paddles, a mouse, graphics tablets, or any attachment which outputs two co-ordinates. This register holds the first analog reading, which may range from 0 to 255. Programming is straightforward (for example, PEEK(36872) in BASIC), although some elaborations are required, as explained in Chapter 16.

$9009 (36873)
This register holds the second analog reading. See notes on $9008.

$900A (36874)–$900D (36877)
These four sound-generating registers are virtually identical in operation. The first three are note generators; the fourth is a noise generator.

Bit 7 switches the register on or off. When bit 7 is set, the VIC chip generates a square wave. If the amplitude (volume) is also turned on in register $900E, a tone is generated and output with the TV signal. Thus, there are two independent switches for sound.

Bits 0–6 control the frequency of the sound; the higher the value in the register, the higher the pitch of the note. The value in the register serves as the start of a countdown; when it has been incremented to 255, the output toggles and a square wave is generated. $900A generates low tones, $900B generates medium tones, and $900C generates high tones.

Each of the tone registers produces notes one octave below the following register, so there is a lot of overlap. The actual frequencies for the high note generator are 15980/(255−X) (for the U.S.) or 17320/(255−X) (for the U.K. and Europe). Divide by 2 for the medium tone register and by 4 for the bass tone register. X represents the actual contents of the register, except that in the case of 255 the denominator becomes 128, producing the lowest note.

Program 5-4 shows how different values of X (given in the DATA statement) affect frequency. Type in and run the program; then press any key to sound the next pitch.

Program 5-4. Values vs. Pitch

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 POKE 36878,10:REM VOLUME ON             :rem 192
20 DATA 254,253,251,247,239,223,191,255   :rem 134
30 READ X: POKE 36875,X:REM MEDIUM TONE GENERATOR
                                          :rem 141
```

```
35 PRINT "{CLR}":PRINT "X ="X:REM DISPLAY X WHILE
   {SPACE}SOUNDING TONE                        :rem 102
40 IF X=255 THEN RESTORE                       :rem 119
50 GET X$:IF X$="" GOTO 50:REM AWAIT KEYPRESS
                                                :rem 47
60 GOTO 30                                       :rem 2
```

Register $900D generates noise by randomly varying the frequency of the square wave. *Noise* is used in the technical sense to describe a sound with no detectable note. The VIC chip reloads the frequency-determining timer with pseudorandom values that are not less than the value in this register. There is no way to get very low frequency noise with this register. However, very high sounds are easy to achieve, because the average pitch rises rapidly as the value approaches 255.

Chapter 13 explores sound in detail.

$900E (36878)

Bits 0–3 control the volume of the sound. When these bits are zero, no sound is sent to the TV even if one or more of the sound registers have bit 7 set. This gives a range of 15 different volumes for bit values of 1 to 15.

All four sound registers are controlled by this register; relative volumes can't be independently controlled. However, two or three registers may be set to generate the same note, which may have the effect of reinforcing the note. The precise effect of adding several sound registers depends on the exact time at which the registers were loaded with their values.

Avoid maximum volume (with bits 0–3 set to 15, or %xxxx1111 in $900E) when using VIC to make music, since the sound is often distorted on this setting. This is the fault of the chip, not the TV.

Program 5-5 varies the volume to produce a steam engine sound. It combines a low frequency component and a noise component, and continuously changes the volume of both in a loop. Volume builds relatively slowly, then drops rapidly back to zero.

Program 5-5. Steam Engine

```
10 POKE 36874,128:REM VERY LOW TONE
20 POKE 36877,129:REM SLIGHTLY DIFFERENT NOISE TON
   E
30 FOR J=0 TO 15:POKE 36878,J:NEXT:REM CHANGE VOLU
   ME
40 GOTO 30:REM REPEAT
```

If the auxiliary color is irrelevant (as it usually is when color graphics isn't in use), it is sufficient to POKE 36878 with a value from 0 (silent) to 15 (maximum volume). To change the volume without altering the auxiliary color, bits 4–7 have to be left alone. Use POKE 36878, PEEK(36878) AND 240 OR X where X is 0 to 15. If you want to use any value of X, PEEK(36878) AND 240 OR X AND 15 will throw away all bits in X adding 16 or more.

Bits 4–7 control the auxiliary color, another tricky concept. To illustrate it, switch on an unexpanded VIC. Color RAM starts at $9600 (38400). Now PRINT

PEEK(38400) AND 15. This returns the value 6, which is the color (blue) corresponding to the asterisk at the top left of the screen. Note that since color RAM uses only four bits, values 0–15 are the only significant values. AND 15 masks off (ignores) any meaningless higher bits.

Note too that values from 0 to 7 are used within the VIC to denote the primary and secondary colors black, white, red, cyan, purple, green, blue, and yellow. The order is identical to that on the keyboard, but the values for internal storage are 1 less than appears on the keyboard. Thus, blue is stored as 6. Try POKE 38400 with any value from 0 to 7; the asterisk, or any other character in the top left, will change color. For example, POKE 38400,0 changes its color to black.

Since four bits are allowed in color RAM, and since you have so far used only three to control the color, what does the fourth bit do? It sets auxiliary color mode, but only for the one character for which the bit is set. Thus, normal and auxiliary modes can coexist freely. Try POKE 38400 with values from 8 to 15. The top left character is displayed in a distorted form and with more colors than are usually possible. Usually a character has two color attributes: the background (one of 16 colors which is common to the whole screen except the border) and the character's own color (one of the 8 colors on the keyboard). However, when multicolor mode is set, a character can take on twice as many possible colors. These are the background color and character color, as well as the border color and a fourth color. That fourth color, the auxiliary color, is identified in this register.

The change in display that occurred when you POKEd 38400 with a value between 8 and 15 depends entirely on bit 4 of color RAM; when it is on, the VIC chip automatically redefines its way of outputting its data to the TV. With ordinary graphics, if a bit is on, it takes on the character color; if it is off, it assumes the general background color. But in auxiliary mode a *pair* of bits determines the color; 00 is background and 10 the character color, while 01 is the border color and 11 is the auxiliary color.

This adds to the color possibilities, but unfortunately (since two bits are required to identify each color) it lowers the horizontal resolution. In fact, it yields characters with widths defined by four characters instead of by eight characters. Thus, there is a noticeable tendency for horizontal character elements to be finer than vertical character elements.

Program 5-6 runs on the unexpanded VIC and shows the effect of using auxiliary color. Lines 20 and 60 use the positions of screen memory and color memory; they can be modified to work with any VIC configuration.

Program 5-6. Auxiliary Color

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
5 PRINT"{CLR}"                                  :rem 153
10 POKE 36879,24:REM BORDER BLACK, BACKGROUND WHIT
   E (0+8+16*1)                                 :rem 202
20 POKE 38650,13:REM GREEN CHARACTER PLUS AUX COLO
   R MODE                                        :rem 25
30 POKE 38654,0:REM BLACK ORDINARY CHARACTER NEXT
   {SPACE}TO AUX COLOR CHARACTER                 :rem 68
```

```
40 POKE 36878,32:REM AUX COLOR, WHERE IT EXISTS, I
   S RED (2*16)                               :rem 154
50 FOR J=0 TO 255:REM LOOP THROUGH EVERY CHARACTER
                                              :rem 211
60 POKE 7930,J:POKE 7934,J:REM POKE SAME CHARACTER
    TWICE                                      :rem 16
70 GET X$:IF X$=""GOTO 70:REM AWAIT KEYPRESS
                                               :rem 51
80 NEXT:GOTO50                                :rem 127
```

Type in and RUN the program. Then, press any key to cycle through the characters, watching the effect of the auxiliary color. Most characters resemble their normal selves, but some appear rather odd. Note that the horizontal resolution is much greater than the vertical resolution. Also, note that the colors sometimes appear weak, since they may be present in small amounts. For instance, a small area of the green character color may appear gray.

The program can be modified to show other color combinations. For example, the function keys could change one color each, or you could replace line 80 with NEXT: POKE 36879, 255 AND PEEK(36879) + 1 : GOTO 50 to cycle all border, background, and reversed character combinations. The colors need not be different.

Auxiliary color mode makes multicolor programming relatively easy, because the characters appear in a disguised form and effectively enlarge the VIC-20's built-in character set. When you need custom characters, for instance, it may be possible to save work by selecting suitable shapes from this extra character set without having to define a second character set of your own. A BASIC program can be developed to move ordinary characters around on the screen while setting colors by POKEs to color RAM. It is simple to convert these POKEs to multicolor mode, or to add POKEs to those parts to be displayed in multicolor, and if the characters have been well selected, the result should be effective.

Contrary to what you might think, multicolor mode operates in exactly the same way in double-size (8 × 16) character mode. This is easy to demonstrate. Just add another program line (for instance, add 0 POKE 36867,47) to set the double-size graphics flag in register $9003. Run the program, and again the whole double character has four colors and less resolution than normal.

If you are using this mode and not using sound, change the auxiliary color by POKE 36878, 16*X where X is 0 to 15, corresponding to the color required. If you wish to leave sound amplitude unchanged, mask off the lowest 4 bits with POKE 36878, PEEK(36878) AND 15 OR 16*X where X is 0 to 15.

$900F (36879)

Bits 4–7 control the background color. Sixteen different background colors (and auxiliary colors) are available. You can number these 0–15 and change the background in BASIC with POKE 36879, PEEK(36879) AND 15 OR 16*X, where X is 0 to 15. The actual colors are these:

0 BLACK	**4 MAGENTA**
1 WHITE	**5 GREEN**
2 RED	**6 BLUE**
3 CYAN	**7 YELLOW**

8 ORANGE	**12 LIGHT MAGENTA**
9 LIGHT ORANGE	**13 LIGHT GREEN**
10 PINK	**14 LIGHT BLUE**
11 LIGHT CYAN	**15 LIGHT YELLOW**

The colors from 0 to 7 are arranged in complementary pairs. In other words, if any complementary pair is turned on together (added together), the result will be white. Other colors are also achieved by mixing colors on the screen. Cyan is blue plus green, magenta is blue plus red, and yellow is red plus green. White is red plus green plus blue.

The monitor creates these colors by turning on combinations of the red, green, and blue color guns in the monitor's cathode ray tube. For example, magenta is created by simultaneously applying a voltage to the red and blue guns. White is made by applying equal voltages to the red, green, and blue guns, thus mixing equal proportions of red, green, blue, and so on.

The lighter colors, numbered 8 to 15, have similar patterns except for colors 8 (orange) and 9 (light orange). Colors 10 to 15 are lightened versions of colors 2 to 7 and can be considered identical except for added white (red, blue, and green). Some of them—for example, light orange, light cyan, and light yellow—can almost be treated as light gray.

Program 5-7 shows a use of the background color and also the reverse flag. Line 10 prints a message in white. If the program stops there, nothing (including the flashing cursor) will be visible if the screen is in its usual white and cyan mode. However, line 30 changes the background colors, omitting white and black, from orange through light yellow. Since the reverse flag is off, the lettering changes in a colorful way at a speed determined by line 40. Even if the whole screen is full of writing, the colors change at the same rate, far faster than they would be changed by POKEing a new color into each color RAM location. Change line 30 to 30 POKE 36879,J+8 to see what you get when the reverse flag is on.

Program 5-7. Changing Colors

```
1Ø PRINT"{CLR}{WHT}MESSAGE"
2Ø FOR J=48 TO 255 STEP 16
3Ø POKE 36879,J:REM REVERSE FLAG OFF, CHANGE BACKG
   ROUND COLOR
4Ø FOR K=1 TO 1ØØ:NEXT
5Ø NEXT:GOTO 2Ø
```

To get a similar effect on an all-black screen, the foreground must be set black. Add the following line: 15 FOR J=38400 TO 38911:POKE J,0:NEXT. Note that this line must be located after the screen-clear.

Bit 3 is the reverse bit which can reverse background and foreground. When this bit is on, characters are displayed normally (that is, 0's in the character generator appear in the background color, while 1's appear in the character color). When it's off, characters appear reversed.

The reverse bit is almost always on. Few programs use reverse mode, because it is conceptually tricky and because there are easier ways to achieve a similar effect, notably to PRINT the CTRL-RVS character. Watch the effect of Program 5-8, which toggles the reverse flag off and on.

Program 5-8. Reverse Flag Toggle

```
100 P=36879
110 POKE P,PEEK(P)+16*(((PEEK(P)AND8)=8)+.5)
200 GET X$:IF X$="" GOTO200
210 GOTO100
```

Line 110 tests for bit 3 and flips it. To show the full effect, print different-colored characters on the screen first. This bit has the same effect with double-sized characters, but note that it has no effect on characters in auxiliary mode. Chapter 12 discusses this bit in more detail.

Bits 0–2 control screen border color. The exterior border can take any of 8 colors, which are the same primaries and secondaries as the character colors (black, white, red, cyan, purple, green, blue, and yellow). In BASIC the border can be changed with POKE 36879, PEEK(36879) AND 248 OR X where X is assumed to take a value from 0 to 7. Sometimes it's easier to let X take consecutive values, for example when looping from 0 through 255, and in this case put (X AND 15) in place of X.

This register is initialized to $1B (27). With that value, the background color is 1 (white), the reverse flag is 1 (nonreversed characters), and the border color is 3 (cyan). In addition, the color RAM is set to 1 (white) on initialization or on clearing the screen, so a POKE to the screen won't be visible (it is white on white) unless the color is set too. Alternately, if you are POKEing to the screen, the POKEs can be made visible by changing the background (for instance, by POKE 36869,8 to produce white characters on black). This is easier and faster than doing something like FOR J = 38400 TO 38400 + 511 : POKE J,0 : NEXT.

Versatile Interface Adapters (VIAs)

The Versatile Interface Adapter, or VIA, is a 40-pin chip called the 6522 that is designed specifically to handle interfacing. Connecting the VIC to external devices is a complex and technically involved subject, and there isn't space to deal thoroughly with its hardware aspects here. However, this section discusses the important software aspects of the VIA. You will find this particularly helpful if you wish to write your own, or decipher other people's input/output routines, or handle such relatively easy matters as disabling RESTORE or turning the cassette motor on.

Your VIC-20 has two VIAs, located at the upper left corner of the circuit board. The best way to introduce yourself to these chips is to examine VIC's schematic, which is the fold-out wiring diagram in Commodore's *VIC-20 Programmer's Reference Guide*. The 6522 chips appear near the bottom of this diagram. They are connected to the keyboard, the RESTORE key, the cassette port, the serial port, the joystick and light pen port, pin 18 of the expansion port, and both interrupt pins (IRQ and NMI) on the 6502 processor. They're also connected to the 24-pin user port at the back of the machine. That port has its own power supply and an eight-bit parallel data port, plus two control lines.

All serial output (to VIC disk drives, printers, etc.) is handled by the VIAs through the serial port. All RS-232 input or output using device number 2 (usually a modem) has to use the user port. Non-VIC gadgets which aren't memory expanders can plug into any of the ports; for example, converters to the IEEE bus standard can use the memory expansion port and intercept the normal input and output from there, or if the converters have an external processor, they can use the serial port or the user port. In fact, given the right software, the cassette port could also send and receive messages. The point is that *all* interfacing goes through the VIAs.

The chip labeled UDE8 on the schematic occupies 16 bytes in VIC's memory at $9010–$901F. It is usually called VIA A or VIA 1. The other, UDE7, appears at $9020–$902F and is called VIA B or VIA 2. A map of these VIAs' functions can be built up by examining schematics, looking at the hardware, and by searching VIC's ROMs to see how the VIA is used. Both chips are quite active, as you can see by PEEKing with Program 5-9.

Program 5-9. Looking at VIA

```
10 PRINT"{CLR}"
20 PRINT"{HOME}"
30 FOR J=36880 TO 36895:REM VIA A--FOR VIA B LET J
   =36896-36911
40 PRINT J, PEEK(J)"{LEFT}":NEXT:GOTO 20
```

Alternatively, you might want to try the ML routine given in Program 5-10. It is faster and can print 16 addresses of either of the VIA chips (in decimal) along with their bit patterns.

Program 5-10. Looking at VIA with ML

```
0 DATA 169,147,44,169,19,32,210,255,162,16 :rem 24
1 DATA 134,253,169,145,133,254,32,205,221 :rem 218
2 DATA 32,63,203,160,0,177,253,133,255,169 :rem 14
3 DATA 48, 6, 255, 144, 2, 169, 49, 32, 210
                                          :rem 187
4 DATA 255,200,192,8,208,240,169,13,32,210  :rem 8
5 DATA 255,230,253,166,253,224,32,208,211 :rem 220
6 DATA 32,225,255,208,197,96              :rem 112
10 FOR J =828 TO 890: READ X: POKE J,X: NEXT: SYS
   {SPACE}828                             :rem 238
100 REM ****{4 SPACES}DISPLAY ADDRESSES AND CONTEN
    TS OF VIA#1{4 SPACES}****             :rem 131
110 REM ****{14 SPACES}FOR VIA#2, CHANGE
    {16 SPACES}****                       :rem 180
120 REM **** 16 TO 32 (LINE 0), 32 TO 48 (LINE 5).
    {9 SPACES}****                         :rem 91
```

At this point it is helpful to look at the VIA chips' pinout on the schematic. Both chips have an eight-bit data bus, as expected, labeled D0 to D7. There are four address bits, RS0 to RS3, allowing 16 addresses to be distinguished, plus a chip select line. There are also Reset, clock, IRQ, Read/Write, +5 Volt power, and ground lines, which have the normal functions of synchronizing timing, sending interrupt request signals, and so on.

The 20 lines actually used in interfacing are, for each VIA, eight lines making up Port A (PA0–PA7), eight lines making Port B (PB0–PB7), and four control lines (CA1, CA2, CB1, CB2). Each of these lines can be either high (at approximately +5 volts) or low (approximately 0 volts). This is how VIA B reads the keyboard: A keypress grounds a connection to the VIA, and a zero value appears and is read. Similarly, VIA A reads a joystick at the controller port; again, a simple grounding action is used.

Generally, VIA programming simply involves arranging or comparing patterns of 0's and 1's, typically by ANDing or ORing data. The same programming principles apply in both BASIC and ML, apart from speed-sensitive processing, so the user port can often be controlled with POKEs and PEEKs.

The other major aspect of VIA programming is the interrupt handling. Each VIA has an IRQ (interrupt request) line which is normally high but can be brought low. If that line is connected to the 6502 processor, then whenever the VIA generates an interrupt the 6502 will process it.

Seven different events can trigger an interrupt, so the VIA has a register from which the processor can determine which event just took place. You can let interrupts take place upon either of two types of transition: *active low,* in which a transition from high (5 volts) to low (0 volts) triggers the interrupt, and *active high,* which is the opposite. If a triggering transition occurs, it is called an active transition. Transitions in the other direction are not considered active.

Figure 5-9 shows the internal arrangement of any VIA. It is helpful to take a systematic look at its features and at the names and abbreviations given to them.

VIA Bit Conventions

Fortunately, the 6522 (and other chips in the series) has a standardized set of conventions for its registers. A bit value of 1 has one of the following effects:

1. Sets a line high, to 5 volts
2. Configures a line for output
3. Defines an active transition as positive (0 to 1)
4. Shows that an active transition has been detected
5. Enables an interrupt to occur when a line receives an active transition

Of course, zero bit values have the opposite effect.

Ports

There are two ports, A and B. These are eight-bit registers held in a single byte; the individual bits, or lines, are labeled PA0–PA7 in port A and PB0–PB7 in port B. Sometimes the ports are called IORA and IORB (input/output register A and B).

Each bit can be configured for either input or output. It's often convenient to configure all eight identically (for instance, to scan the keyboard). If the configuration is for input, PEEKing the ports shows whether a bit is high (+5 volts) or low (0 volts). Each bit configured for output will set its line at 5 volts or 0 volts, depending on whether 1 or 0 is POKEd into the port's bit. Some hardware knowledge is needed to keep from drawing too much power from the chip.

Figure 5-9. General Diagram of VIA's Internal Arrangement

Bits	7	6	5	4	3	2	1	0	
Bit Address:	Port B								IORB
Bit Address +1	Port A (with CA2 handshake)								IORA
Bit Address +2	Data Direction Register for Port B								DDRB
Bit Address +3	Data Direction Register for Port A								DDRA
Bit Address +4	Timer 1 (low byte)								T1C-L
Bit Address +5	Timer 1 (high byte)								T1C-H
Bit Address +6	Timer 1 Latch (low byte)								T1L-L
Bit Address +7	Timer 1 Latch (high byte)								T1L-H
Bit Address +8	Timer 2 (low byte)								T2-L
Bit Address +9	Timer 2 (high byte)								T2-H
Bit Address +A	SHIFT Register								SR
Bit Address +B	Timer 1 Control		Timer 2 Control	SHIFT Register Control			Port B Latch	Port A Latch	ACR
Bit Address +C	CB2 Control			CB1 Control	CA2 Control			CA1 Control	PCR
Bit Address +D	IRQ flag on/off	T1 out	T2 out	CB1 trans'n	CB2 trans'n	SW-Reg out	CA1 trans'n	CA2 trans'n	IFR
Bit Address +E	interrupt enable/ disable	T1	T2	CB1	CB2	SW-Reg	CA1	CA2	IER
Bit Address +F	Port A (without CA2 handshake)								IORA

Port B appears before port A in the memory map. Even more confusingly, port A can be found in memory in two different places, and the same data appears in each. The difference is that one has a handshake effect with CA2 while the other does not, so when CA2 is irrelevant, the two are interchangeable.

Using control lines CA1 and CA2, ports A and B can be latched, meaning that their current value will be retained. For example, an active transition on CA1 freezes port A's contents until the next active transition on that pin. This is useful whenever a value to be read depends on some external event.

Data Direction Registers

DDRA and DDRB are the data direction registers for ports A and B. A value of 1 sets the corresponding bit in the port for output, while a 0 sets it for input. Thus, $00 configures the entire port for input, and $FF configures it for output.

Note that on switching on, the RESET line to the VIA causes all the internal registers to be set to zero, so the data direction is set for input. This prevents hardware connected to the chip from being turned on, however briefly.

Control Lines

Each port has two control lines. Port A has control lines CA1 and CA2, while port B has lines CB1 and CB2. Each control line has its bit connected to a pin on the chip, so the ports have ten bits each if all the lines are used.

The purpose of the control lines is, not surprisingly, to control the ports (for example, to determine the precise point at which reading or writing takes place). This is necessary, because inputs will hardly ever synchronize with VIC. CA1 and CB1 are always input lines; CA2 and CB2 may be configured either for input or output. Note that each VIA has its own ports and control lines. The VIC-20 has two CA1 lines, one for each VIA, so CA1 doesn't refer to a specific line, as you may be led to think.

A slightly tricky point is that the control lines' status is not present as a bit in the VIA. The programmer can set a line for input and wait to detect changes in its status through the interrupt facilities (or set a line for output and then set it high or low), but there's no bit which directly reveals the current high or low status of any control line.

Timers

Each VIA has two 16-bit timers, called T1 and T2, occupying two registers apiece. Generally speaking, timers have two functions. One, by far the most important, is timing in the usual sense; the other is counting, which isn't used in VIC. When a value is POKEd into a timer, it is immediatley set to decrement once every clock cycle. When any timer reaches zero, an interrupt flag is set if it has been enabled. With a 1 megahertz clock, the maximum time interval between interrupts is therefore about $FFFF millionths, or 1/15, second.

Timer T1 has a latch feature, actually another 16-bit register which stores T1's value. When T1 reaches zero, the latched value is reloaded and the process repeated. Thus, an indefinite series of regularly repeated interrupts is possible, though with a maximum delay between interrupts of only 1/15 second. This is how the keyboard interrupt is generated.

In this way, T1 takes up four bytes and T2 takes up only two. The rule is that reading the low byte of either timer (but not the latch) clears its own interrupt flag, while writing to the high byte clears the flag and starts the timer decrementing. One-shot interrupts, which time out only once instead of continually repeating, can be used for such purposes as tracing machine language (where an interrupt can be made to occur before the next ML command is complete) or for timing the response of equipment which may not be connected. In precision timing like this, the low byte is POKEd into the timer first. Then the high byte is POKEd to start the countdown.

If T2 is set for input, it doesn't time in the manner just described. Instead, it counts the number of pulses received on line 6 of port B and gives an interrupt signal when a preassigned number has been detected. As mentioned, the VIC doesn't use this facility.

T1 has an analogous feature: It can be set to output a pulse to line 7 of port B. In free-running mode this continually toggles (reverses) PB7.

The Shift Register

This eight-bit register is connected to CB2. On command, it performs eight shifts, having the effect either of moving eight bits singly to CB2 or (if configured for input) of inputting eight bits from CB2 one at a time into the shift register. The shift out is analogous to the machine language instruction ASL, with the high bit moved first into CB2, then into bits 6, 5, 4, and so on, so the byte is output in serial form. Shifting in is analogous to ROR on the 6502.

The shift register can be timed by T2, or at the same rate as the 6502 using the so-called phase two clock. Alternatively, another external clock may be used. This is a versatile register, which in principle extends the VIC's usefulness. However, it contains a number of bugs and has to be treated with caution. For example, start-up delay is dependent on instruction sequence and timing.

Control Registers of the VIA

Three bytes control the configuration of everything about the VIA, from timers and shift register to ports, control lines, and interrupts. The interrupt flag register (IFR), which can be regarded as a fourth control register, hasn't the same status as the others, since it only provides a record of what's happened without actually controlling events.

Auxiliary Control Register (ACR). This register controls the timers, the shift register, and the latch status of ports A and B. Figure 5-10 reflects the conceptual arrangement of bits 7–0. The shift register control has three bits, and therefore eight combinations, which explains its prominence.

Figure 5-10. Auxiliary Control Register

ACR7	ACR6	ACR5	ACR4 ACR3 ACR2	ACR1	ACR0
Timer 1 Control		Timer 2 Control	SHIFT Register Control	Port B Latch	Port A Latch
0=PB7 Unused	0=One Shot	0=One Shot	000=SHIFT Reg. Disabled	0=Disabled	0=Disabled
1=Output to PB7	1=Continuous	1= Count Set No. of PB6 Pulses	001=SHIFT In by Timer 2	1=Enabled on CB1 Transition (In/Out)	1=Enabled on CA1 Transition (In)
			010=SHIFT In, Syst. Clock		
			011=SHIFT In, Ext. Clock		
			100=Free RUN by Timer 2		
			101=SHIFT Out by Timer 2		
			110=SHIFT Out, Syst. Clock		
			111=SHIFT Out, Ext. Clock		

Peripheral Control Register (PCR). This register controls the operating modes of the four control lines CA1, CA2, CB1, and CB2. CA1 and CB1 are each allocated one control bit in this register; their function is to set active transitions high or low. However, CA2 and CB2 are much more complex and have three control bits each.

If either CA2 or CB2 is set for input, the active transition may be set high or low. There's also a choice of two methods to allow the flag in IFR to be cleared: POKEing a 1 into IFR0 or IFR3, or simply reading or writing to port A or B.

The second option is often very useful. If CA2 or CB2 is set for output, manual mode allows the line to be set high or low. Configurations of the form 10x allow for handshaking. For instance, 100 sets CA2 low when port A is written to or read from and resets high on an active transition of CA1, while 101 sets CA2 low for just one cycle. Port B is similar.

Figure 5-11 diagrams this register.

Figure 5-11. Peripheral Control Register

PCR7	PCR6	PCR5	PCR4	PCR3	PCR2	PCR1	PCR0
CB2 Control			CB1 Control	CA2 Control			CA1 Control
Direction: 1=Out	Port Read/Write =0	CB2 Pulse =1 CB2 Hand-shake=0	Active Trans'n: High=1 Low=0	Direction: 1=Out	Port Read/Write =0	CA2 Pulse =1 CA2 Hand-shake=0	Active Trans'n: High=1 Low=0
	Manual=1	CB2 Hi=1 CB2 Lo=0			Manual=1	CA2 Hi=1 CA2 Lo=0	
0=In	Active High=1 Low=0	Clear IFR only=1 IFR or B=0		0=In	Active High=1 Low=0	Clear IFR only=1 IFR or A=0	

Figure 5-12. Interrupt Flag and Enable Registers

IFR7	IFR6	IFR5	IFR4	IFR3	IFR2	IFR1	IFR0
1-IRQ 0=No IRQ	Timer 1 Timed Out	Timer 2 Timed Out	CB1 Trans'n Detected	CB2 Trans'n Detected	SHIFT Complete	CA1 Trans'n Detected	CA2 Trans'n Detected

IER7	IER6	IER5	IER4	IER3	IER2	IER1	IER0
1=1 Enables 0=1 Disables	Timer 1	Timer 2	CB1	CB2	SHIFT Rgr	CA1	CA2

Interrupt Flag Register (IFR) and Interrupt Enable Register (IER). These registers are symmetrical with respect to each other and can be considered together. The first indicates whether an interrupt request has occurred, and if so, which VIA device caused it. This is important, for it allows the interrupt processing to call the appropriate interrupt servicing routine. These flags are cleared by reading (or writing) the corresponding registers (though, as noted in PCR above, CA2 and CB2 may be exceptions). This explains why VIC's ROM occasionally has seemingly pointless reads of some VIA registers.

IER7 controls the function of the rest of IER. When IER7 is 0, each bit set to 1 clears its corresponding interrupt enable; when IER7 is 1, each bit set to 1 sets its interrupt enable bit. Thus, $7F first clears all interrupts; then $81 enables CA2 interrupt. Figure 5-12 will help make this clear.

VIAs have a feature not found in ordinary memory: Reading from some registers alters others. This is useful with interrupts—reading data automatically clears flags—but displays of the VIA's contents aren't entirely accurate. A potential problem can arise because of this. Its solution requires some ML knowledge. A POKE (or PEEK) to certain VIA locations has the effect of reading them first, so the result differs from that obtained by a simple absolute address mode ML command like STA $9020. The reason: Both commands use indirect addressing, which in effect calculates the address before POKEing. This can be avoided with a ML subroutine using absolute addressing to POKE in values.

The User Port

Before looking at the VIC's usage of the two VIAs, it is helpful to briefly discuss the user port. This is a 24-pin port at the back of the VIC, which doubles as the RS-232 port. The pinout, as it appears from the back of VIC, is shown in Figure 5-13.

Figure 5-13. VIC User Port

GND	+5VDC	RESET	Joy 0 PA2	Joy 1 PA3	Joy 2 PA4	Light Pen PA5	Cassette Button PA6	Serial Atn In	+9VAC	+9VAC	GND
1	2	3	4	5	6	7	8	9	10	11	12
A	B	C	D	E	F	H	J	K	L	M	N
GND	CB1	PB0	PB1	PB2	PB3	PB4	PB5	PB6	PB7	CB2	GND

All the port and control lines apply to VIA A. Lines A to N are the most important. Be aware that some VIC manuals erroneously describe one of the 9-volt AC supply lines as ground.

VIC's VIAs

Figure 5-14 shows the registers of VIA A. However, since the contents of those registers can vary, such diagrams are not very satisfactory. DDRA, which corresponds to port A, is normally configured as shown, with most bits set for input. The ACR values are also normal: Timer 1 generates continuous interrupts, timer 2 gives one-shot output, and neither port is latched.

Note that the cassette motor control uses manual output, where CA2 high switches the motor off, and vice versa. Thus, PCR is either xxxx111x (motor off) or xxxx110x (motor on). Pressing the cassette PLAY button appears to switch on the motor like an ordinary recorder, but the motor is in fact controlled by VIC.

Note that this VIA's Interrupt Request line is connected to the 6502's NMI line. The RS-232 handler uses NMIs in its processing.

Figure 5-14. VIA A Registers

	Bit 7	6	5	4	3	2	1	0	
$9110 (37136)	User Port or RS232 Port:								
	Data Set Ready	Clear To Send	Unused	Received Line Signal	Ring Indicator	Data Terminal Ready	Request To Send	Received Data	Port B
$9111 (37137)	Other Port A Mostly Used								Port A
$9112 (37138)	(Varies)								DDR B
$9113 (37139)	1	0	0	0	0	0	0	0	DDRA
$9114 (37140) $9115 (37141)	RS-232 Output Timer of Tape Write Timer								T1
$9116 (37142) $9117 (37143)	(Sets repeating value in Timer 1)								T2 Latch
$9118 (37144) $9119 (37145)	RS-232 Input Timer								T2
$911A (37146)	SHIFT Register—Not Used								SR
$911B (37147)	0	1	0	0	0	0	0	0	ACR
$911C (37148)	User Port Pin M Control			User Port Pin B Control	Cassette Motor Control			Restore Key Control	PCR
$911D (37149) $911E (37150)	Interrupt Flag	RS-232 Out or Tape Write	RS-232 Input	User Port Pin B	User Port Pin M		Restore Key		IFR IER
$911F (37151)	Serial ATN Out	Cassette Button	Joy Fire	Joy 0/ Paddle Fire	Joy 1	Joy 2	Serial Data In	Serial Clock In	Port A

CA1—RESTORE Key
CA2—Cassette Motor
CB1—User Port Pin B/RS-232 Receive Data
CB2—User Port Pin M/RS-232 Transmit Data

VIA B

This chip controls most of the keyboard and tape interfacing, as well as most of the serial communications (primarily with disks and printers). Again, ACR sets T1 to repeat; changing the latched value will alter the interrupt rate. The rate is increased when tape is read, which is why the built-in clock gains time. DDRA and DDRB are left as shown at all times, unless RS-232 or the joystick is read, but note that a bug will occur if a key is pressed while the joystick is being pulled to the right.

Figure 5-15 diagrams VIA B.

Figure 5-15. VIA B Registers

Bit	7	6	5	4	3	2	1	0	
$9120 (37152)	RS-232 In Joy 3				Tape Out				Port B
$9121 (37153)	Keyboard Input								Port A
$9122 (37154)	1	1	1	1	1	1	1	1	DDRB
$9123 (37155)	0	0	0	0	0	0	0	0	DDRA
$9124 (37156) $9125 (37157)	Keyboard Interrupt or Tape Read Timer								T1
$9126 (37158) $9127 (37159)	Sets Repeating Value in Timer 1								TI Latch
$9128 (37160) $9129 (37161)	Serial Device Time Out or Tape Write Timer								T2
$912A (37162)	SHIFT Register—Not Used								SR
$912B (37163)	0	1	0	0	0	0	0	0	ACR
$912C (37164)	Serial Data Out Control			Serial SRQ In Control	Serial Port Clock Out Control			Cassette Read Control	PCR
$912D (37165) $912E (37166)	Interrupt Flag	Keybd. Interrupt or Tape Read Time	Serial Time Out or Tape Write Time	SRQ In			Tape Read Data		IFR IER
$912F (37167)	Other Port A Mostly Used								Port A

CA1—Cassette Read Line
CA2—Serial Port Clock Out
CB1—Serial SRQ In
CB2—Serial Data Out

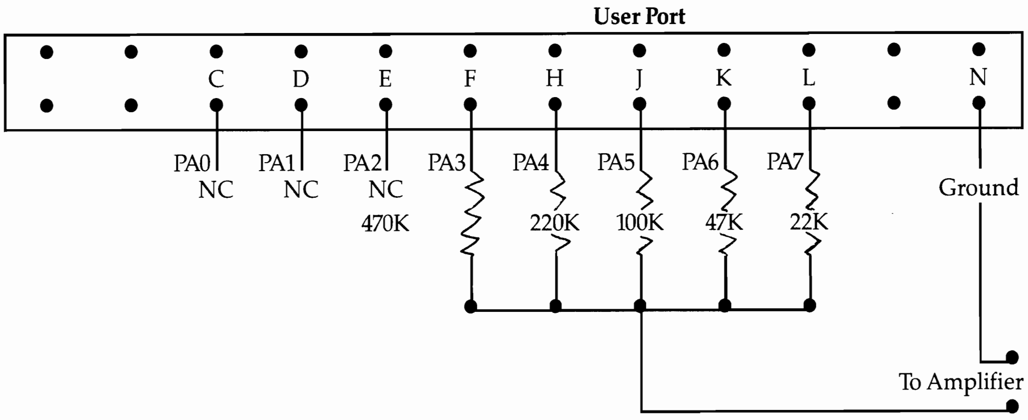
Examples of VIA Programming

Here are some simple programming experiments—about as simple as they can be—that let you hear the results of VIA programming. All you need is a 24-pin edge connector to fit the user port and lengths of wire to be soldered to the edge connector, plus (for some demonstrations) a small transistor radio or an audio amplifier. Caution: Don't inadvertently connect power lines to data lines, or you may damage your VIC.

CB2 line sound. This is an old favorite. Connect a wire to pin N of the edge connector; this is line CB2. Wrap the other end of the wire around the radio several times. CB2 is connected to the shift register of VIA A, and the radio can detect output square or rectangular waves generated by the shift register. If you type in and run Program 5-11, you should hear a glissando effect. Note that the order of lines 10 and 20 is important; reversing them doesn't work. Obviously, it'll sound the same if a value such as 120 (%01111000) is POKEd in line 20. The value 51 (%00110011) sounds as though it has twice the frequency, and 85 (%01010101) produces four times the frequency. In fact, there are 17 fundamentally different bit patterns, not including repetitions and inversions; their smallest decimal values are 1, 3, 5, 7, 9, 11, 15, 17, 19, 21, 23, 27, 37, 43, 45, 51, and 85.

User port sound with 8-bit resolution. Roughly sinusoidal waveforms can be generated from the user port with the aid of the simple digital-to-analog converter shown in Figure 5-16.

Figure 5-16. Simple Digital-to-Analog Converter



The following simple machine language program will produce audio signals at the output of the D/A converter. It needs a table of 256 bytes POKEd into RAM. For example, I=0: FOR J=5000 TO 5255: POKE J,128+125*SIN(I):I=I 2* π/256: NEXT gives a sine wave output, by storing one cycle of a sine curve over 256 locations.

```
     LDA  #$FF
     STA  $9112       ; CONFIGURE PORT B FOR OUTPUT
LOOP LDA  TABLE,X     ; TABLE OF 256 VALUES IN RAM
     STA  $9110       ; OUTPUT VALUE TO USER PORT
     INX
     BNE  LOOP
     LDA  $91
     CMP  #$FE        ; TEST STOP KEY
     BNE  LOOP
     BRK  (or RTS)
```

PB7 square wave for clocking. Pin L of the user port is PB7, which is related to timer 1. If ACR is 11xxxxxx, output to PB7 continually toggles when timer 1 times out, generating a square wave. A wire to pin L will give you a square wave, using Program 5-12.

Program 5-12. Square Wave Generator

```
10 POKE 37147,191
20 POKE 37142, LO:REM LOW LATCH VALUE
30 POKE 37143, HI:REM HIGH LATCH VALUE
40 POKE 37141, HI:REM START TIMER 1
```

Changing the IRQ rate. The IRQ rate is readily changed. Simply alter the latched value in $9126 and $9127 (37158 and 37159).

Disabling RESTORE. $911E (37150) normally holds $82 (130). The value of 2 in the lower half-byte is the RESTORE key flag, since CA1 is connected to RESTORE. The flag can be reset to zero by storing $7F in the register, which turns off all interrupts.

Storing $02 turns off only the RESTORE key, leaving other interrupts intact (although, in practice, there usually are no others). POKE 37150,127 or POKE 37150,2 therefore disables RESTORE. POKE 37150,130 returns operation to normal. This is only one example of setting interrupt flags; as the diagram of VIA A shows, pin B or pin M can be made to cause interrupts on an active transition. Thus, to take just one illustration, your VIC could count external events by making them trigger interrupts (which in turn could be counted by the interrupt routine written for the purpose).

Using cassette read and write lines. A connection to the cassette read line (line 4 on the cassette port) lets you aurally check tape reading. You will hear a harsh, raspy noise. Program tapes played on conventional tape recorders produce a similar sound.

If you monitor the write line (line 5 on the cassette port), you'll hear the tone and subsequent program or file recording.

Checking the cassette button and switching the cassette motor on and off are explained in Chapter 14. So is the possibility of converting normal recorders to work with VIC.

"Tape Talker," the following short machine language routine, demonstrates several interesting aspects of the VIA by attempting to play ordinary tape recordings. Some tones, like whistled tunes, are reproduced fairly recognizably, and some recorded voices may be comprehensible. The program might actually be useful for copying digital tapes.

```
     SEI          ; INTERRUPT PROCESSING ISN'T REQUIRED
     LDA #$7F     ; 0111 1111 TURNS OFF ALL INTERRUPTS:
     STA $912E    ; STA IERB TURNS THEM OFF IN VIA B.
     STA $911E    ; STA IERA TURNS THEM OFF IN VIA A.
     LDA #$82     ; 1000 0010 ENABLES AN INTERRUPT:
     STA $912E    ; STA IERB ON CA1 TRANSITION (I.E., TAPE READ) IN VIA B,
                  AND
     STA $911E    ; STA IERA ON CA1 TRANSITION IN VIA A (I.E., RESTORE KEY)
LOOP LDA #$01     ;
     STA $912C    ; STA PCRB SETS ACTIVE TRANSITION ON TAPE READ HIGH
     LDA #$EC     ; 111 0 110 0 IN PCRA; 111 SETS CB2 OF VIA A = PIN M OF
     STA $911C    ; STA PCRA; USER PORT, HIGH. (110 SETS TAPE MOTOR ON.)
     LDA #$02
W1   BIT $912D    ; BIT IFRB WAITS TILL IFRB SHOWS AN INTERRUPT HAS
                  OCCURRED
     BEQ W1       ; ON TAPE READ TRANSITION.
     STA $912D    ; STA IFRB; STORING BIT LIKE THIS CLEARS THE INTERRUPT
                  FLAG.
     LDA #$00
     STA $912C    ; STA PCRB SETS ACTIVE TRANSITION ON TAPE READ LOW.
     LDA #$CC     ; 110 0 110 0 IN PCRA; 110 SETS CB2 OF VIA A = PIN M OF
     STA $911C    ; STA PCRA; USER PORT LOW (110 KEEPS TAPE MOTOR ON)
     LDA #$02
W2   BIT $912D    ; BIT IFRB WAITS TILL IFBR SHOWS ANOTHER INTERRUPT
                  OCCURRED
     BEQ W2       ; ON TAPE READ TRANSITION IN THE OTHER SENSE.
     STA $912D    ; STA IFRB CLEAR THE INTERRUPT FLAG.
     JMP LOOP     ; CONTINUE WITH LOOP.
```

Here's how it works. The cassette read line is CA1 of VIA B. The motor is turned on, and tape signals are input on CA1. Then, when CA1 is high, the program sets pin M of the user port (bottom, next to the right, as seen from the back of the VIC) high; when it's low, the pin is set low. Pin M is a CB2 line, and is therefore controllable in manual mode. The original tape is output in a simplified square-wave form which only considers transitions from high to low or vice versa; that is why definite tones can be reproduced with some faithfulness. Try monitoring pin M of the user port. Note that the RESTORE key is also allowed to interrupt, so the program doesn't loop indefinitely. It is pure coincidence that $82 sets both interrupts.

EPROMs. Although the user port has no address lines, it is possible to connect 11 of its lines to a 2K EPROM socket and treat them in software as though they were address lines. EPROMs can then be burned to save your own ML programs. An accurate external power supply of 25 volts (not obtainable directly from the VIC) is necessary.

An EPROM can be read back through the user port, but it can't be run as a program since it doesn't use the normal addressing. To run an EPROM, it must be connected to the VIC's memory expansion system.

Ports and External Adapters

You have seen that the user port doubles as a sort of RS-232 port. RS-232 is a standard for transmitting serial data; the alternative transmission system is parallel transmission, where several (typically eight) bits are sent together. As might be expected, serial transmission tends to be slower, but parallel transmission is more expensive in terms of hardware. Serial transmission has another advantage: It's easier to transmit reliably over long distances.

An RS-232 connector has 25 pins arranged in two rows, although the VIC isn't equipped with a direct connector. To keep the costs down, VIC's port is part of its main board, so some adapter is necessary. RS-232 and RS-232C standard devices are designed to operate with ideal voltages of −12 volts and +12 volts, although the working range is much larger (3 to 20 volts, with luck).

RS-232 and RS-232C differ in their pinouts. Acoustic modems generally use RS-232C, so they must have an adapter which allows for that configuration. VIC's RS-232 capability is achieved mostly through software; VICTERM, for example, has only one reference to the VIA and relies mostly on ROM routines. RS-232 devices are assigned device number 2 when a file is opened with a statement like OPEN 10,2,0,baud rate. Any subsequent input or output (by GET#10, for example) uses two buffers at the top of BASIC memory for storage, and the VIAs are used for moving data through the user port.

VIC's other input/output port, the six-pin serial port, is located next to the cassette port and operates VIC disks and printers. It is nonstandard, and is in fact a modification of the IEEE interface system found in PET/CBM machines. IEEE-488 is a 24-pin parallel transmission standard which CBM hardware uses in a slightly modified form. Commodore's 8050 double-density dual disk drives and the discontinued 4040 dual disk drives (compatible with VIC 1540 format) are typical of these IEEE devices; however, VIC cannot simply plug into them. These devices are faster than VIC disks and offer some other advantages, but they aren't widely used items because of cost.

Yet another type of port, for printers, is the Centronics interface. This is a 36-pin parallel system. Again, VIC needs some sort of adapter to handle this. Many printer users have VIC printers which can be plugged directly into the serial port, but the attraction of Centronics compatibility is that many fast and high-quality printers are equipped with Centronics interfaces.

Adapters and Interfaces

These words are more or less synonymous and refer to a so-called black box which connects VIC to some other piece, or pieces, of equipment. For example, adapters which allow several VICs to use the same printer or disk drive are available, and these can be very useful in certain settings. However, some caution is required. The easiest interfacing methods don't allow for simultaneous operation, so (for example) users must warn each other when they're about to write to disk. In addition, the hardware doesn't react instantaneously; it takes a bit of time for half a dozen people to read a single disk.

Several IEEE adapters are on the market. One type plugs into the serial port, converting VIC's modified IEEE signals into a form acceptable to IEEE devices, and vice versa. The black box contains an external power supply plus ROM and RAM, with an IEEE socket and another serial socket so that data can be transferred from one type of device to another. This system has the advantage of being *transparent* (not interfering in any way with normal operations). But since VIC software will not normally expect IEEE devices to be present, programs to use the system need to be specially written. Thus, this may not be as great an advantage as it seems.

A different, somewhat less expensive approach is to put the hardware on a board fitting into the expansion port. Typically, such a board contains a duplicate of the expansion port (so RAM expanders can still be inserted) and also has an IEEE socket. If it also has a switch to turn itself off, this is an advantage, because the board needn't be taken out to run other software. The actual IEEE translation program is likely to be stored at $A000 (40960); any hardware which prints a sign-on message when VIC is turned on is this type. Alternatively, a SYS call will activate it. The software almost always works by changing the input and output vectors around ($0300), so SYS 64850 (which restores the normal vectors) kills the IEEE feature.

Centronics printers can be easily interfaced with the VIC. Port B on the user port, plus control lines CB1 and CB2 to arrange handshaking, provides a convenient parallel port, so all that's needed in hardware is a Centronics cable fitted with an edge connector for the user port. Special software is necessary to control the lines, in contrast with RS-232 equipment. This type of system can be excellent if the software is bug-free; if it isn't, a tiresome process of repeated reloading of software may be necessary.

Another type of adapter is a set of relays, connected to the user port, which allows the VIC (with suitable software) to control external equipment. BASIC, or a set of BASIC subroutines, may well be adequate. In principle, since there are eight or so output lines, approximately 200 relays could be controlled.

The Audio-Video Port

This five-pin port has a single audio line, two very similar video lines, and a voltage line plus ground. The outputs can be used directly by a video monitor such as the Commodore 1701/1702, or by a television with the RF modulator module supplied with the VIC. For higher-quality sound, you could even make a connection from the audio output line to an input of your home stereo system.

Some Notes on Commercial Software

Most of these notes apply to games; similar comments apply in principle to all other software.

Cartridges

Cartridges almost always have a 4K ROM or EPROM (at $A000–$AFFF) in order to autostart. Sometimes that is all there is, but where 4K isn't adequate, another chip must be used. Most Commodore cartridge games have a second chip at $B000; this is popular with non-CBM cartridges too. ROM can be elsewhere (for example, at $6000), providing some protection against disassembly by VICMON, which also starts at $6000. Cartridges tend to look a little lean if you open them up to look inside—just a small board with a couple of chips.

Most non-Commodore cartridges have protection systems designed to prevent a copy of the cartridge in RAM from working properly. For example, in a copy of a program loaded into RAM at $A000, a simple ML instruction like INC $A300 will increment the contents of $A300, if $A300 is RAM, but will have no effect on ROM. In this way, the fact that RAM can be written to can be exploited as a form of protection, with the program writing into chunks of itself. However, if the write line is disconnected, RAM in effect becomes ROM. It is possible for the software to test for time elapsed since reset, but if the program is stored with battery backup, this provides no security either. Mixing RAM and ROM on the cartridge seems to be the only method to make copying difficult.

There are valid reasons to want RAM versions of one's own cartridge software, mainly so that small modifications can be put in (which is of course impossible with ROM). An appealing idea is to alter games, so that instead of a paltry three or four spaceships you have 250 or 500. Modifications can be very easy, even for non-ML programmers. For instance, if three rockets are allowed, the programmer probably used LDA #3/ STA 10 or something similar. All that's needed is to replace 3 with 255, and a superlong game is yours. BASIC programmers can look for the sequence 169, 3, 133 (this applies where there are three objects; if there are six, look for 169, 6, 133, and so on). Enter FOR J=40960 TO 12*4096: IF NOT (PEEK(J)=169 AND PEEK(J+1)=3 AND PEEK(J+2)=133) THEN NEXT. Then, when it stops, try POKE J+1,255 with a RAM game. If you're lucky, you'll be rewarded with a much longer game. If it doesn't work, replace 40960 by the value you found for J+1, and try again.

The following POKEs work with the specified programs when they are copied into RAM, but there may be other versions too. For *Galaxians*, POKE 41153,255. For *Gorf*, POKE 47780,255. POKE 41200,255 for *Pac-Man*. For *Ratrace*, POKE 45031,255, and for *Titan*, POKE 42627,255. POKE 44790,255 gives 255 helicopters in *Choplifter*. There are many possible variations, of course; for example, *Gorf* uses LDX rather than LDA.

Games can be made to last forever if the counter is disabled. For instance, POKE 43560,0 makes *Cloudburst* continue indefinitely.

The VIC poses some problems of transatlantic compatibility, as you noticed in the VIC chip section. For example, programs from the United Kingdom are likely to be off the right side of the screen when used in the United States, and vice versa. But this too is often easily cured, simply by hunting for ML operations which set the screen positions and altering their parameters.

Commodore's Adventure series. The *Adventure* cartridges are unusual in not using the autostart feature. They occupy 16K, from $4000 to $7FFF, and are activated by SYS 32592. Their tables of words aren't coded in any way, so Program 5-13 can be used to go through ROM and output phrases and words when it reaches them. Typically, the output will be NOR for north, IDO for idol, KNI for knife, and so on.

Program 5-13. *Adventure* Game Searcher

```
10 FOR J=16467 TO 32767
20 P=PEEK(J):IF P=0THEN P=ASC("/")
30 PRINT CHR$(P);
40 WAIT 197,32
50 NEXT
```

Tape Software

Tape software has a single overriding characteristic: It is slow to load. It's worth noting that some tape software is in fact in BASIC; sometimes you'll even see programs claiming to be in machine language which are not.

Tape software is usually modifiable by loading it into RAM with a special loader, editing it, and dumping it back onto tape. This technique is explained in Chapter 14.

Generally, it isn't possible to automatically put tape software onto disk, except in the case of straightforward BASIC or ML programs. Any protection method, such as the use of the tape buffer to hold ML, requires either that the protection be removed or that an identical copy of both parts of the program to be loaded off disk. Any program chaining won't work; the device number would need to be changed from 1 to 8. In these examples, again excepting simple cases, some ML knowledge is necessary.

Chapter 6

Beyond VIC BASIC

Chapter 6

Beyond VIC BASIC

This chapter explains the advanced BASIC capabilities which the VIC-20 has to offer. It will be particularly valuable for programmers who want to optimize their work and avoid some of the more subtle programming bugs.

The chapter is divided into four sections.

Storage of BASIC. Shows exactly how BASIC, along with its variables, strings, and arrays, is stored in RAM. Discusses accuracy, garbage collection in strings, LOADing one program from another, calculating memory requirements for variables, and other common queries.

Special features of BASIC. This section looks at buffers and the screen and includes a listing of useful locations. It also examines the keyboard, showing how to disable STOP or RESTORE, control automatic key repeats, read keys without GET, and modify the keyboard. A program for function keys is included.

Alphabetic list of BASIC extensions. This section includes useful routines to BLOCK LOAD and BLOCK SAVE from within BASIC. It also includes routines to DUMP (display current variables), MERGE (interlink programs), OLD (recover programs), PRINT USING (format decimal numbers), RECONFIGURE (adjust VIC's memory), SORT (arrange data in sequence), and UNLIST (provide some BASIC program security).

Detailed description of Programmer's Aid.

How BASIC Is Stored in Memory

Major Memory Locations Controlling VIC-20 BASIC

The previous chapter explained how pointers controlling BASIC are set up when the VIC-20 is turned on. These pointers, in decimal, are summarized below.

START OF BASIC RAM = PEEK(641) + 256 * PEEK(642)
START OF BASIC PROGRAM = PEEK(43) + 256 * PEEK(44)
END OF PROGRAM + 1 = PEEK(45) + 256 * PEEK(46) = START OF SIMPLE VARIABLES
END OF SIMPLE VARIABLES + 1 = PEEK(47) + 256 * PEEK(48) = START OF ARRAYS
END OF ARRAYS + 1 = PEEK(49) + 256 * PEEK(50)
CURRENT BOTTOM OF STRINGS = PEEK(51) + 256 * PEEK(52)
PREVIOUS BOTTOM OF STRINGS = PEEK(53) + 256 * PEEK(54)
END OF MEMORY USABLE BY BASIC = PEEK(55) + 256 * PEEK(56)
END OF BASIC RAM = PEEK(643) + 256 * PEEK(644)
START OF SCREEN MEMORY = 256 * PEEK(648)
START OF COLOR RAM = 37888 + 4 * (PEEK(36866) AND 128)
START OF CURRENT TABLE OF CHARACTER GENERATOR BITS = 32768 * (1 + ((PEEK(36869) AND 8) = 8)) + 1024 * (PEEK(36869) AND 7)

The main set of pointers occupies 14 consecutive bytes, from locations 43 to 56, an arrangement found in all Commodore BASICs. Most of these pointers mark a boundary between one type of item and the others, which is why END OF PROGRAM + 1 = START OF VARIABLES (and so on).

All this means, in this case, is that the program ends one byte before the pointer and that the variables start exactly at the pointer, so there's no overlap. A convention like this is obviously necessary, and this pattern is general to Commodore; this is the reason why the last byte can easily be forgotten when saving ML programs.

These pointers are the most important when considering BASIC on its own, and much of this chapter is concerned with them, directly or otherwise. For completeness the list also includes other pointers set by VIC-20 on switch-on. The start and end of BASIC RAM are stored in an extra set of these pointers when the position of the screen is set, but these have little function and are less useful than the main pointers.

Oddly enough, there are no pointers delimiting the whole available RAM. The screen and character generator pointer values are dependent on the VIC chip, as you saw in Chapter 5. Setting their values is relatively complex, since they aren't ordinary two-byte pointers.

Experimenting with BASIC Storage in Memory

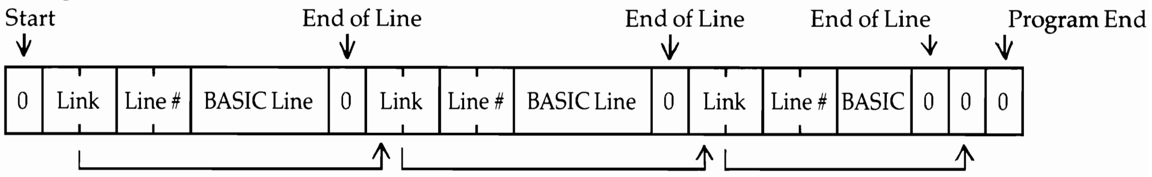
As in introduction to BASIC pointers, switch on an unexpanded VIC-20 and PRINT some two-byte PEEKS, starting with PRINT PEEK(641) + 256 * PEEK(642). You should find that the BASIC RAM extends from 4096 ($1000) to 7680 ($1E00). The screen starts at 7680 ($1E00). The VIC-20 allows 512 bytes for the screen (22*23 = 506, and 512 is the smallest unit by which the screen can be moved), so the screen ends at 8185 ($1FF9) with six bytes left over (8186–8191). These can be used to store a small amount of data. If a top-of-RAM pointer existed, in this case it would hold $2000 (8192).

You will also find that the actual start of BASIC is one byte beyond the start of BASIC RAM. In other words, 43 and 44 point to 4097 ($1001). This is because BASIC always starts with a zero byte; PRINT PEEK(4096) should be 0. The BYTES FREE message in this case has already calculated that bytes 4097–7679 (total of 3583) are available for BASIC. The pointers to end of program, simple variables, and arrays are all the same (4099). Any program has two consecutive zero bytes marking its end; as there are not yet any variables, all these pointers are in their starting positions (just after the program) which is where variables will be stored. Zero (or *null*) bytes are convenient for markers because they are easy to test for in ML. Thus, PEEK(4097) and PEEK(4098) both return 0 at present.

Each line of BASIC starts with a two-byte link address, a two-byte line number, the BASIC line itself, and a final zero byte which also marks the start of the following line. The link address is simply a pointer to the next line; in fact, it points to the next line's link address, forming a chain (Figure 6-1) which can be scanned at high speed. Each link needs two bytes. The line numbers also have two bytes, to allow for line numbers greater than 255.

Figure 6-1. Storage of BASIC as Linked Lines

Storage of BASIC as Linked Lines



The built-in operating system automatically arranges BASIC like this as lines are typed in at the keyboard. When the method is fully understood, you can modify BASIC to produce nonstandard effects. For instance, you can use it to insert longer-than-usual lines, to add normally unavailable line numbers, or to arrange a line of BASIC to contain things it ordinarily could not.

To begin, it is helpful to see standard BASIC storage in action. Type in Program 6-1.

Program 6-1. Bytes in a BASIC Line

```
1 PRINT "HELLO"
30 FOR J=4096 TO 4110
40 PRINTJ;PEEK(J);CHR$(PEEK(J))
50 NEXT
```

Run it and you will get this output, without the comments:

```
4096 0        ; Zero byte at the beginning
4097 15       ; Link address; points to start of next line
4098 16         at 4111 = 15 + 16 * 256
4099 1        ;Line number; this line number is 1
4100 0          = 1 + 0 * 256
4101 153      ;Tokenized form of PRINT
4102 32       ;Space
4103 34   "   ; Quote and following characters in PET ASCII
4104 72   H
4105 69   E
4106 76   L
4107 76   L
4108 79   O
4109 34   "
4110 0        ; End-of-line null byte
```

This program shows the contents of every byte of a single typical line of BASIC. Notes are included to show the function. Incidentally, you will get slightly different results if you make small changes to line 1. For example, if you remove the space between PRINT and "HELLO" in line 1, it will be missing from location 4102, as you would expect. In addition, the link address pointer will point to 4110 instead of 4111, because the next line starts a byte earlier. An extra space increases the link pointer. Similarly, try a different line number in place of 1; you'll see it reproduced in locations 4097 and 4098 when the program is RUN. Conversely, POKE 4100,2 will convert the line number to 1 + 2 * 256 = 513, despite the fact that this number is out of sequence.

BASIC lines have five bytes of overhead, consisting of the two-byte link address pointer, the two-byte line number, and the ending null byte. The end of BASIC is marked by two zero bytes—that is, when a link address is found to be zero, RUN and LIST will automatically treat it as an END and return to direct mode with READY. Try POKE 4111,0: POKE 4112,0 with the program above. It will LIST only one line. Then POKE 4111,35: POKE 4112,16 which will replace the original values (if the spacing was identical to the version above), and LIST shows the entire program again.

Normally the end-of-BASIC pointer in 45 and 46 will point just after these three consecutive zeros. If you think about it, you will realize that only the second of the pair of terminating bytes is necessary to signal an end. It is of course possible that the low byte of a link address could be zero. But any normal BASIC line will be in $0400 at the absolute minimum, so its high byte will never be less than 4.

Watching BASIC

There are several ways to watch BASIC and its variables as they do their work. Program 5-1 from the previous chapter, for example, can display a dynamic picture of BASIC's storage and is an excellent way to get the feel of BASIC lines, variables, and strings.

An alternative way is to move the screen. On the unexpanded VIC, POKE 648,16:SYS 64818 to move the screen to coincide with BASIC's start. Then {HOME}{RVS}{BLK} with spaces, followed by SHIFT-RETURN, prepares the screen for a dynamic display of small amounts of BASIC.

Another way to display a BASIC program's contents is to POKE it to the screen. The following routine can be added to a program on an unexpanded VIC-20. RUN 50000 finds the start and end of BASIC and puts all the bytes between those values directly into the screen memory. Press RUN/STOP–RESTORE to recover the white screen background.

Program 6-2. POKEing BASIC to the Screen

```
50000 POKE 36879,8:REM BLACK BACKGROUND
50010 FOR J=PEEK(43)+256*PEEK(44)TO PEEK(45)+256*P
      EEK(46)
50020 POKE 7680+Q,PEEK(J):Q=Q+1:NEXT
```

Table 6-1 shows the significance of each BASIC byte except for the links and line numbers). Note that all BASIC keywords are stored as a single byte with bit 7 set (that is, with value 128 or more in decimal). This means that the ML routine can instantly detect a keyword. It also means that when BASIC is poked to the screen, lines of BASIC have their keywords converted into a single character which appears in reverse on the screen.

Table 6-1. Internal Storage of VIC BASIC Bytes

0-31	32-	64-	96-	128-	160-	192-	224-255
	32 20 sp			128 80 END	160 A0 CLOSE	192 C0 TAN	
		65 41 A		129 81 FOR	161 A1 GET	193 C1 ATN	
	34 22 "	66 42 B		130 82 NEXT	162 A2 NEW	194 C2 PEEK	
	35 23 #	67 43 C		131 83 DATA	163 A3 TAB(	195 C3 LEN	
	36 24 $	68 44 D		132 84 INPUT #	164 A4 TO	196 C4 STR$	
	37 25 %	69 45 E		133 85 INPUT	165 A5 FN	197 C5 VAL	
		70 46 F		134 86 DIM	166 A6 SPC(	198 C6 ASC	
		71 47 G		135 87 READ	167 A7 THEN	199 C7 CHR$	
	40 28 (	72 48 H		136 88 LET	168 A8 NOT	200 C8 LEFT$	
	41 29)	73 49 I		137 89 GOTO	169 A9 STEP	201 C9 RIGHT$	
		74 4A J		138 8A RUN	170 AA +	202 CA MID$	
		75 4B K		139 8B IF	171 AB –	203 CB GO	
		76 4C L		140 8C RESTORE	172 AC*	204 CC SYNTAX	
		77 4D M		141 8D GOSUB	173 AD /	ERROR	
	46 2E .	78 4E N		142 8E RETURN	174 AE↑		
		79 4F O		143 8F REM	175 AF AND		
	48 30 0	80 50 P		144 90 STOP	176 B0 OR		
	49 31 1	81 51 Q		145 91 ON	177 B1 >		
	50 32 2	82 52 R		146 92 WAIT	178 B2 =		
	51 33 3	83 53 S		147 93 LOAD	179 B3 <		
	52 34 4	84 54 T		148 94 SAVE	180 B4 SGN		
	53 35 5	85 55 U		149 95 VERIFY	181 B5 INT		
	54 36 6	86 56 V		150 96 DEF	182 B6 ABS		
	55 37 7	87 57 W		151 97 POKE	183 B7 USR		
	56 38 8	88 58 X		152 98 PRINT#	184 B8 FRE		
	57 39 9	89 59 Y		153 99 PRINT	185 B9 POS		
	58 3A :	90 5A Z		154 9A CONT	186 BA SQR		
	59 3B ;			155 9B LIST	187 BB RND		
				156 9C CLR	188 BC LOG		
				157 9D CMD	189 BD EXP		
				158 9E SYS	190 BE COS		
				159 9F OPEN	191 BF SIN		255 FF π

Note: This table shows all valid bytes as they are held within BASIC. Bytes not listed will LIST as apparently meaningful BASIC, but will not run. Within quotes, the full range of VIC ASCII characters can be obtained; see Appendix I for a table.

Use this when PEEKing BASIC or modifying BASIC with an ML monitor.

BASIC as stored in RAM looks strange, partly because of this compression and partly because the apparatus of pointers and line numbers becomes visible. The LIST instruction has the function of presenting this stored collection of bytes in the familiar form, by expanding each token into its correct keyword. Of course, compressing BASIC in this way is a very valuable space-saving feature. This special coded form means that the storage system is different from VIC's ASCII and also different from the screen display system, which may cause some confusion.

It is easy to show how this table could be compiled. Type NEW and enter the following line:

```
0 X
```

An unexpanded VIC-20 stores X at location 4097 + 4 = 4101. Thus, POKEing 4101 with some value, then LISTing, reveals how BASIC treats the selected value. POKE 4101,128 lists as 0 END, for example, as the table indicates. If you wish to investigate this methodically, enter the following line:

```
0 ***********************************
```

or something similar and POKE values from within a loop. The results may be unexpected; if you POKE a value of 5, this will switch the color of the character to white, giving a simple type of UNLIST.

If non-BASIC bytes are POKEd in, LIST will often list as something apparently sensible, but the line will crash with a ?SYNTAX ERROR message. However, literals within quotes (or after REM or DATA statements) can generally take any value except a null byte, which will be treated as an end of line. This is why REM lines are a favorite place for simple anti-LIST methods, as you'll see.

Note that numerals are stored without any attempt at compression, so the 10000 in GOTO 10000 takes five bytes and 123.456 in PRINT 123.456 takes seven; all the components are stored in ways which exclude ambiguity, and there is no way that numbers could be compressed without making them resemble tokens or other BASIC features. Note also that +, −, *, /, and ↑ don't appear in the ASCII list; they are not stored in this form, but as tokens. Finally, note that BASIC punctuation includes the comma, the colon, and the semicolon, but not the period, which is treated as the decimal point.

The token for GO allows BASIC to accept GO TO as well as GOTO. It was tacked on the end when Commodore replaced an earlier BASIC which ignored spaces.

How BASIC Stores Its Variables

Suppose you type A=123 and press RETURN. PRINT A will then print 123. How is this information stored? You've seen that simple variables are allocated space after the program and before arrays, but even if there is no program, they are stored after three zero bytes at the start of BASIC RAM in exactly the same way. Because variables are stored when they're first used, their sequence in memory is the order in which they were encountered by VIC.

Four types of variables can be stored in this area: numeric or floating-point variables (X), integer variables (X%), strings (X$), and function definitions like DEF FN X(Y). The name is always stored in two bytes (the second may be a space character). The four types are distinguished by the high bit of each letter being set or unset. This obviously gives four permutations on each name. It is also the reason that only the first two characters of a name are significant; NUMERAL and NUMBER are both treated as NU. The name carries an implicit type-of-variable declaration, which is interconverted with the BASIC declarators %, $, and FN, and () for arrays.

Figure 6-2. Storage of VIC Simple Variables

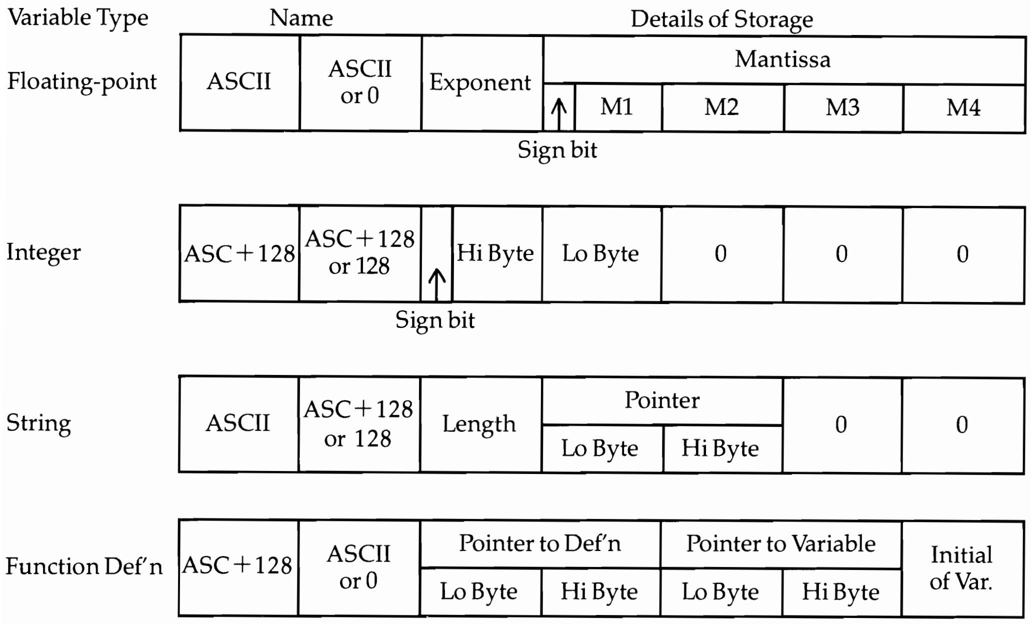


Figure 6-2 shows how simple variables are stored in the VIC. Each variable type takes seven bytes, roughly speaking. This means that when BASIC looks for a simple variable it always adds a constant offset of seven as it searches the table, minimizing search time. However, with this scheme three bytes are wasted with integer variables, two with strings, and one with function definitions.

It may be helpful to look at the different ways in which these seven bytes are interpreted.

Floating-point or real variables (X). The value is held in five bytes to an accuracy of one part in 2↑31, or roughly one in two billion.

These numbers are liable to rounding errors. A subsequent discussion explains floating-point storage in more detail, for the benefit of VIC owners interested in insuring accuracy in financial or other calculations.

Integer variables (X%). Integer variables are held in a signed, two-byte form, within the range −32768 to 32767. The following formula, which allows for the sign bit, gives the value:

(HI AND 127)*256 + LO + (HI>127)*32768

For example, HI=0 and LO=100 correspond to 100; HI=255 and LO=156 are -100. The two expressions add to 0 with overflow.

Strings (X$). Strings cannot be fitted into seven bytes. To allow freedom in assigning strings (without needing to specify their lengths, as some languages require), they are stored in a dynamic way in RAM. Three of the seven bytes allotted to a string variable are relevant; two are wasted. One byte holds the length of the string; LEN (X$) simply PEEKs this value. Another pair of bytes points to the start of the string. Between them, these provide a complete definition. This storage system explains why any string has a maximum of 255 characters. It also explains why CHR$(0) gives no syntax error in cases where "" (a null string) does; the former character has length 1, but the latter has 0 length.

Most strings are stored in the area of free RAM after the BASIC program and variable storage. As they are redefined (or as new string variables are defined), they fill progressively lower memory areas until a so-called garbage collection is forced. To save space, some strings aren't stored after BASIC but reside within the program itself; X$'s pointer in the assignment 10 X$="HELLO" points back within BASIC, where HELLO is already stored.

Generally, any string involving calculation is stored after BASIC. For example, 10 X$="HELLO" + "" assigns exactly the same string to X$, but stores it after BASIC. Again, this has consequences which will be examined shortly. For the moment, note that

```
10 DIM X$ (200): FOR J=0 TO 200: X$ (J) = "1234567890": NEXT
```

uses much less memory—2000 bytes less—than the same program with X$(J)="12345"+"67890" because of this internal storage feature. Any string with any element of calculation (for example, one defined via INPUT or GET or even by A$=B$) is stored after BASIC.

Function definitions. These appear in the variables table, too, and it's quicker to store them here than search the whole program for a definition.

Like strings, function definitions store the solid information elsewhere. A function definition has two pointers, one (within the program) to the defining formula and one to its principal variable, which is set up in the table if it doesn't yet exist. Running 0 DEF FN Y(X) = X↑2+5*X+3 makes two entries in the variables' table (X and of the function definition Y). Actually the formula pointer holds the address of the = sign in the function definition, and the variable pointer marks the address of the first byte of the floating-point value of its argument. The five bytes are also used as temporary storage for calculations when running.

Storage of Arrays

Arrays (subscripted variables) are stored after the simple variables; since they can be of any length, they would spoil the consistency with which seven can be added to each simple variable's pointer.

All three array types—real, integer, and string (there is no array equivalent to the function definition)—have a similar layout, except for the data, which is stored in five-byte batches for real numbers, two-byte batches for integers, and three-byte pointers plus characters for strings. Figure 6-3 summarizes how VIC arrays are stored.

Figure 6-3. Storage of VIC Arrays

Subscripted Variables (Arrays)

Array Name	Offset		No. of DIMs	Last DIM+1		...	First DIM+1		... Data or String Lengths & Pointers ...
	Low	High		High	Low		High	Low	

Arrays are stored in the order they are first used. The array defined last is therefore immediately below free memory. Because of this, it's possible to erase an array which is no longer needed; this can be a useful memory-saving trick if an array is used for some utility purpose (sorting or merging). The general approach is AL=PEEK(49): AH=PEEK(50): DIM X$(100): POKE 49,AL: POKE 50,AH which stores the low and high bytes of the previous top-of-arrays pointer, then restores them after using a new array in some way.

The DIM command defines the size of arrays. Obviously this is necessary unless you have unlimited RAM, since the computer can't know in advance how much storage you'll need. DIM defaults to 10, so it is not necessary with small arrays. Without DIM, X(8)=1 is accepted happily, but X(ll)=1 gives ?BAD SUBSCRIPT ERROR.

Housekeeping with arrays is more complex than that with simple variables, although the stored items are essentially identical. The bit 7 conventions for the name are identical to those for simple variables. The offset figure is the length of the entire array, including numeric data or string pointers; however, it excludes strings, which are stored elsewhere.

The number of dimensions is 1 for a one-dimensional array (A(X)), 2 for an array like A(X,Y), and so on. The actual values of the dimensions have to be stored; each needs two bytes, to allow for arrays like A(500) with more than 256 items. DIM+1 in the diagram is the true number, since dimensions count from the zeroth item (the first item is numbered 0). Finally, there is the data (or, with a string array, the string lengths and pointers). Spare bytes are not wasted; for instance, each integer takes only two bytes.

The data or string lengths and pointers are held in strict sequence, which is ascending order of argument, with the lattermost arguments changing least often. For example, DIM A(1,2) stores its variables in the order A(0,0) A(1,0) A(0,1) A(1,1) A(0,2) A(1,2).

The position of any one item of an array can be calculated. For example, X(*a*,*b*,*c*) is at [*a*+*b**(1+DIM1)+*c**(1+DIM1)*(1+DIM2)] elements along the array, where DIM1 is the number of elements for which *a* was DIMensioned and DIM2 is the DIMension of *b*.

Figure 6-4. A Typical BASIC Program During Its Run

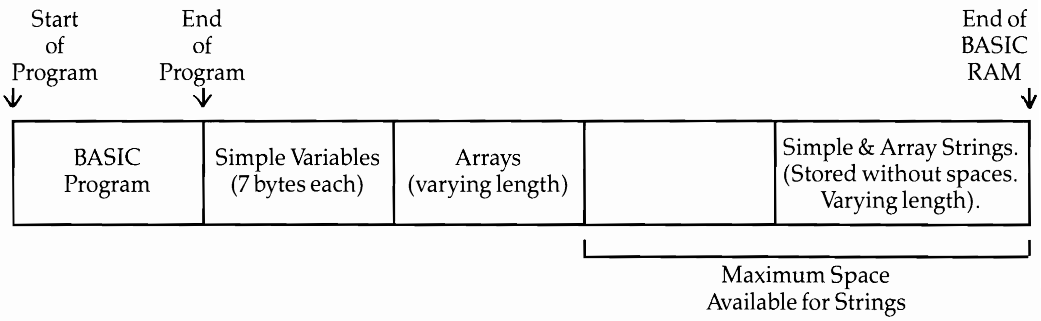

Start of Program
End of Program
End of BASIC RAM
BASIC Program
Simple Variables (7 bytes each)
Arrays (varying length)
Simple & Array Strings. (Stored without spaces. Varying length).
Maximum Space Available for Strings

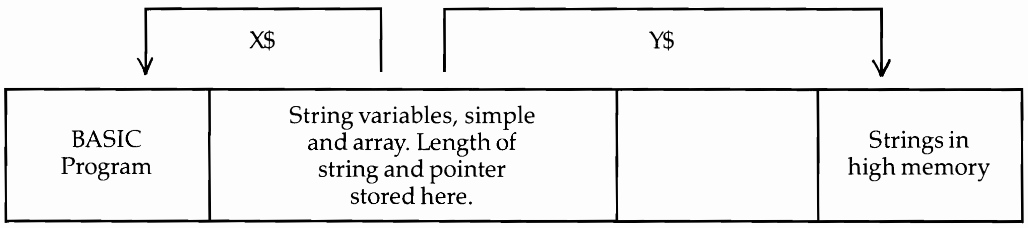
Figure 6-4 shows a typical BASIC program during a run, followed by its variables and arrays.

Some Consequences of VIC's Methods of Storing BASIC and Its Variables

A number of consequences follow from these methods of storage. Strings and arrays are of particular significance with serious programs, so it's worthwhile to explain them thoroughly. If you understand them, you'll be able to write better programs.

String storage. All strings are stored as ASCII characters in ascending order in memory. For instance, Program 6-3 shows how a pair of pointers (S and E for start and end) delimit any current string in memory, and how a string is held in conventional sequence in ASCII. Add lines 11, 12, 13, and 14, like line 10; the program as it stands prints the last of them. But if E is altered to PEEK(55)+256*PEEK(56), which is the top of memory available to BASIC, you'll see how each string is stored and the top-down way each is positioned. Figure 6-5 illustrates this. Note that if a string is redefined, the old string is still left behind in memory; redundant data like this is called garbage.

Figure 6-5. String Storage


X$
Y$
BASIC Program
String variables, simple and array. Length of string and pointer stored here.
Strings in high memory

X$ is stored only in a BASIC program line (for example, 10 X$="HELLO"). Strings which must be built from other strings are stored in high memory (for example, 20 Y$=Y$+Z$).

Program 6-3. Using Pointers to Delimit Strings

```
10 X$="HELLO"+""
20 S=PEEK(51)+256*PEEK(52):E=PEEK(53)+256*PEEK(54)
30 FOR J=S TO E-1:PRINT CHR$(PEEK(J));:NEXT
```

Some loops are heavy users of memory space. For example, FOR J=1 TO 40: X$=X$+"": NEXT could be used to generate a string of 40 spaces. But the first time the loop executes, X$ is defined as a string of one space character; the next time, as two space characters, and so on; so the total memory used is 1 + 2 + 3 +....+ 40 bytes, or 820 bytes, of which 780 are garbage.

Loops to input data in a controlled way, using GET, do something very similar. These rely fundamentally on routines like this one:

```
10 GET X$: IF X$="" GOTO 10
20 Y$=Y$+X$
30 GOTO 10
```

and use a lot of memory. A short word like SIMON processed like this leaves SIMONSIMOSIMSIS in memory.

How String Storage Can Corrupt User-Defined Graphics and ML in RAM

Provided the end-of-BASIC pointer is correctly set, strings will not disturb redefined characters or machine language stored at the top of memory. Programs 6-4 and 6-5 are small-scale, controlled examples (for the unexpanded VIC) of what happens when BASIC strings encroach into the wrong area.

Program 6-4. Unwanted Screen Characters

```
10 POKE 55,22:POKE56,30:REM END OF BASIC INSIDE SC
   REEN
20 POKE 36879,8:REM BLACK, MAKES POKED CHARACTERS
   {SPACE}VISIBLE
30 INPUT N$:N$=N$+"":REM STRING GOES IN SCREEN RAM

40 GOTO30
```

Program 6-5. Corruption of User-Defined Characters and Graphics

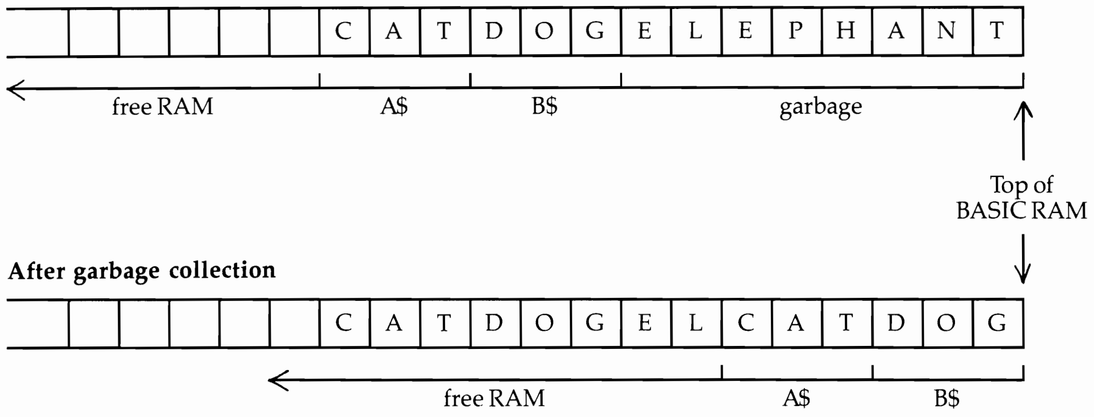
```
10 POKE 36869,255:REM $1C00 IS START OF USER CHARA
   CTERS
20 POKE 55,8:POKE 56,28:REM BUT BASIC ENDS AT $1C0
   8
30 FOR J=7168 TO 7680:POKE J,PEEK(32768+Q):Q=Q+1:N
   EXT:REM MOVE VIC CHARACTER SET
40 PRINT"{HOME}{WHT}@ABC":REM PRINT A FEW CHARACTE
   RS
50 INPUT N$: N$=N$+"":GOTO50:REM @ SYMBOL IS CORRU
   PTED
```

Garbage Collection

Programs using many strings are subject to so-called garbage collection delays. It is fairly easy to see why. Figure 6-6 shows a simplified situation where RAM contains several strings, some of which are now redundant.

Figure 6-6. Garbage Collection

Before garbage collection: A$ *was* ELEPHANT, B$ is DOG, A$ is now CAT.



Let's suppose BASIC tries to set up a new string but finds there's insufficient room. It calls a routine, the same as FRE(0), to find out which of the strings after BASIC are still being used. Strings stored within BASIC itself are outside the scope of this algorithm, and are ignored.

The routine has to check every string variable to determine which is nearest the top end of memory. This string is moved up as far as possible. The process is repeated with the remaining strings until the whole collection is cleaned up. The number of strings is important, not their lengths; generally, with N strings this takes time proportional to N plus N−1 plus N−2, etc. Mathematically inclined people will realize this adds up to an expression including N multiplied by itself (N squared). What this means is that, like the well-known bubble sort, a process that is acceptably fast with a small number of items can become painfully slow with a larger number. In fact, the whole process is analogous to the bubble sort: Intermediate results are thrown away, which saves space but wastes time.

The VIC takes roughly .00075 times the number of strings squared to free memory. The actual relationship is a surprisingly precise quadratic; this is only an approximation. For instance, 100 strings take .8 seconds, 200 take 3 seconds, 300 take 6.6 seconds, and so on.

Note that, during garbage collection, the keyboard locks in an apparent hangup. This is normal; if a long ML routine runs, the STOP key hasn't a chance to work. STOP-RESTORE will interrupt if necessary. In practice, you'll be likely to encounter garbage collection only if you're using memory expansion and string arrays; even 100 ordinary strings will only cause an occasional delay of less than a second. For example, Program 6-6 measures the time required to calculate free memory, which is roughly the same as the time required to perform the garbage collection. Chapter 12 has a demonstration graphics program with a garbage-collection time of 5 seconds, where so many strings are generated that the unexpanded VIC takes 36 seconds to run the program, and where memory expansion cuts this down to 5 seconds.

Program 6-6. Time Required to Calculate Free Memory

```
10 INPUT D:DIMX$(D):FOR J=1TOD:X$(J)=STR$(RND(1)):
   NEXT
20 T=TI:J=FRE(0):PRINT(TI-T)/60"SECONDS"
```

If garbage collection is a problem, you must rewrite the program to reduce the number of strings. There is no other easy solution. For example, pairing strings together roughly divides the delay by 4. Note that performing FRE(0) whenever there's time available (that is, shifting the delay to an acceptable period) can work. It's also possible to do a limited FRE on part of the strings, altering 55 and 56 down or 53 and 54 up.

Calculating Storage Space

Calculating storage space can be important with the VIC. Economizing on memory may have advantages, such as allowing a program to work without expansion. Simple variables and function definitions all take seven bytes, plus allowance for strings, so reusing variables saves memory.

The RAM occupied by an array is easy to calculate. The figure is identical to its own offset pointer, plus strings where applicable. The number of bytes is 5+2*NUMBER OF DIMENSIONS+(DIM1+1)*(DIM2+1) *...* 2, 3, or 5, where the value 2, 3, or 5 depends on the type of array (integer=2, string=3, real=5). In addition, the strings of a string array must be counted.

Integer arrays are very economical. If you have large quantities of numeric data, it often pays to convert it into this form, provided the range −32768 to 32767 is sufficient. It may be worthwhile combining two smaller numbers to increase the efficiency.

Examples. (1) X%(500) has one dimension, and DIM1=500. Therefore, it occupies 5 + 2 + 501*2 = 1009 bytes.

(2) AB(10,10) has two dimensions; DIM1=10 and DIM2=10. This much data will occupy 5 + 4 + 121*5 = 614 bytes.

(3) X$(100) defined with strings on average ten bytes long occupies about 5 + 2 + 101*3 + 101*10 = 1320 bytes.

Order of Definition of Variables

The order in which variables are defined may have a significant effect on the speed of BASIC. This occurs for two reasons. First, whenever BASIC uses a variable, it has to search the table for it. If much-used variables are encountered first, there's some time savings. Second, if BASIC finds a new simple variable and there are arrays in memory, the array table has to be moved up memory.

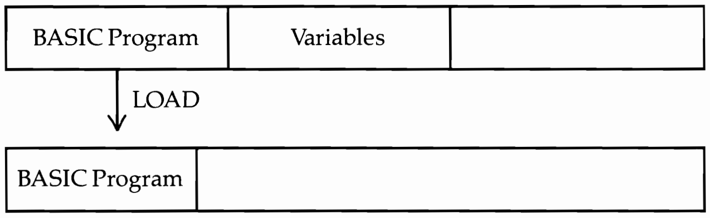
DIM is usually the most efficient way to define variables. It operates on simple variables just as it does on arrays. A statement like DIM J, X, S%, M$, X$(23) has the same effect on BASIC as searching for each variable, not finding it, and therefore positioning it with its default value of zero or the null string after the program.

BASIC Storage—LOAD and SAVE

In the direct mode, SAVE stores a program to tape or disk. It assumes that the pointers at locations 43 and 44, and locations 45 and 46, mark the start and end of the BASIC program, and it saves the bytes between these pointers. As a consequence, it is possible to save a program with its variables by moving the end-of-program pointer up to include variables. This technique works very well with integer arrays, which are an economic way to store numeric data. A similar technique can save character definitions along with BASIC; see Chapter 12 for the method.

In the program mode, LOAD (from within a program) causes what's usually called a chain. The next tape program, or a disk program, is loaded, generally into normal BASIC memory, overlaying the program which performed LOAD. Automatically, an instruction to GOTO the first line is executed, so the new program runs while retaining variables from the earlier program. In this way, BASIC programs too long to fit memory can still be run; for example, a series of programs explaining how to use the computer could be stored on tape and run on an unexpanded VIC. Of course, there's a delay between programs while the next one is loaded.

Figure 6-7. Program Chaining



Note: LOAD command on first program causes new BASIC program to load, then run. In the diagram, the new program is shorter than the old, so variables' values are mostly retained.

This technique, illustrated in Figure 6-7, can be extremely powerful. However, there are two complications. First, the new program may be longer than the second; in this case, the variables will be overwritten. More seriously, the end-of-BASIC pointer still thinks the new program ends where the old program did; so whenever a variable is defined or changed, the program towards the end will be corrupted. For a complete solution to this problem see OLD later this chapter.

A second problem, which is relevant to the storage of variables, is that some strings and all function definitions are stored within BASIC. Thus, a chained program cannot generally make use of function definitions or strings within BASIC. If this is ever a problem (and it could be if strings of user-defined characters are being passed between chained programs), it is easily avoided with strings. Simply use something like A$="ABCDE" + "" in the loader, which forces the string into higher RAM. Function definitions must be redefined in each new program.

Accuracy of Number Storage

All number systems have limitations. Just as the decimal system cannot exactly express thirds, the square root of 2, or pi, so computers with digital storage all have problems with rounding errors. The difficulty is inherent in the machines. Some computer chips designed to perform calculations have registers which indicate when a result has been rounded, and also the lower and upper limits of the result. In practice, great precision is usually unnecessary (or misleading), and for many purposes this subsection will not be needed.

The only reason accuracy is a possible difficulty with the VIC is the fact that numbers as they are printed don't match the precision with which they are stored. If they were printed in full, errors would be obvious.

Try these examples:

```
PRINT 8.13
PRINT 3/5*5: PRINT 3/5*5 =3
```

The first gives 8.13000001. This is the smallest value where an evaluation stores a number in a form which appears changed on PRINT. The second evaluates the result of 3/5*5 and prints it as 3, but the subsequent test shows that it isn't considered equal to 3. In fact, it is stored as 3.0000000009.

Obviously, PRINT is designed to work sensibly in most cases. However, since precision is inevitably lost in some calculations, there must be rounding rules, and exceptional cases are likely to turn up.

Special techniques can avoid these problems. The first is to allow a range of possibilities (for example, treating X and Y as identical if ABS(X−Y)<.0001 or whatever low value is chosen. The classic example of the need for such an approach is the computerized accounting system which, at the end of a long session of inputting of figures, announces that the totals don't balance and that the sum outstanding is zero. Obviously the figure was held as something very small, which should have been rounded to the nearest cent.

Another technique is to use only integers wherever feasible. Yet another is to use special routines; BASIC is generally too slow, but ML routines to multiply digits, for example, aren't all that difficult to write. In fact, BASIC could be extended to include commands like ADD, SUBTRACT, MULTIPLY, and DIVIDE with string arguments, as in COBOL's ADD S$ TO Y$ GIVING Z$. Chapter 4 has some BASIC routines which perform their own rounding, and PRINT USING (see below) is a handy ML routine.

Storage and Errors with Floating-Point Numbers

Except for integer variables, where no loss of precision is possible, floating-point variable values are stored in five bytes (Figure 6-8). Extra bits are used during calculations, but these are lost when the final computed value is stored.

Figure 6-8. Storage of Floating-Point Variables

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5
Exponent	Sign Bit and Mantissa 1	Mantissa 2	Mantissa 3	Mantissa 4

This is a standard arrangement in which every increase in the exponent doubles the value, and where the mantissas are stored in decreasing order of significance. A single bit holds the sign; there is no point in taking up more space, since a sign must have one of only two values. A high bit holds the sign, corresponding with the minus flag of the 6502 chip.

Between them the mantissas hold 31 bits and span a range of 1 to 1.9999... which, when multiplied by the exponent (itself of form 2↑n), takes in the entire range from about 10↑−38 to 10↑38 with an accuracy of one part in 2↑31. Outside these limits, either an overflow error will occur or a very small number will be rounded to zero. There's no underflow error to indicate that a number is too small to be handled.

The following formula will convert any numeral stored in this form into a more comprehensible form:

(−1)↑(M1 AND 128)*2↑(EX−129)*(1+((M1 AND 127)+(M2+(M3+M4/256)/256)/256)/128)

The examples in Table 6-2, PEEKed from VIC memory, will help to clarify this:

Table 6-2. Floating-Point Storage

	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5
-1.5	129	192	0	0	0
0	0	any	any	any	any
.1234	125	124	185	35	163
1.5	129	64	0	0	0
3	130	64	0	0	0
4	131	0	0	0	0
5	131	32	0	0	0
6	131	64	0	0	0
7	131	96	0	0	0
8	132	0	0	0	0
144.75	136	16	192	0	0
99999999	155	62	188	31	224

Note that numbers from 4 to 7.9999... have the same exponent; their bit patterns run from 00000... to 11111... as the value increases. Adding 1 to the exponent doubles the value, subtracting 1 halves it, and so on. Note how negatives have the sign bit set. Note also that an exponent of zero is always taken to indicate a zero value.

To decode a number, the easiest method is to start at the lowest significant byte, divide by 256, add the next, divide by 256, add the next, divide by 256, add M1 (less 128 if necessary), divide by 128, and add 1. Scale up the result (which will be from 1 to 1.999...) by 2↑(EXPONENT−129).

Conversely, if you wish to express a number in this format, either PEEK the values from RAM or (if you can't access a computer) use the method outlined below.

Example. Expressing −13.2681. The minus sign means you must set the high bit of M1. The nearest power of 2 below 13 is 8 (2↑3), so the exponent is 129+3=132. 13.2681 is 8 * (1 + .6585125).

The value following 1 is the number stored by 31 bits in the mantissa:

```
    .6585125 * 128 = 84.2896,
    .2896    * 256 = 74.1376,
    .1376    * 256 = 35.2256,
and .2256    * 256 = 57.75.
```

Thus, the nearest floating-point approximation of −13.2681 is 132/ 212/ 74/ 35/ 58.

Storage Errors

Typically, a number giving aberrant results is stored with the final bit(s) incorrect. For instance, X=3/5*3 stores X as 130/ 64/ 0/ 0/ 1, and the final bit makes X unequal to 3.

Integers and Fractions

Any whole number between 1−2↑32 and 2↑32−1 is held exactly by the VIC, without any error. This is why loops like FOR J=1 TO 100000: PRINT J: NEXT can continue without error, while the same loop with STEP .9 soon prints numbers with rounding errors.

Note that 2↑32−1 is stored as 160/ 127/ 255/ 255/ 255/ 255 and is the highest accurately stored integer; 2↑32 is stored as 161/ 0/ 0/ 0/ 0. Similar rules apply to fractions; provided they are combinations of 1/2, 1/4, 1/8, 1/16,..., 1/2↑31, they can be held exactly. Because of this it may be best (particularly in financial calculations) to store values as integers.

Special Locations and Features of VIC BASIC

BASIC uses a lot of the low end of memory for temporary storage, and many of these storage locations are programmable from BASIC. This section describes, alphabetically, some of the more useful storage methods. Most have been used, with examples, earlier.

The keyboard and some aspects of screen handling are also included here, as both have special points of interest in BASIC programming.

Buffers

The input buffer, keyboard buffer, and tape buffer occupy locations 512–600 ($0200–$0258), 631–640 ($0277–$0280), and 828–1019 ($033C–$03FB), respectively. During normal operations, each of these areas has one exclusive function. Program 5-1 from the preceding chapter allows you to watch the first two of these in action.

Input buffer. Program 6-7 demonstrates the use of the input buffer.

Program 6-7. Using the Input Buffer

```
10 N$="X=X+1:PRINTX:GOTO10"+CHR$(0)
20 FOR J=1 TO LEN(N$)
30 POKE 511+J,ASC(MID$(N$,J)):NEXT
40 POKE781,255:POKE782,1
50 SYS50310
```

An ASCII string, terminated by a null byte and POKEd into the buffer, behaves exactly as the same line would if typed in from the keyboard. Lines 40 and 50 execute the buffer, after first setting a pointer to $01FF (the start of the buffer) less 1. The buffer would also be executed if the end of the program was reached, or if an END statement was encountered, but using the SYS shown in Program 6-7 is often more useful.

Keyboard buffer. It's easy to show that VIC has a queuing system for keystrokes. A short routine (1 GET X$: PRINT X$: FOR J=1 TO 2000: NEXT: GOTO 1) prints characters which must have been typed before GET was reached. Up to ten characters can be stored here. POKE 649 to change this, but if the value exceeds 10, you'll corrupt some important pointers.

Location 198 ($C6) independently stores the number of characters in the buffer. POKE 198,0 therefore removes all characters; it has the same effect as FOR J=1 TO 10: GET X$: NEXT. SHIFT-RUN puts LOAD:RUN [RETURN] into this buffer, using a routine at $E609.

Many examples in this chapter (AUTO, DELETE, LIST) rely on this buffer. Program 6-8 shows how POKEs into the buffer work; try it to get the feel of the technique.

Program 6-8. Using the Keyboard Buffer

```
10 DATA 72,69,76,76,79
20 FOR J=631 TO 635:READX:POKEJ,X:NEXT
30 POKE198,5
```

POKEing one or more return characters (13) into the queue is also a popular trick, since it allows messages printed on the screen to be input later. In effect, that extends the range of the command beyond ten characters.

The next example, Program 6-9, puts a quote in the line just before INPUT. This is very useful when a string which is to be input may contain commas, colons, or other separators. A quote inputs the entire string without error. Run this program, typing in something like A, B, C, and contrast the result with that achieved by an unadorned INPUT statement.

Program 6-9. Using a Quote Before INPUT

```
1000 POKE 198,1:POKE 631,34
1010 INPUT X$
1020 PRINT X$:GOTO 1000
```

Program 6-9 actually eliminates any stored characters, an effect that can be annoying. A much more sophisticated (but longer) version of that program is given in Program 6-10.

Program 6-10. Moving Characters Along the Buffer

```
1000 P=PEEK(198):P=P+1:IF P>9 THEN P=9
1010 FOR J=631+P TO 631 STEP -1:POKE J,PEEK(J-1):N
     EXT
1020 POKE 198,P:POKE 631,34
1030 INPUT X$
1040 PRINT X$:GOTO1000
```

This program moves the characters along the buffer, just as VIC's operating system does.

A more exotic use is to transfer BASIC programs to the VIC from another computer by inputting them in ASCII via a modem, printing individual lines on the screen, and inputting each line, adding a RETURN at the end. It is quicker than typing them in, although the work of conversion is likely to be a problem.

Tape buffer. The VIC's operating system reserves this area for tape use, and it is therefore a popular place to put ML routines once no more tape activity is expected (after a program has been loaded and is running). It is not actively programmable like the two previous buffers. In addition, it is overwritten whenever tape is written to or read from, so don't put ML here if you're using tape to load or save data, or if you are chaining programs.

Spare RAM areas. These aren't buffers in the usual sense. The VIC has 1K of RAM at the low end of memory for its own use, but some isn't allocated and can be taken over by programmers. Locations 251–254 ($FB–$FE), 673–767 ($02A1–$02FF), 784–787 ($0310–$0313), 820–827 ($0334–$033B), and 1020–1023 ($03FC–$03FF) are available. The second of these areas is 95 bytes long; the tape buffer has 192 bytes.

Clock

The three-byte jiffy clock is stored at locations 160–162. Location 162 is the fastest-changing byte. At each interrupt, about every 1/60 second (a unit of time called a *jiffy*, hence the name jiffy clock), that location is incrememted, with overflow when necessary into the higher bytes. Thus, location 161 is incremented every 256/60 seconds, or about every 4.2667 seconds; location 160 is incremented every 65536/60 seconds, or about every 18.204 minutes. The PAUSE routine later in this chapter shows a possible use of the jiffy clock.

The TI and TI$ reserved variables discussed in Chapter 3 are derived from these bytes by a straightforward conversion. TI equals PEEK(162) + 256*PEEK(161) + 256*256*PEEK(160); for TI$, the value of TI (in jiffies) is converted into hours, minutes, and seconds. Although the speed of the clock is constant, it is not identical to that of real clocks, since the interrupts aren't at precise 1/60 second intervals. The error varies internationally, and between VICs and 64s, but the maximum error will not be more than one part in 33000 (a couple of minutes a day).

Disabling STOP and STOP–RESTORE

Turning off, or disabling, these keys is a useful way to provide some program security and to guard against accidental use of the STOP key or of SHIFT-STOP (which will try to load a program).

A completely reliable way to disable this key, while keeping it usable for emergencies, is to put a metal guard over the key itself.

Four approaches using software are given below. Remember, however, that if your computer goes into an infinite loop with STOP disabled, you may have to turn your VIC off to correct the problem. Be sure to include a subroutine to reenable STOP.

To disable both STOP and STOP-RESTORE (method 1) POKE 808,109: POKE 809,220: POKE 808,54. To reenable, POKE: 808,109: POKE 809,247: POKE 808,112. This is one of the best methods, since it leaves the clock working, disables both STOP and STOP-RESTORE, doesn't affect tape operations, and leaves LIST working normally.

To disable both STOP and STOP-RESTORE (method 2) POKE 808,109. To reenable, POKE 808,112. This simple POKE disables both STOP and STOP-RESTORE. It leaves the clock working, but it may have an effect on tape loading. If you're not using tape once the program is loaded, this method is fine. Note, however, that LIST will be scrambled. Every time LIST checks for the STOP key, the pointer telling it the length of the current line is changed. It is strange to see a listing composed of apparent garbage run properly.

To disable STOP, POKE 788,194. To reenable, POKE 788,191. This simple POKE modifies the interrupt vector so that it bypasses the Kernal routine to increment the clock and check STOP. However, it doesn't disable RESTORE. Tape operations defeat this procedure; during reading from tape the interrupt sequence is reset, and STOP breaks into the program. The jiffy clock is turned off.

To disable STOP-RESTORE but not STOP, POKE 37150, PEEK(37150) AND 127. To reenable, POKE 37150, PEEK(37150) OR 128. This turns off the VIA interrupt. The usual values are POKE 37150,2 (off) and POKE 37150,130 (on). POKE 792,91: POKE 793,255 has the same effect; it works by altering the NMI vector so that it immediately exits without carrying out any of its normal processing.

Notes. The STOP key is not an interrupt-like device, as it may appear to be. In fact, every 1/60 second the Kernal routine which tests the STOP key is called. That routine looks at location $91 (145), and if the smallest bit is low (that is, if the contents equal $FE (254)), STOP is pressed. ML programmers can therefore use JSR $FFE1/BEQ to STOP a routine.

LIST and RUN also use the Kernal routine which tests the STOP key, which is why a listing or a running program can be stopped. Tape LOAD and SAVE also use it; so does RESTORE.

The Kernal routine at location 65505 ($FFE1) jumps to the address in locations 808 and 809. Method 1 changes this from F770 to DC36, via an intermediate meaningful address. The ML it finds there is simply LDA #$1/RTS. This nonzero value insures that STOP will never occur. There are several versions of method 2, POKEing around 130, but these call a routine to close files and can give odd effects with disks.

Function Keys

The simplest way to program function keys f1–f8 is to use a simple GET. The range of ASCII values is 133 to 140, in sequence 133, 137, 134, 138, 135, 139, 136, and 140. For example, to test whether f7 is pressed, you could use:

```
500 GET A$:IF A$="" THEN 500
510 IF A$=CHR$(136) THEN PRINT"F7 PRESSED"
```

More sophisticated function key programming requires machine language. Program 6-11 enables all eight function keys (unSHIFTed and SHIFTed) to be defined with individual strings up to 32 characters long.

Program 6-11. Function Key Handler

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
Ø DATA 32,253,2Ø6,32,158,2Ø5,32,141,2Ø5,32,247,215
  ,136,152,1Ø,1Ø,1Ø                          :rem 188
1 DATA 1Ø,1Ø,133,253,169,29,133,254,32,253,2Ø6,32,
  158,2Ø5,32,143,2Ø5                           :rem Ø
2 DATA 16Ø,Ø,177,1ØØ,24Ø,22,17Ø,2ØØ,177,1ØØ,133,25
  1,2ØØ,177,1ØØ,133                          :rem 179
3 DATA 252,16Ø,Ø,177,251,145,253,2ØØ,2Ø2,2Ø8,248,1
  38,145,253,169,28                          :rem 228
4 DATA 141,144,2,169,75,141,143,2,96,24,165,215,23
  3,132,2Ø1,8,144,3                          :rem 213
5 DATA 76,22Ø,235,17Ø,189,124,28,133,253,169,29,13
  3,254,16Ø,Ø,177,253                         :rem 83
6 DATA 24Ø,237,2Ø1,95,24Ø,6,32,21Ø,255,2ØØ,2Ø8,242
  ,166,198,169,13,157                         :rem 71
7 DATA 119,2,23Ø,198,2Ø8,216,Ø,64,128,192,32,96,16
  Ø,224                                      :rem 152
1ØØ POKE 56, PEEK(56)-2: CLR{8 SPACES}:REM LOWER M
    EMORY BY 256 BYTES                       :rem 241
11Ø S=PEEK(56): S1=256*S                      :rem 24
12Ø FOR J=S1 TO S1+131: READ X: POKE J,X: NEXT
                                             :rem 2Ø2
13Ø POKE S1+22,S+1: POKE S1+65,S: POKE S1+9Ø,S: PO
    KE S1+94,S+1                              :rem 19
14Ø FOR J=S1+256 TO S1+511: POKE J,Ø: NEXT: REM SE
    T ALL FN KEYS NULL                       :rem 174
15Ø PRINT "USE SYNTAX:-                      :rem 128
16Ø PRINT "SYS" S1 ",N,STRING:CLR"           :rem 199
```

The program reserves 512 bytes at the top of memory for the ML and the eight 32-character strings. Strictly speaking, only 31 characters are available for each key definition because each has a null byte as a terminator.

When Program 6-11 is run, all function definitions are initially blank. To define a function key, use a statement of the form

SYS ***address, n, "string"*****:CLR**

Where *address* is the start of the ML routine (the program will tell you the proper value to use), *n* is the number of the function key you wish to define, and *string* is the string of up to 31 characters you wish to assign to that key. CLR sets the pointers properly. For example, if you have an unexpanded VIC and wish to have HELLO printed whenever you push the f5 key, you would use SYS 7168,5,"HELLO":CLR.

The vector at locations $028F/$0290 is central to the method; see the section in this chapter on the keyboard for other illustrations. It is changed to point at our ML routine, where location $D7 is tested, and, if its contents are in the function key range, the original string is reconstructed and printed out.

The program does not check the length of your definition strings. You can enter a string longer than 31 characters, but if you do, the string will overwrite part of the next function key's definition.

The software is written so that the left-arrow key (not to be confused with the cursor-left key) can be used to insert a RETURN into the keyboard buffer. Thus, SYS 7168,1,"LIST 100-300←":CLR defines function key 1 to type LIST 100-300 and press RETURN, so those lines will be listed. POKEs into hard-to-memorize locations, loops, SYS calls to routines in memory, and printouts of current variable values are typical applications of user-defined keys. The *Super Expander* and *Programmer's Aid* cartridges have a similar built-in routine.

Keyboard

Commodore keyboards are very reliable. The VIC keyboard has 66 keys, which are light-colored plastic stems surmounted by colored plastic caps with white legends. The tops of the keys can be levered off and replaced. It is possible to rearrange the keys, perhaps into German QZERTY style or the Dvorak layout with ergonomically arranged letters, or for that matter alphabetically; as you'll see, the software which decodes the keyboard can allow for this. Single-key entry of BASIC keywords is possible, too, and the function keys can be programmed to output strings.

The keyboard can handle most ASCII characters, with the exception of control characters and less common punctuation symbols (curly brackets, backsloping single quotes, and the like). Its software is not quite free of bugs; for instance, the 2 key plus a joystick held to the right simulates a STOP key and can crash BASIC.

Keystrokes are queued in a buffer (see the discussion of the keyboard buffer above for details). Holding down the CTRL key delays screen scroll and is designed to act with the color keys or RVS ON and RVS OFF. CTRL has no other effect; CTRL-C doesn't generate CHR$(3), for instance.

RUN/STOP, when used with the left SHIFT key, puts LOAD:RUN into the keyboard buffer, giving single-stroke loading and running of tape programs. The right SHIFT key fails to do this. You will soon see how the SHIFT keys can be distinguished in software.

How the keyboard is read. When the VIC is operating normally, its interrupt routine performs several functions. The clock is updated and the status of the STOP key is saved, the cursor may be flashed, the cassette motor is turned off unless a flag is set, and the keyboard is scanned and the keyboard buffer updated if a key is pressed.

This can be traced by disassembling the Kernal ROM routines. When an interrupt is generated, the 6502 finishes its current instruction, saves a few values, and jumps to the address held in locations $FFFE/FFFF at the very top of memory. In the VIC, those locations point to a ROM routine which checks that an interrupt (rather than a BRK instruction) has been encountered, and then jumps to the address in $0314/0315, which is normally $EABF. The first instruction in the routine at $EABF is JSR $FFEA, which calls another Kernal routine to increment the clock, save STOP status, etc., followed by screen and tape handling and a call to $EB1E. This is the same routine used by the Kernal's SCNKEY routine ($FF9E) which, as the label implies, scans the keyboard.

Only 64 of the 66 keys are detectable by the keyscan routine. RESTORE is one odd key out; it causes a nonmaskable interrupt (NMI) when pressed and isn't decoded with the other keys. SHIFT LOCK is the other missing key.

To read the keyboard, two ports of a VIA, at $9120 (rows) and $9121 (columns), are examined for bits set to zero. These bits are almost all 1; only the grounding action of a key being pressed sets a zero value. That is why Table 6-3 has values of 127, 191, and so on, because the bit patterns are 01111111, 10111111, and so on. In turn, the row register is rotated to take one of eight values, and each bit of the column is tested each time. A counter increments with each loop, and this is the value which is stored when a keypress is found. Since there are eight rows and columns, 64 different values are theoretically possible. VIC uses all of them.

Table 6-3. Rows and Columns of Keyboard VIA

Contents of $9121 (column)

Contents of $9120 (row)

	7F (127)	BF (191)	DF (223)	EF (239)	F7 (247)	FB (251)	FD (253)	FE (254)
7F (127)	F7	CLR	–	0	8	6	4	2
BF (191)	F5	↑	@	O	U	T	E	Q
DF (223)	F3	=	:	K	H	F	S	Logo
EF (239)	F1	Right/SHIFT	.	M	B	C	Z	Space
F7 (247)	CRSR Down	/	,	N	V	X	Left/SHIFT	STOP
FB (251)	CRSR Right	;	L	J	G	D	A	CTRL
FD (253)	RETURN	*	P	I	Y	R	W	←
FE (254)	INST	£	+	9	7	5	3	1

Normal row value.

Note: The row is always set to 247 except during the actual reading, so STOP can be detected merely by testing whether $9121 holds $FE. Left-SHIFT, X, and several other keys can be checked in this manner too.

Machine language is necessary to read the keyboard. The following short BASIC program:

```
10 POKE 808,109        : REM DISABLE STOP
20 INPUT "ROW"; R   : REM USE 127,191, ETC
30 POKE 829, R
40 SYS 828: GOTO 40
```

along with this machine language routine:

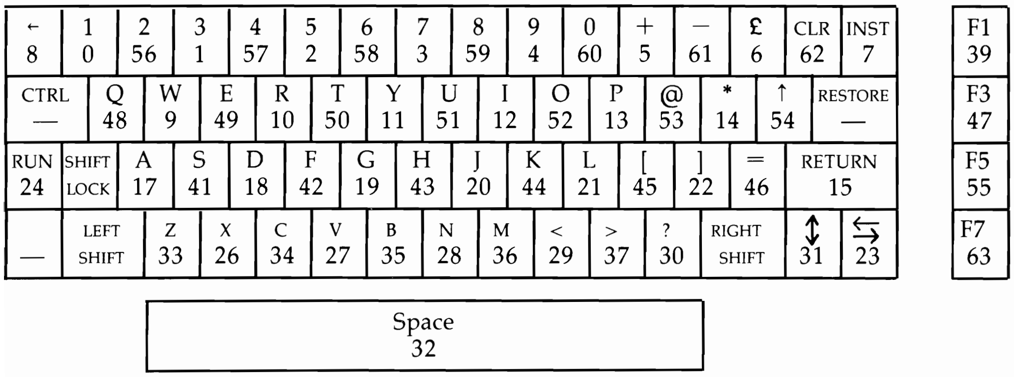
```
$033C LDA  #$00
$033E STA  $9120   ; Set row
$0341 LDX  $9121   ; Read column
$0344 LDA  #$00
$0346 JSR  $DDCD   ; Print X in decimal
$0349 LDA  #$0D
$034B JMP  $FFD2   ; new line
```

illustrates the way that rows and columns interact. At this level SHIFTed and unSHIFTed keys aren't distinguished.

Note that $9120 is configured for output, and $9121 for input, by setting $9122 to $FF and $9123 to $00. Thus, PEEK(37154) is 255 and PEEK(37155) is 0. If these values are changed, which can happen, the keyboard cannot function correctly. POKE 37154,191 is typical; it allows only every other key in the top rows to operate. POKEing the correct values restores normal functioning. P SHIFT-O 37153+1, 155+99+1 is one way to get around the deactivated 2, 4, and E keys.

How the keyboard is decoded. When a key is pressed, the number of times the keyscan routine has looped (a number from 0 to 63) is stored in $CB (203). The previous value of this location is in $C5 (197), and comparing the two shows whether a new key is being pressed. PEEKing either location (see Figure 6-9) is a very useful way to test for key depressions without bothering with GET, and it has the advantage of retaining the value as long as the key is held down. If the keyscan routine loops 64 times and does not detect a keypress, it leaves a value of 64 in these locations, so PEEK(197)=64 means no key is pressed. In practice, locations $C5 and $CB can be used interchangeably to detect keypresses.

Figure 6-9. Key Values Stored in $C5 and $CB



SHIFT, Commodore, and CTRL keys. Location $028D (653) stores information on these keys, which are assigned bit values 1, 2, and 4, respectively. Try FOR J=1 TO 9E9: PRINT PEEK(653): NEXT which will give a value from 0 to 7 depending on which of these three keys is pressed. The chess game *Sargon* looks for all three of these keys pressed simultaneously (by testing for a value of 7) before centering the screen.

STOP key, left SHIFT key. Location $91 (145) stores a copy of the normal keyboard row and is used to identify the left SHIFT (and distinguish it from the right). It also indicates when STOP is pressed. Try PRINT PEEK(145) in a loop; it prints 253 ($FD) with left SHIFT, and 254 ($FE) with STOP.

Converting the key value to ASCII. Converting these values into ASCII is the final step. The VIC has four character tables built into ROM, for unSHIFTed, SHIFTed, Commodore key, and CTRL character sets, starting at $EC5E, $EC9F, $ECE0, and $EDA3, respectively. They're each 65 bytes long and convert $CB's contents (0–64) into ASCII values, making allowance for the SHIFT, Commodore logo, or CTRL keys. In decimal, these tables start at 60510, 60575, 60640, and 60835.

After storing the key value in $CB, an indirect jump is executed via $028F/0290 to $EBDC. This routine's function is to set the address in $F5/F6 to point to one of the four keyboard tables, depending on whether a SHIFT key or Commodore is pressed, so that the correct ASCII value can be determined. In addition, if the value in location $0291 (657) is less that 128, SHIFT-Commodore will switch graphics sets from lowercase/uppercase to uppercase/graphics, or vice versa.

With $F5/F6 pointing to one of the four keyboard tables, the routine at $EB74 is entered and the key's ASCII value is determined and stored in location $D7. The routine then deals with keyboard repeats as well as with cursor control and other special keys.

Detecting several simultaneous keypresses is possible only with a machine language loop; normal keyboard reading goes no further than the first keypress encountered. This means that a music-style keyboard must be read with ML.

Using the keyboard vector. The contents of $028F/0290 can be changed to point to your own routine, so that keys can be intercepted and their effect changed. This is a machine language technique; see the earlier example showing how to program the function keys to print out strings of characters.

Method 1: Intercepting one or more keys. Generally, you intercept keys to trigger an activity, like printing a message. The technique is to test either $CB or $D7 for your key or keys, $CB if you're only concerned with the physical key, $D7 if you need to distinguish unSHIFTed and SHIFTed keys. Additionally, left SHIFT can be detected by testing location $91, and SHIFT/CTRL/Commodore can be detected by testing location $028D. Jump to $EBDC if the desired key isn't pressed; end your own routine with JMP $EB74. In this way, keyboard processing is perfectly normal, SHIFT keys and all, except for your own specially inserted routine. This very simple example changes VIC's background color whenever the back-arrow (←) key is pressed if you change the contents of $028F/0290 to point to the start of this routine:

```
        LDA $CB
        CMP #$08   ; back-arrow key
        BEQ LABEL
        JMP $EBDC  ; Continue normal keyboard operation
  LABEL LDA $900F
        CLC
        ADC #$10   ; adding 16 changes background
        STA $900F  ; color to next in sequence
        JMP EAE0
```

As a more complex example, consider how to print BASIC keywords with single keystrokes. You could use left-SHIFT plus a key, or CTRL-Commodore plus a key, or many other combinations. There are roughly 64 keywords, so almost every key can be assigned its unique word. The example program below uses the Commodore key in combination with other keys to print BASIC words; for instance, the Commodore key with R prints RUN. Many people like this type of entry of BASIC, even though VIC's key tops are not labeled with the keywords. One of the points to watch for is the debouncing feature just after LABEL; without this, the words will print repeatedly, instead of only once.

```
        LDA $028D
        CMP #$02    ; Is Commodore key pressed?
        BEQ LABEL
EXIT:   JMP $EBDC   ; Continue as usual if not.
LABEL:  LDY $CB     ; Now look at ordinary scanned keys;
        CPY #$40
        BEQ EXIT    ; exit if no such key pressed.
        CPY $C5
        BEQ EXIT    ; Exit if same key pressed as last time;
        STY $C5     ; if new key, record it in $C5.
        INY         ; Loop to choose Yth BASIC word.
        LDX #$00
LOOP    INX
        LDA $C90C,X ; BASIC words are stored from $C09E.
        BPL LOOP    ; Look for high bits set,
        DEY         ; and, when found, decrement
        BNE LOOP    ; Y until it counts down to zero.
PRINT   INX
        LDA $C09C,X ; Load and print consecutive characters
        BMI LAST    ; end signaled by high bit set.
        JSR $FFD2
        BMI PRINT   ; Make the routine freely relocatable.
        BPL PRINT
LAST    AND #$7F    ; Turn off high bit of last character
        JSR $FFD2   ; then print it.
        JMP $EBDC   ; Continue normal keyscan.
```

The BASIC equivalent is given in Program 6-12. Variable S controls the place in memory into which the routine is POKEd; any free RAM area is acceptable. Note how line 40 changes the value of the keyboard decode table vector in locations $028F/0290 (655/656).

Program 6-12. Single Key Entry of Keywords

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
Ø DATA 173,141,2,2Ø1,2,24Ø,3,76,22Ø,235,164,2Ø3,19
  2,64,24Ø,247,196                              :rem 159
1 DATA 197,24Ø,243,132,197,2ØØ,162,Ø,232,189,156,1
  92,16,25Ø,136                                  :rem 3Ø
2 DATA 2Ø8,247,232,189,156,192,48,7,32,21Ø,255,48,
  245,16,243,41                                  :rem 43
3 DATA 127,32,21Ø,255,76,22Ø,235                 :rem 3Ø
1Ø REM SINGLE KEY BASIC                          :rem 82
2Ø S=828                                        :rem 148
3Ø FOR J=S TO S+54:READ X:POKEJ,X:NEXT           :rem 12
4Ø POKE 656,S/256:POKE 655,S-INT(S/256)*256:REM PU
   T S IN ($Ø28F/Ø29Ø)                          :rem 166
```

Method 2: Redefining the keyboard. If you wish to redefine the keyboard, the best way is to copy all four tables into RAM, modify them, and access them by changing the vector in $028F/0290 to point to a routine like the following:

```
     LDA $028D       ; test SHIFT, CTRL, Commodore key
     ASL             ; Omits test for SHIFT-Commodore in ROM
     CMP #$08        ; which switches lower- and uppercases
     BCC OK
     LDA #$06        ; Rest of keyboards are CTRL
OK   TAX
     LDA TABLE,X
     STA $F5
     LDA TABLE+1,X
     STA $F6
     JMP $EB74
```

where TABLE is six bytes that hold the starting addresses of your unSHIFTed, SHIFTed, and Commodore key ASCII decode tables (in standard low-byte/high-byte format).

The following BASIC program will also do this for you, keeping three tables in the protected top of BASIC memory. It has 31 bytes of ML, then 8 bytes pointing to the tables in RAM, and finally 260 bytes of tables.

Program 6-13. Redefining the VIC's Keyboard

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
9 REM{3 SPACES}** LOWER MEMORY TOP/ MOVE KEYBOARD
  {SPACE}TABLES TO RAM/{6 SPACES}**              :rem 62
1Ø T=PEEK(55) + 256*PEEK(56) :S=T-299          :rem 148
2Ø POKE 55,S AND 255: POKE 56,S/256: CLR        :rem 35
3Ø S=PEEK(55) + 256*PEEK(56)                   :rem 166
4Ø FOR J=6Ø51Ø TO 6Ø7Ø4                        :rem 114
45 POKE S+39+X,PEEK(J): X=X+1: NEXT            :rem 237
5Ø FOR J=6Ø835 TO 6Ø899                        :rem 14Ø
55 POKE S+39+X,PEEK(J): X=X+1: NEXT            :rem 238
```

```
59 REM{2 SPACES}** ML TO HANDLE SHIFT ETC/ ADD ADD
   RESSES OF RAM TABLES **                      :rem 2
60 FOR J=S TO S+30: READ X: POKE J,X: NEXT::rem 67
70 TABLE=S+31: K1=S+39: K2=S+104: K3=S+169: K4=S+2
   34                                         :rem 242
80 POKE TA,K1 AND 255: POKE TA+1, K1/256    :rem 17
90 POKE TA+2,K2 AND 255: POKE TA+3, K2/256:rem 115
100 POKE TA+4,K3 AND 255: POKE TA+5, K3/256
                                              :rem 161
110 POKE TA+6,K4 AND 255: POKE TA+7, K4/256
                                              :rem 168
120 POKE S+19,TA AND 255: POKE S+20,TA/256:rem 176
130 TA=TA+1: POKE S+24,TA AND 255: POKE S+25,TA/25
    6                                         :rem 175
139 REM ** REDIRECT ($028F) TO THE NEW KEYBOARD PR
    OCESSING{5 SPACES}**                      :rem 140
140 POKE 655,S AND 255: POKE 656, S/256:  :rem 225
200 DATA 169,1,208,3,76,220,235,173,141,2,10
                                               :rem 97
210 DATA 201,8,144,2,169,6,170,189,31,160,133
                                              :rem 157
220 DATA 245,189,32,160,133,246,76,116,235 :rem 25
300 PRINT "4 TABLES AT" S+39 "TO" S+298    :rem 218
308 REM ** ACCEPT KEY/ SEARCH RAM FOR IT
    {23 SPACES}**                               :rem 9
309 REM ** SEARCH ALL 4 KEYBOARDS/ POKE NEW VALUE
    {SPACE}INTO RAM{5 SPACES}**               :rem 110
310 PRINT "KEY TO BE REDEFINED?"             :rem 61
320 GET X$: IF X$="" GOTO 320                :rem 133
330 PRINT X$: X=ASC(X$)                       :rem 18
340 FORJ=S+39TOS+298:IFX<>PEEK(J)THENNEXT :rem 124
350 J=J-S-39                                 :rem 130
360 PRINT "NEW VALUE OF KEY?"                :rem 142
370 GET X$: IF X$="" GOTO 370                :rem 143
380 PRINT X$: X=ASC(X$)                       :rem 23
390 POKE S+39+J,X: GOTO 310                  :rem 181
1000 PRINTPEEK(197)PEEK(653)                 :rem 121
1010 :GOTO1000                               :rem 246
```

This program moves the keyboards from ROM (60510–60704) into the protected top of RAM. Its machine language resembles that above. However, there's an extra feature; POKEing location S+1 (just after the start of the protected area) with 0 reverts to the normal keyboard.

Lines 80–130 are all concerned with the table addresses, which must be entered since they can vary. The loop at 310 changes keys. The table definitions can be saved to tape or disk; line 300 prints the addresses.

The keyboard as a device. The keyboard is treated as device number 0 by the operating system. You can open a file to the keyboard and treat it as an input device using the following program.

```
10 OPEN 5,0
20 INPUT#5,X$
30 PRINT X$: GOTO 20: REM LOOK AT WHAT'S BEEN INPUT
```

Commas and other punctuation symbols, which BASIC treats as separators, will no longer give ?EXTRA IGNORED (because a file is considered open), but parts of a string may be lost. The normal question-mark prompt is not printed.

Repeating keys. Location 650 controls which keys, if any, will repeat if held down. POKE 650,0 causes the space bar, INST/DEL, and cursor control keys to repeat. POKE 650,64 prevents any key from repeating, and POKE 650,128 makes all keys repeat.

Repeat rates, as well as the delay before repeat takes place, are not easily controlled from BASIC. The constants are built into ROM routines, so POKEd values quickly revert to normal. An easy way to speed repeat is simply to change the rate at which interrupts occur, but this has the disadvantage of making the jiffy clock count faster.

Location 652, the countdown, can be used too for fine-tuning (for example, where it's necessary to move from one value to another through a large range of values). Program 6-14 shows how this can be done.

Program 6-14. Repeat Rates

```
9000 GET X$: IF PEEK(652)>0 AND PEEK(652)<16 THEN
     {SPACE}J=0
9010 J=J+1
9020 IF J>100 THEN J=100
9030 IF X$="" THEN 30000
9040 RETURN
```

Add line 8000 GOSUB 9000: X=X+J: PRINT X: GOTO 8000 and watch the value of X increase faster the longer a key is held down. The + and − keys can be used with a subroutine like this to step through memory efficiently, allowing small changes by hitting + or − and increasingly rapid changes if either key is held down.

Important keyboard locations. Table 6-4 shows some of the more important locations and ROM routines associated with reading and decoding the keyboard.

Table 6-4. Summary of Keyboard Locations

$91	145	STOP Key/Left SHIFT key record
$C5	197	Previous key pressed
$C6	198	Number of characters in keyboard buffer
$CB	203	Most recent key pressed
$D7	215	ASCII value of key pressed
$F5/F6	245/246	Keyboard table pointer
$0277-$0280	631-640	Keyboard buffer
$0289	649	Maximum number of characters in the keyboard buffer
$028A	650	Repeat key flag (0: space and cursor keys repeat, 64: no keys repeat, 128: all keys repeat)
$028B	651	Repeat delay after repeating begins
$028C	652	Delay before repeating begins
$028D	653	SHIFT, Commodore, CTRL status (1, 2, and 4, respectively)
$028E	654	Previous configuration of SHIFT, Commodore and CTRL
$028F/0290	655/656	Vector enabling user-written keyboard tables
$0291	657	SHIFT-Commodore case switching enable/disable (0 enables, 128 disables)
$EABF	60095	Starting address of normal interrupt handling routine
$FF9F	65439	Kernal Routine to read the keyboard (SCNKEY); jumps to $EB1E

Table 6-5. Summary of Screen Locations and ROM Routines

$C7	199	Reverse flag (0 reverse off, 18 character reverse on)
$C9	201	Cursor row and column for input from screen
$CC	204	Cursor flash (0 flashes cursor)
$CD	205	Cursor countdown before blink
$CE	206	Character under cursor
$CF	207	Cursor blink phase (0 or 1)
$D0	208	Input from screen/keyboard (flag is 3 for screen or 0 for keyboard
$D1	209	RAM Address of start of current line
$D3	211	Position of cursor on line
$D4	212	Quotes flag (0 not in quotes, 1 in quotes)
$D5	213	Current length of screen line (21, 43, 65, or 87)
$D6	214	Cursor's row
$D9-$F0	217-240	Table of screen line links (1E continues; 9E doesn't)
$F3	243	Color RAM address
$0286	646	Color code (0, black, through 15, light yellow) in use
$0288	648	High byte of start of screen (usually 16 or 30)

Interesting ROM routines include:

$E55F	58719	Clear screen
$E5C3	58819	Set VIC chip to normal values
$E64F	58959	Input from screen (or keyboard)
$E912	59666	Converts CHR$ (<u>color</u>) in A into 0-7
$EA8D	60045	Clear entire row (POKE 781,X:SYS 60045 when X is 0-23)
$EAA1	60065	Plots character and color on screen. A = character, x = color (0-7)
$EAB2	60082	Finds color RAM relevant to current cursor position
$EA08	59912	Scrolls screen down one row (contents of 242 may affect this)
$E975	59765	Scrolls screen up one row

Screen

The screen is treated as device number 3, so files can be opened to the screen for input and output with OPEN 3,3. This provides INPUT without the normal prompt and can occasionally be useful in this sort of construction: OPEN 3,3: PRINT {HOME};:INPUT#3,X$ which reads the top line from the screen, subject to the usual rules governing INPUT.

Screen handling is complicated. Each ASCII value has to be converted into a POKE value; if it has some special purpose (like clearing the screen or setting reverse mode), a subroutine must carry this out. The way BASIC lines are temporarily made up of from one to four screen lines has to taken into account. ML programmers can trace this process from $FFD2, the Kernal routine CHROUT which prints a character, to $F27A and to $E742. From there, an entire range of processes is traceable, including delete and insert, cursor movements, screen scrolling, and placing the character and its color into the screen.

The line link table helps the system keep track of BASIC lines, which can wrap around up to four lines. Some bugs, notably when INPUT takes in its own prompt, occur as a result of this process.

Important screen locations. Some relevant locations are listed in Table 6-5. Chapter 12 (Graphics) explores many of them.

Alphabetical List of Utilities and Extensions to VIC BASIC

VIC's BASIC lacks a number of useful commands and structures that are built into some other BASICs. Many can be simulated, however. The following examples are grouped under headings of typical keywords, which indicate their functions. Note that these are typically BASIC subroutines, which must be run as usual, or machine language routines called by SYS.

The actual words listed (such as APPEND) will not by themselves activate any of these routines. A wedge or an intercept using a BASIC input vector is necessary to incorporate new keywords; while these techniques are interesting (they're explained elsewhere), they would significantly complicate matters here.

APPEND

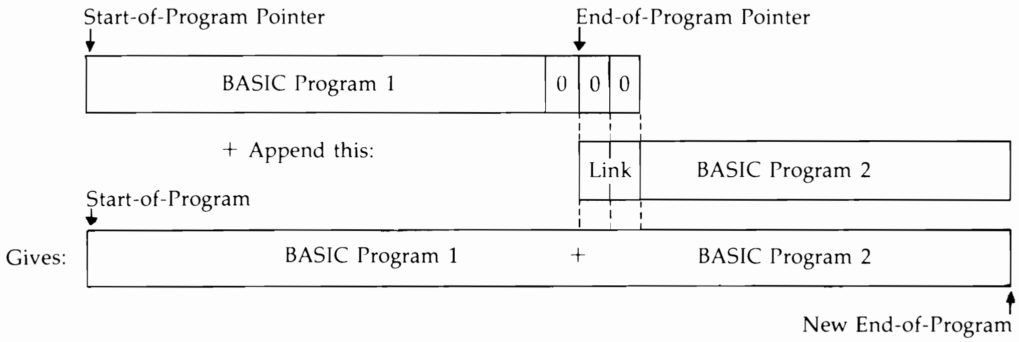
This BASIC system command can either add one file to the end of another, making a composite file, or link two BASIC programs end to end in a single program. Machine language can be linked like this too.

Disk files can be appended (see Chapter 15). Tape files can be appended; however, since the VIC has only one tape port, the process is more difficult.

BASIC programs are easy to append because the LOAD address is easily varied. Standard subroutines with high line numbers can be put onto the end of programs without the need to list the subroutines to the screen, load the program, enter some subroutine lines, save, and repeat. If the line numbers of the programs overlap, the normal editing won't work and you'll have unremovable lines of BASIC.

Figure 6-10 shows a program in memory, plus two of its pointers. Note how two zero bytes signify the end of the program. If the new program loads and overwrites this zero link address with its own link address, the programs append perfectly.

Figure 6-10. Appending Programs



The easiest approach is to first enter POKE 43, PEEK(45)−2:POKE 44, PEEK(46) in direct mode; then load or type in the new lines of BASIC and POKE 43,1: POKE 44,16 to start BASIC at $1000 (on the unexpanded VIC; substitute POKE 44,4 on a VIC with 3K expansion and POKE 44,18 with 8K or more expansion). Perfectly appended BASIC should result. Actually, this method is a shortcut; if PEEK(45) happens to be 0 or 1, you'll get an illegal quantity error and will need to edit your instructions to POKE 43, PEEK(45)+256−2: POKE 44, PEEK(46)−1, then continue as before.

AUTO

AUTO is a system command, not available in VIC BASIC, which automatically generates line numbers. Many utility packages (such as *Programmer's Aid*) contain versions of this command. Program 6-15 is a BASIC subroutine, which uses the keyboard buffer to take in complete lines. The POKE in line 60010 flashes the cursor, line 60040 prints the current values of S and I, and line 60050 puts two carriage returns in the keyboard buffer.

Program 6-15. AUTO: Automatic Line Numbering

```
60000 PRINT"ENTER START, INCREMENT":INPUT S,I

60010 PRINT "{CLR}{3 DOWN}";S;:POKE204,0
60020 GET A$:IF A$=""GOTO 60020
60030 PRINT A$;:IF ASC(A$)<>13 GOTO60020
60040 PRINT"S=" S+I ":I=" I ":GOTO 60010":PRINT"
      {HOME}"
60050 POKE 631,13:POKE632,13:POKE198,2:END
```

BLOCK LOAD and BLOCK SAVE

VIC's LOAD and SAVE commands are designed solely for the benefit of users of BASIC. They automate BASIC program recovery and storage, in a way which is pleasantly transparent. Programs load into memory into the correct area, and are saved to tape or disk without any need to know about pointers or other inside information.

However, there are situations when the special assumptions connected with BASIC do not apply. When a block of machine language, a collection of graphics characters, or a set of variables and arrays after BASIC is to be saved intact to tape or disk, normal saving cannot possibly work since the machine can't know what area of memory you want saved. In addition, loading such blocks back into memory may be tricky; the machine language or data is liable to be treated as though made up of BASIC lines, and become corrupted through being linked.

Note that ML monitors (like *VICMON*) have commands like .S "NAME ", 01,1000,2000 (save the contents of $1000–$1FFF to tape and call it NAME) and .S "DISK NAME", 08,1000,2000 (save the same data to disk with the title DISK NAME) to perform block loads and saves. Note also that data above $8000 can't be saved to tape with the VIC-20, although it can be with disks. Chapter 14 explains how memory areas in this region, notably the ROM area from $A000 to $BFFF, can be stored on tape by first moving them lower down in memory.

BLOCK SAVE. The obvious way to save data other than BASIC programs is to poke new start and end addresses. For example, you could use POKE 43,0: POKE 44,24: POKE 45,0: POKE 46,26: SAVE "NAME", 1,1. This will save data from locations $1800 to $19FF because the value in the start-of-BASIC pointer is changed to $1800 and the value in the end-of-BASIC pointer is changed to $1A00. As far as VIC is concerned, this becomes the correct area to save. (Note that the last byte at $1A00 is not saved; SAVE stops when it reaches it.) The secondary address of 1, with tape, forces the data to LOAD back into the same area as that from which it was saved.

There may be problems in using this technique within a BASIC program. It's necessary to restore the pointers after use; they must be saved with POKEs and recovered with PEEKs, or they will be lost.

An alternative technique is given in Program 6-16. It has the same effect as the POKE-and-SAVE method. SYS 57809 takes in the parameters which save or load. You can replace NAME with the name of your choice, and change the first 1 to an 8 to save to disk instead of tape. POKE the starting address of the SAVE into locations 193 and 194, and the ending address + 1 into locations 174 and 175.

Program 6-16. BLOCK SAVE

```
1000 SYS57809"NAME",1,1
1010 POKE 193,0:POKE194,24
1020 POKE174,0:POKE175,26
1030 SYS63109
```

You can watch the effect of this program by typing SYS 57809 "HELLO",8,1: PRINT PEEK(186); PEEK(183); PEEK(187) + 256*PEEK(188); PEEK(185). The values displayed are the device number, name length, pointer to name, and secondary address, respectively.

BLOCK LOAD. To load a machine language routine into memory, the easiest way is simply to use LOAD "NAME", 1,1 (assuming the machine language has been saved with a forced LOAD address). Within a program, a maneuver like the one below is needed to avoid the automatic chaining feature:

0 IF X=1 GOTO 2
1 X=1: LOAD "NAME", 1,1
2 REM CONTINUE FROM HERE...

There are, of course, limitations on these methods. The first may corrupt its data; the second is tiresome if you have several LOADs. Program 6-17 shows one technique that works within programs without interrupting program flow.

Program 6-17. BLOCK LOAD

```
1000 POKE147,0
1010 SYS 57809"SCREEN",8,1
1020 SYS 62795
```

Example. To see how these routines work, consider Program 6-18 which saves and loads a VIC color screen to disk. Lines 10–30 save the contents of locations $1E00–$1FFF to disk with the title SCREEN. This is the correct area for screen memory in an unexpanded VIC-20. Line 20 saves the corresponding color RAM ($9600–$97FF); line 30 saves the VIC registers. If user-defined characters were used, these must be SAVEd too. Between them, these completely define any picture starting at $1E00.

Lines 500 to 520 reconstruct the picture. Try typing this program into an unexpanded VIC attached to a disk drive, then put a few random colored characters on the screen. RUN will store the screen's information on disk. Clear the screen; type RUN 500, and you'll see the screen reconstruct itself.

Program 6-18. Using BLOCK SAVE and BLOCK LOAD to Save a VIC Screen

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 SYS 57809 "@:SCREEN",8,1:POKE 193,0:POKE194,30:
   POKE174,0:POKE175,32:SYS 63109         :rem 216
20 SYS 57809 "@:COLOR",8,1:POKE194,150:POKE175,152
   :SYS 63109                              :rem 59
50 END                                     :rem 60
70 REM                                     :rem 75
500 POKE147,0: SYS 57809 "SCREEN",8,1:SYS 62795
                                           :rem 13
510 SYS 57809 "COLOR",8,1:SYS 62795       :rem 108
600 GOTO600:REM DISPLAY SCREEN UNTIL STOP KEY
                                           :rem 20
```

Writing a version of this program for tape is more difficult. Memory in the area used for both color RAM and the VIC chip cannot be saved directly, and the process is slow. Change lines 10 and 500 to the tape equivalents (10 SYS 57809 "SCREEN",1,1 etc.) and delete 20, 30, 510, and 520. The screen can be saved, but without its color information; it's easiest to supply this separately.

CHAIN

Chaining is the process by which one program loads and runs another. For example, a set of programs may exist on disk, each separately accessible by a menu, so that only one program is in memory at one time and the menu is reentered on exit from any called program. VIC BASIC (and PET/CBM and 64 BASICs) chains whenever LOAD takes place inside a program. LOAD, then run without CLR (to retain previous variables), is automatic.

Although basically simple, this is not quite as straightforward as it seems. Earlier in this chapter you saw how problems can occur when the chained program is longer than the program which loaded it. You may also encounter occasional difficulties with strings and function definitions.

Try the following very short illustration:

1. Save this on tape:

```
10 PRINT "FIRST PROGRAM"
20 A=10: B%=100: C$="HELLO" + ""
30 LOAD "SECOND PROGRAM": REM CHAINS SECOND PROGRAM
```

2. Now type this in, and save it as "SECOND PROGRAM":

```
10 PRINT "SECOND PROGRAM"
20 PRINT A,B%,C$
```

Rewind the tape and load and run the first program. It will almost immediately reach line 30. Load the second program, and run it, while retaining the variables. Line 20 of the second program prints 10, 100, and HELLO, the values assigned by the first program. Note that when the LOAD is within a program, there's no LOADING PROGRAM message and no other messages unless the cassette button isn't pressed.

Chaining machine language. The easiest way to load and run ML is to use the Kernal LOAD routine followed by a jump to the newly loaded ML program. This is explained in detail in Chapter 8.

COLOR Border, Screen and COLOR Character

BASIC graphics packages often have a command called COLOR. All it does is poke a border and background color into the relevant VIC chip register at $900F (36879). A BASIC version might assume BO is the border color (0–7) and SC is the screen background color (0–15). In that case, all that's needed is POKE 36879,8 + BO + 16 * SC. Alternately, the keypress values (1–8, or 1–16 with some work) may be more suitable.

Chapter 12 has a routine to set all current color RAM to any selected color.

Compile

Compilation is a process by which a language like BASIC is converted into pure machine language. A program which performs this conversion is called a compiler. BASIC (the source code) is translated into ML object code; typically it will LIST as a single SYS call, which is followed by a large (but not listed) ML program.

Compilation is a different order of magnitude from other utilities discussed here. So far as I'm aware no compiler exists for VIC, mainly because of its small memory.

However, it's possible that VIC programs compiled on a 64 could run on the VIC with expansion memory, if the compiler uses only Kernal routines, so a short discussion is justified.

Briefly, any compiler of an unstructured language like CBM BASIC must first build up a table of all the program's variables and arrange a position for each of them in memory. Strings will need pointers, and will be liable to garbage collection problems unless they are each assigned 256 bytes. When the variables are dealt with, every BASIC statement must be converted into its ML equivalent; the result is typically a set of segments which are linked to make up the compiled code.

Speed increases of 10 to 50 or more times are claimed, but in practice even a tenfold increase is probably optimistic. Some of the improvement is directly due to the replacement of BASIC statements, with all their overhead and housekeeping, by relatively straightforward processing.

By itself this is not a major factor. Well-written compilers have their own arithmetic routines, using integers where possible to save time. There's considerable room for ingenuity; for example, a line like 100 GET X$: IF X$="" GOTO 100, which is often found in CBM BASIC, could be replaced by five bytes of machine language.

So-called tiny compilers, working with a restricted set of BASIC (to save the effort of implementing every command) are occasionally encountered.

Computed GOSUB and Computed GOTO

These functions use a formula or label, instead of a number, for their destination line. Some computer languages use GOSUB VALIDATE to perform a subroutine called VALIDATE. Obviously, statements like this are likely to be more readable than CBM BASIC's GOSUB 10000. But there are a couple of things to watch out for. Don't confuse computed destinations with ON-GOTO-, which provides a choice of destinations according to the value just after ON. Note, too, that any parts of BASIC using computed destinations can't usually be renumbered by a utility.

Can you get a version of these functions with VIC BASIC? It is not too difficult, and versions of each (using a SYS call plus an expression for a line number) are given here. For example, using the first of the ML routines below, SYS (828) DATE when DATE=1000 acts just like GOTO 1000, and SYS (840) CHECK when CHECK=2000 acts just like GOSUB 2000.

SYS (828) 500+10*X and other arithmetic functions can be used too. A wedge to intercept BASIC is more trouble, but more readable; you can use GOTO DATE, GOSUB CHECK, and so on. See Chapter 9.

Computed GOTO. This is shorter than GOSUB, because nothing has to be stored on the stack. If you try SYS (51531) 10, you'll find it behaves just like GOTO 10. This is not surprising, since 51531 is the beginning of the ROM routine that performs GOTO. If you replace the routine to fetch an ASCII number with a routine that calculates a value, you have a short computed GOTO:

```
$033C JSR  $CD8A  ; Input and evaluate numeric expression
$033F JSR  $D7F7  ; Convert to two-byte integer or error message
$0342 JSR  $C8A3  ; Enter GOTO, skipping input of fixed number
$0345 JMP  $0079  ; Continues at next BASIC character
```

Computed GOSUB. SYS (51331) 10 acts like GOSUB 10. Again, if you insert the following ML to calculate the destination, you have a computed GOSUB:

```
$0348  LDA #$03   ; Test whether stack has room for six bytes
$034A  JSR $C3FB  ; and print OUT OF MEMORY message if not
$034D  LDA $7B
$034F  PHA        ; GOSUB saves the pointer into BASIC,
$0350  LDA $7A
$0352  PHA
$0353  LDA $3A
$0355  PHA        ; and the current line number.
$0356  LDA $39
$0358  PHA
$0359  LDA #$8D   ; GOSUB token is for RETURN to identify.
$025B  PHA
$025C  JSR $0079  ; Move to next BASIC character, at start of numeric expression
$025F  JSR $CD8A  ; Input and evaluate (like computed GOTO),
$0262  JSR $D7F7  ; Convert to two bytes,
$0265  JSR C8A3   ; Find computed line number, go to it,
$0268  JMP C7AE   ; Continue with BASIC.
```

Both routines can be freely relocated. If you want to try these routines, but do not yet know ML, load the ML with the simple BASIC loader in Program 6-19.

Program 6-19. BASIC Loader for Computed GOTO and GOSUB

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 DATA 32,138,205,32,247,215,32,163,200,76,121,0
                                        :rem 83
20 DATA 169,3,32,251,195,165,123,72,165,122,72,165
   ,58                                  :rem 57
30 DATA 72,165,57,72,169,141,72,32,121,0,32,138,20
   5                                    :rem 199
40 DATA 32,247,215,32,163,200,76,174,199  :rem 181
50 FOR J=828 TO 874:READX:POKEJ,X:NEXT    :rem 25
```

CRUNCH

This utility is not generally a part of any BASIC dialect. The idea of crunching a program is to delete any parts which are not necessary for the program's operation, with the aim of increasing BASIC's execution speed. For this reason, compacting is an alternative name for the process. Conversely, uncrunching means spacing a program out. It has nothing to do with speed; the intention is to make a program more readable and documentable. For example, if you wish to decipher someone else's crunched program, a utility which lists each instruction on a separate line and puts in spaces may well help legibility. See LIST, as well as Chapter 8 (which has a short ML routine to separate BASIC statements onto new lines).

The rationale for CRUNCH is that REM statements, spaces, short lines, and so on slow the BASIC interpreter by making it waste time jumping past spaces, switching to new lines, and so on. How much speed increase is likely? Not a great deal. The appeal is really of the "every little bit helps" type. Combined with renumbering lines (0,1,2,...) and adding an extra line or two of DIM statements to order the main variables, CRUNCH will help things along as much as can be hoped from such mechanical methods.

Crunching should remove REMs. If these are referenced by GOTO or GOSUB, they may be retained, or the reference may be changed to the next line. It should also remove all spaces not within quotes; however, a program crunched in this manner may require some adjustment. For example, if the statement X=T AND U is crunched, BASIC will think it contains the function TAN.

Crunching should also combine as many lines together as possible. Lines spanning more than 255 bytes should be avoided, since most BASIC pointers—the one for DATA, for example—are only a single byte. Thus, the longest line is generally limited to 250 BASIC characters. Note, however, that a line may be referenced (for instance, by GOTO) and therefore not be combinable with previous line(s).

Crunching could also renumber from 0 upwards in steps of 1; reduce all variables' names to a single character; remove spare semicolons from PRINT statements; modify CHRGET to remove its test for spaces (see Chapter 8); slow the rate of interrupts (or temporarily stop them if the keyboard will not be used); or remove wedges which intercept BASIC and usually slow it.

In practice, it is often simplest to manually remove spaces and REMs and collect lines together. Since crunching actually changes the program, BASIC has to use a keyboard buffer POKE technique, listing an individual line without spaces, entering it, and starting over with the next line.

DEEK

Double-byte PEEK returns the value in two consecutive addresses, assuming they follow the 6502 convention of low byte then high byte. Use this formula:

DEF FN DEEK (X) = PEEK(X) + 256 * PEEK(X+1)

DEL

Deletion of an unwanted portion of a BASIC program, to remove test routines and temporary features, is performed in other dialects of BASIC by DEL a-b, with syntax similar to LIST (except that DEL alone should not delete everything). The *Programmer's Aid* cartridge includes a version. DEL seems to have been omitted from Commodore's originial BASIC specifications.

Program 6-20 is a BASIC subroutine designed to reside at the end of a program. It works by searching out line numbers within a specified range, then deleting the line using a trick with the keyboard buffer to simulate entry of the line number at the keyboard.

Program 6-20. DELete

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
61000 INPUT "DELETE FROM, TO"; L,U:{2 SPACES}A=PEE
      K(43) + 256*PEEK(44)                :rem 238
61010 DEF FN DEEK(A) = PEEK(A)+256*PEEK(A+1)
                                          :rem 255
```

```
61020 IF FN DEEK(A+2)<L THEN A=FN DEEK(A): GOTO 61
      010                                    :rem 3
61030 IF FN DEEK(A+2)>U OR FN DEEK(A)=0 THEN END
                                            :rem 11
61040 N=FN DEEK(A+2): PRINT "{CLR}" N       :rem 14
61050 PRINT "A=" A ":U=" U ":GOTO 61010"  :rem 160
61060 POKE 631,19: POKE 632,13: POKE 633,13: POKE
      {SPACE}198,3: END                      :rem 0
```

Line 61010 skips through link addresses until a line number in the specified range is found. Line 61020 stops either out of the range or at the end of a program. Line 61030 prints a line number on the screen, and the rest of the subroutine simulates three RETURN keypresses. A more elegant approach is to move the entire program from the upper line number down, to overlay BASIC from the lower line number; the link addresses then have to be restored.

DOKE

Double-byte POKE puts a value from 0 to 65535 into any two consecutive bytes, assuming that the standard 6502 convention of low byte/high byte applies. There's no way to write this as a function without writing a SYS routine of the form SYS m, n or using a wedge; instead, DOKE ADDRESS, VALUE can be represented by POKE AD,VA−INT(VA/256)*256: POKE AD+1,VA/256

DUMP

One form is a screen dump, which prints a duplicate of the screen onto paper; it may be useful as a record or interesting as a picture. It is relatively easy to print normal characters when user-defined characters aren't used, since all that's needed is a PEEK into RAM followed by printout of the corresponding characters. Complications include high-resolution graphics, color (where conversion to black-and-white may lose detail) and the fact that Commodore printers have the Commodore character set while other printers may not.

Program 6-21 is a BASIC dump that assumes the normal VIC character set.

Program 6-21. Screen Dump

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
40010 SCREEN = 256*PEEK(648):OPEN 4,4:CMD 4:rem 19
40100 FOR J=0 TO 22:FOR K=0 TO 21:X=PEEK(SC+J*22+K
      )                                     :rem 205
40110 IF X>128 THEN X=X-128:PRINT CHR$(PRINTER REV
      S ON);                                :rem 239
40120 IF X<32 THEN PRINT CHR$(X+64);        :rem 181
40130 IF X>31 AND X<64 THEN PRINT CHR$(X);:rem 243
40140 IF X>63 AND X<96 THEN PRINT CHR$(X+32);
                                            :rem 142
40150 IF X>95 AND X<128 THEN PRINT CHR$(X+64);
                                            :rem 197
40160 PRINT CHR$(146);                      :rem 176
40170 NEXT:PRINT:NEXT:PRINT#4:CLOSE4        :rem 125
```

Line 40010 computes the top-left screen position and OPENs a file to the printer. Line 40100 starts a loop, which PEEKs every individual screen location (assuming a 22-column × 23-row screen) and prints the corresponding character. Line 40110 looks for reversed characters and turns on the printer's reverse mode if any are detected. CHR$(18) turns on reverse mode for Commodore's printers, with CHR$ (146) also needed in line 40160 to turn it off. Replace these values with those appropriate to your printer. Chapter 17 contains further information on the use of printers.

Another form of dump, the variable dump, lists the current values of variables. Often array variables are ignored. Of course, values can simply be printed by inserting a program line, so a dump of this kind is not essential to debugging BASIC.

There's no difficulty writing dumps in BASIC. You've seen how variables and their types are stored, so variables' names and values can be deciphered and printed out. They're printed in the same order that BASIC defined them (that is, in the sequence in which they are stored after BASIC).

An alternative procedure which gives a sorted list is to cycle through all the variable names and types from A, A0–A9, AA–AZ, ..., B%, and so on; each variable can be sought by the ROM routine (also used in the VARPTR utility in this chapter) and printed with its name. That is what Program 6-22 does.

Program 6-22. Variable Dump

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
0 DATA 165,22,72,160,32,162,11,189,185,3,157,32,2,
  202,16,247                               :rem 125
1 DATA 140,41,2,192,32,240,28,140,36,2,208,23,142,
  34,2,173                                   :rem 6
2 DATA 18,232,201,239,208,8,104,133,22,104,104,76,
  116,196,162                              :rem 175
3 DATA 48,142,35,2,169,32,133,122,169,0,133,72,32,
  134,206,165                              :rem 176
4 DATA 72,240,19,169,32,133,122,173,34,2,141,39,2,
  173,35,2                                  :rem 25
5 DATA 141,40,2,32,157,202,174,35,2,232,224,58,144
  ,211,224                                  :rem 16
6 DATA 65,144,247,224,91,144,203,174,34,2,232,224,
  65,144,251                               :rem 137
7 DATA 224,91,144,171,160,36,204,41,2,240,174,144,
  139,200,208                              :rem 174
8 DATA 136,34,32,32,65,146,61,34,32,65,32,59
                                           :rem 120
10 REM ******************************************
   ****************                        :rem 243
11 REM *{3 SPACES}ALPHABETICAL ORDER DUMP OF CURRE
   NT BASIC VARIABLES{4 SPACES}*             :rem 89
12 REM * SYS 828 (DIRECT MODE) DUMPS NON-ARRAY VAR
   IABLES' VALUES *                          :rem 29
13 REM ******************************************
   ****************                        :rem 246
```

```
100 FOR J=828 TO 963: READ X: POKE J,X: NEXT
                                                  :rem 68
```

FIND

See SEARCH

LIST

LIST can be modified fairly easily; two particularly useful versions follow. The first, Program 6-23, gives you a window of up to six lines of BASIC at a time, which can be scrolled up or down. This is helpful when examining BASIC without the benefit of a printer. The second routine, Program 6-24, is machine language; it alters LIST to expand the not-very-readable reversed characters of VIC listings into text. Printer owners may like to list programs in this format.

A window on BASIC. If you append this routine to your BASIC program and RUN 63000, it will list several lines on the screen. The actual number is selectable in line 63000. Change the 6 in the part of the statement that currently reads N=M+6 to the desired number of lines. Obviously, since any single line of BASIC can occupy as many as four screen rows, six lines may be too much for the screen to hold. Lines 63110 and 63120 are printed on the screen (in white) and list several lines (in black, to underline the difference from a normal listing) before returning to test for the f1 key pressed (which LISTs upward through the program) or the f7 key pressed (which LISTs downward through the program). The current starting line is M, and subroutine 63300 scans the program finding which line numbers to list. After LIST, the keyboard buffer is POKEd to simulate {CLEAR} RETURN {CLEAR}{DOWN} RETURN.

Program 6-23. Window Listing of BASIC

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
63000 N=M+1:GOSUB 63300: L1=L: N=M+6: GOSUB 63300
                                                  :rem 76
63110 PRINT "{CLR}{WHT}LIST"L1"-"L               :rem 218
63120 PRINT "M="M":GOTO 63200{BLK}"              :rem 230
63130 POKE 198,5: POKE 631,19: POKE 632,13: POKE 6
      33,19                                      :rem 247
63140 POKE 634,17:POKE 635,13: END                 :rem 6
63200 GET X$: IF X$="" GOTO 63200                 :rem 81
63210 IF X$="{F1}" THEN M=M+1                    :rem 111
63220 IF X$="{F7}" AND M>0 THEN M=M-1              :rem 3
63230 GOTO 63000                                  :rem 48
63299 REM *** FIND VALUE OF M'TH LINENUMBER, START
      ING FROM M=1 ***                           :rem 168
63300 J=0: L=PEEK(43)+256*PEEK(44)                :rem 35
63310 J=J+1: IF J<N THEN L=PEEK(L)+256*PEEK(L+1):
      {SPACE}IF L>0 GOTO 63310                    :rem 10
63320 IF (L=0) OR (PEEK(L)+256*PEEK(L+1)=0) THEN L
      =63999: RETURN                             :rem 174
63330 IF J=N THEN L=PEEK(L+2)+256*PEEK(L+3): RETUR
      N                                          :rem 102
```

BASIC can't be edited while using this routine, since the entire process is under program control; listing is all that is allowed. However, it is possible to write a line-editing program by combining this method with parts of AUTO.

Legible LIST. Program 6-24 loads a transparent machine language program which provides bracketed text in place of most of VIC's special reverse video control characters. For the sake of user friendliness, the BASIC program POKEs the machine language into the topmost available BASIC memory, then lowers the pointers to protect itself. It can be turned off with a SYS call, so that editing retains the original characters without introducing unwanted expressions in brackets.

Program 6-24. Listing VIC-20 Control Characters in Brackets

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
Ø DATA 173,7,3,73,223,141,7,3,173,6,3,73,11,141,6,
  3,96,8,133,255,152,72,36,15,48,8          :rem 185
1 DATA 1Ø4,168,165,255,4Ø,76,26,199,162,Ø,232,189,
  -188,24Ø,24Ø,197,255,2Ø8,246,16Ø          :rem 221
2 DATA Ø,2ØØ,185,-156,2Ø1,91,2Ø8,248,2Ø2,2Ø8,245,3
  2,21Ø,255,2ØØ,185,-156,2Ø1,93              :rem 23
3 DATA 2Ø8,245,32,21Ø,255,1Ø4,168,165,255,4Ø,76,24
  6,198,144,5,28,159,156,3Ø,31,158          :rem 217
4 DATA 18,146,147,19,148,2Ø,145,17,157,29,16Ø,255,
  133,137,134,138,135,139,136,14Ø           :rem 163
5 DATA Ø,Ø,Ø,Ø,91,66,76,65,67,75,93,91,87,72,73,84
  ,69,93,91,82,69,68,93,91,67,89,65          :rem 61
6 DATA 78,93,91,8Ø,85,82,8Ø,76,69,93,91,71,82,69,6
  9,78,93,91,66,76,85,69,93,91,89           :rem 255
7 DATA 69,76,76,79,87,93,91,82,86,83,93,91,82,86,8
  3,79,7Ø,7Ø,93,91,67,76,82,93,91           :rem 249
8 DATA 72,79,77,69,93,91,73,78,83,93,91,68,69,76,9
  3,91,85,8Ø,93,91,68,79,87,78,93            :rem 11
9 DATA 91,76,69,7Ø,84,93,91,82,73,71,72,84,93,91,8
  3,72,45,83,8Ø,67                          :rem 234
1Ø DATA 93,91,8Ø,73,93,91,7Ø,49,93,91,7Ø,5Ø,93,91,
   7Ø,51,93,91,7Ø,52,93,91,7Ø,53,93         :rem 231
11 DATA 91,7Ø,54,93,91,7Ø,55,93,91,7Ø,56,93:rem 85
1ØØ T=PEEK(55) + 256*PEEK(56)               :rem 213
11Ø L=T-268                                  :rem 6Ø
12Ø FOR J=L TO T-1: READ X%                 :rem 117
13Ø IF X%<Ø THEN Y=X%+T: X%=Y/256: Z=Y-X%*256: POK
    E J,Z: J=J+1                             :rem 33
14Ø POKE J,X%: NEXT                          :rem 48
15Ø REM L+17 IS ENTRY POINT TO MACHINE CODE ROUTIN
    E IN RAM                                 :rem 97
16Ø REM HI BYTE $C7=199                     :rem 19Ø
162 H%=(L+17)/256                            :rem 63
164 P= (H% OR 199) AND NOT (H% AND 199)      :rem 34
166 POKE L+4,P                              :rem 243
17Ø REM LO BYTE $1A=26                      :rem 134
172 L%=(L+17) - 256*H%                      :rem 217
```

```
174 P= (L% OR 26) AND NOT (L% AND 26 )      :rem 181
176 POKE L+12,P                              :rem 35
200 POKE 55,L-INT(L/256)*256: POKE 56,L/256 :rem 3
210 SYS L                                   :rem 222
300 PRINT "{CLR}{2 DOWN}SYS"L               :rem 100
310 PRINT "TOGGLES THE SPECIAL               :rem 58
320 PRINT "'LIST' FUNCTION ON/OFF           :rem 219
330 PRINT "NOTE: POKE" L+99 ",29            :rem 168
340 PRINT "IF SH-SPACE NOT WANTED           :rem 189
400 NEW: REM DELETE BASIC, RETAINING SPECIAL LIST
                                            :rem 187
```

A SYS call modifies LIST's vector so that it points within the routine, where all characters in quotes are checked. This part of the program is in fact quite small. Entering SYS to the same address toggles LIST off; a special exclusive-OR feature has been used, so that only one address need be noted.

Most of this program is made up of two tables (one of the ASCII characters and one of their translated form within brackets), so the program can be easily modified to allow for graphics characters or to put in your own alternative forms. The ASCII values of the brackets [and] are 91 and 93; Table 6-6 lists the ASCII values and text for the special characters for the program as it stands.

Table 6-6. ASCII Values of Special Characters for Program 6-24

{BLACK}	144	{CLR}	147	π	255
{WHITE}	5	{HOME}	19	{F1}	133
{RED}	28	{INS}	148	{F2}	137
{CYAN}	159	{DEL}	20	{F3}	134
{PURPLE}	156	{UP}	145	{F4}	138
{GREEN}	30	{DOWN}	17	{F5}	135
{BLUE}	31	{LEFT}	157	{F6}	139
{YELLOW}	158	{RIGHT}	29	{F7}	136
{RVS}	18	{SPACE}	32	{F7}	140
{RVSOFF}	146	{SHIFT-SPACE}	160		

Printers can use this program successfully. However, lines containing special characters will be longer than usual.

MERGE

Combining two BASIC programs into a single program, with the lines sorted correctly, is called a *merge.* You can use MERGE, for example, to insert standard subroutines without the need for retyping. Many BASIC extension packages have MERGE; because of the flexible way merging is done, this command can also perform other functions (such as loading PET/CBM tapes into the unexpanded VIC).

Tape merge. The following procedure involves storing the subroutine(s) to be merged as sequential file(s), not as tokenized programs, then reading them back using the keyboard buffer to simulate entry of each line by RETURN.

To save the subroutine on tape as a file, use OPEN 1,1,1,"NAME OF SUBROUTINE": CMD 1: LIST [OPTIONAL LOW–HIGH LINES]. When the cursor returns, type PRINT#1:CLOSE 1 to close the file and write the last portion of data to tape.

Merging can be carried out whenever you have a program in memory. The result will be a fully merged program, as if the lines had been separately typed at the keyboard. Note that lines entered with any BASIC abbreviations and which are abnormally long when listed may need to be divided into shorter lines.

Use the following procedure to merge program lines. Start with a program in memory and the tape in the cassette; then POKE 19,1: OPEN 1,1,0, "NAME OF SUBROUTINE" to read the tape until it finds the correct header. This will be signaled by FOUND.

At that point, it will wait for the file to be read. Type {CLEAR} and {DOWN}{DOWN}{DOWN}. Then POKE 153,1: POKE 198,1: POKE 631,13: PRINT CHR$ (19) and press RETURN, and the tape file will be automatically read and merged. Eventually, a ?SYNTAX ERROR message appears; this is not a mistake, but a result of either the tape or the program having no more lines left. It means that the merge is finished.

Disk merge. Program 6-25 uses 144 bytes from ROM, modified slightly in RAM, to merge new lines into a BASIC program in memory. It has a driver routine starting at $033E (830) which fetches single characters of BASIC, building them into the input buffer.

Program 6-25. Disk Merge

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 S=PEEK(55) + 256*PEEK(56)                 :rem 164
20 S=S-145: REM S IS START ADDRESS           :rem 173
22 POKE 56,S/256: POKE 54,S/256: POKE 52,S/256
                                              :rem 129
24 POKE 55, S AND 255:POKE 53,S AND 255:POKE 51, S
   AND 255                                    :rem 105
30 X=0: FOR J=S TO S+144:POKE J, PEEK(50338+X): X=
   X+1: NEXT                                  :rem 180
50 POKE S+85,210: POKE S+139,76                :rem 91
60 POKE S+75,234:POKE S+76,234:POKE S+77,234
                                              :rem 119
70 POKE S+136,234:POKE S+137,234:POKE S+138,234
                                                :rem 2
90 REM MERGE PROGRAM FOR BASIC PROGRAMS ON DISK
                                               :rem 81
100 FOR J=830 TO 889: READ X: POKE J,X:NEXT:rem 68
1000 DATA 162,8,32,198,255,32,207,255,32,207,255,1
     60,0,32,207,255                          :rem 120
1001 DATA 32,207,255,240,33,32,207,255,133,20,32,2
     07,255,133,21                              :rem 1
1002 DATA 32,207,255,153,0,2,240,3,200,208,245,152
     ,24,105,5,168                            :rem 254
1003 DATA 32,162,196,208,215,240,213,32,89,198,76,
     128,196                                   :rem 10
```

Use this program by entering LOAD, RUN, and NEW. Load or type a program into memory, and additional programs can be merged into it with OPEN 8, 8, 8, "PROGRAM NAME": SYS 830. Turn off the disk light with OPEN 15,8,15: CLOSE 15.

MOD

This is an arithmetic function, found in some BASICs, which calculates the remainder when one integer is divided by another. MOD is an abbreviation of *modulo,* a mathematical term used in number theory. The statement "4=19 modulo 5" means that 4 and 19 have the same remainders when divided by 5. The simplest BASIC version is DEF FN MOD(N) = N−INT(N/D)*D, where D is the divisor. This formulation may be of use when converting other BASICs to CBM BASIC.

Examples of the use of MOD are D=12:H=FN MOD(16), which converts 16 hours to 4 o'clock, and D=256: PRINT FN MOD (50000), which prints the low byte of 50000.

OLD and BASIC Recovery

Originally, OLD was used to restore a program which had been inadvertently removed by NEW. However, the VIC-20 offers two other important uses, which are covered under the heading BASIC recovery. These are best distinguished from the start.

Program 6-26. OLD

```
10 FOR J=320 TO 340:READX:POKEJ,X:NEXT
20 DATA 169,1,168,145,43,32,51,197,165,34
30 DATA 105,2,133,45,165,35,105,0,133,46,96
```

OLD as UnNEW. This restores BASIC because NEW leaves most of the program intact, simply arranging pointers as though no program were present, and putting a zero link address at the very start of BASIC. (It also sets GETCHR and the RESTORE pointers, clears variables, and closes files.) Run the program, type NEW, then type SYS 320:LIST. The program is restored perfectly. In fact, its variables are even retained. If BASIC has not been NEWed, or even if it's running, the SYS call does no harm.

OLD after chaining. When used with LOAD within a program, this restores BASIC when the new program is longer than the old one. As explained under CHAIN, in order to pass variables from one chained program to the next, the end-of-program pointers are not set. Thus, if the newly loaded program is too long, its top end will be corrupted. However, if the ML routine has been POKEd in by the loader program, 0 SYS 320:CLR at the start of the new program will prevent corruption. (CLR is needed to remove garbage after the program, which may appear as pseudo-data; it is not possible to recover the overwritten variables.)

The reason location 320 was chosen as a starting address for this routine was to allow it to be used with tape LOADs. If locations 828 and following had been used, as in some of the other examples, the tape buffer where the routine resides would be overwritten. The routine can be relocated anywhere in free RAM.

OLD restores BASIC after SYS 64802 or after a hardware reset switch has apparently returned the machine to its post-switch-on state, as explained in Chapter 5. Both these routines leave BASIC RAM unaltered and in effect perform NEW, so SYS 320 is just as effective as with NEW. Note that 320 is also protected against resets. The stack isn't touched. Thus, any BASIC program, including one which disables STOP and RESTORE, can be freely reset and recovered by this method (if you have a hardware reset switch).

The following ML routine could be put in RAM or ROM. The machine-language uses the built-in feature to link BASIC lines. It must have a nonzero link address, which is why the first commands put 1 just after the location which 43 and 44 point to (the first link address which NEW zeros). The end-of-program pointers also have to be set, because the relinking routine doesn't set them.

```
LDA   #$01
TAY
STA   ($2B),Y  ; Nonzero byte in first link address
JSR   $C533    ; Make all links consistent
LDA   $22      ; Add 2 to the final address,
ADC   #$02     ; Giving the end-of-program address
STA   $2D      ; Which is stored in its pointer
LDA   $23
ADC   #$00
STA   $2E
RTS
```

BASIC version of OLD. Is there a BASIC equivalent of the above? Yes, but the trouble is that the end-of-program pointers get lost and take some effort to retrieve. If the end-of-program isn't moved up, variables will overwrite your program when it runs:

```
POKE PEEK(44)*256+2,1:SYS 50483:POKE 46, PEEK(56)-1:CLR
```

This assumes that BASIC starts in one of the normal places and that the end-of-program pointer's position isn't critical (it becomes set to a location 256 bytes below the end of BASIC memory). The program will now LIST properly.

A more complete version, which can also be used from within a chained program when a short program loads a longer one, is given in Program 6-27.

Program 6-27. A BASIC Version of OLD

```
Ø POKE PEEK(43) + 256*PEEK(44)+1,1:SYS5Ø483:POKE46
  ,PEEK(56)-1:CLR
1 L=PEEK(43)+256*PEEK(44):FORJ=1 TO 9E9:L2=PEEK(L)
  +256*PEEK(L+1)
2 IF L2>ØTHENL=L2:NEXT
3 POKE 45,(L+2)AND 255:POKE 46,(L+2)/256
```

ONERR

VIC has an indirect vector at $0300/0301, which is locations 768 and 769 in decimal, to process error messages. Usually, this is set to $C43A, and the actual error is dictated by the contents of the X register. You can mimic this with POKE 781, *error* *number*: SYS 50234, where *error number* is between 1and 30.

ONERR usually works by specifying a line number to GOTO in the event of error. The advantage is that the program cannot crash; the drawback is that processing ONERR properly is liable to be unacceptably long, since many possible errors may have caused the problem.

Chapter 8 gives an example which identifies the position of errors in BASIC program lines.

PAUSE

There are two versions of this command. The first waits for a timed delay; the second is a coffee-break type command which temporarily freezes BASIC or ML.

Timed delays are useful with some types of music programs. BASIC delay loops (FOR J=1 TO 500: NEXT) provide an excellent method, though the actual timing varies with the stored position of the loop variable in memory; if J is set up as the first variable, this problem no longer applies.

The VIC's internal clock is another obvious way to get accurate timing. The clock is stored in 160, 161, and 162, with 162 changing fastest. The tidiest routine is therefore POKE 162,X: WAIT 162,2↑N which has a maximum delay of 255/60, or about four seconds.

To see how the formula works, note that WAIT stops until just one bit is set. POKE 162,0:WAIT 162,64 delays until location 162 reaches 64, and so pauses for 64/60, or just over one second. POKE 162,4:WAIT 162,64 therefore gives a one-second delay. The timing is reasonably constant, although the first POKE could occur at any time between interrupts, so there's a sixtieth of a second maximum difference in pauses. Unless the interrupt rate is changed, resolution below about 1/60 second isn't possible.

Delays longer than about four seconds involve location 161. POKE 161,0:POKE 162,0:WAIT 162,2 pauses for 2*256/60 = about eight seconds.

The easiest way to implement a pause for an indefinite period of time is to intercept the normal IRQ interrupt routine and check for a keypress. The SHIFT key is useful, because SHIFT-LOCK can pause indefinitely. However, any SHIFTed entry will then temporarily stop the program. The following ML will do the trick with normal keys (for example, the left arrow key); it is easily modifiable to check the keyboard twice, so the key needn't be held down, or to test for Commodore, SHIFT, or CTRL keys. To use the pause routine, change the interrupt vector at locations 788/789 ($0315/0315) to point at the start of this routine:

```
PAUSE JSR  $FF9F   ; Scan keyboard
      LDA  $C5     ; Look at keypress
      CMP  #$08    ;This is left arrow
      BEQ  PAUSE   ; Pause while pressed
      JMP  $EABF   ; Continue interrupt
```

POP

POP discards a RETURN from the stack and erases the effect of the previous GOSUB, so that if RETURN is encountered, the address returned to will be that of the previous GOSUB (or, alternatively, the error ?RETURN WITHOUT GOSUB will be signaled). This is useful in providing an escape from subroutines, although it's quite difficult to explain why. To give just one example, a game may contain a long loop to redraw the screen and move a player. End-of-game may be tested by a subroutine; if RETURN isn't then necessary, redundant RETURN addresses can build up. Typically, you'll get an otherwise inexplicable ?OUT OF MEMORY message after 24 or so games, as stack space runs out. You can cure this problem and others like it with Program 6-28.

Program 6-28. POP

```
10 DATA 104,104,169,255,133,74,32,138,195,201,141
20 DATA 240,3,76,224,200,232,232,232,232,232,154,9
   6
30 FOR J=828 TO 850:READX:POKEJ,X:NEXT
```

From the viewpoint of structured programming, this command is unnecessary (and even harmful) since such programming demands that subroutines have a single entry and exit without irregular exits by GOTO or POP.

This program is relocatable (as written, it starts at 828) and called by a simple SYS from within a program (SYS 828). RUN and test with SYS 828; you should get ?RETURN WITHOUT GOSUB.

Another type of POP, using a part of CLR, is more thorough, clearing away all loops and subroutines within a program by resetting the stack pointer and deleting all evidence of FOR-NEXT and GOSUB. Variable values, DATA pointers, and so on are retained. On an abort or escape, this routine would cut through any tangle of loops and subroutines. With VIC, machine language is necessary; the following routine shows how it's done.

```
PLA          ; Remove SYS address
PLA
JMP   $C67E  ; Enter CLR to reset the stack
```

Program 6-29 shows the same thing in decimal.

Program 6-29. Super POP

```
10 DATA 104,104,76,126,198
20 FOR J=828 TO 832:READX:POKE J,X:NEXT:REM SYS 82
   8 IS SUPER-POP
```

PRINT @

This command moves the cursor rapidly anywhere on the screen, as specified by horizontal and vertical parameters (HTAB and VTAB is another formulation of this idea). Graphics in BASIC can often be much improved with one of these methods, in place of printing {HOME} and many cursor downs, lefts, and rights. The fastest versions need ML, and therefore space to be stored; less speedy versions use ROM routines, and are more convenient but slightly slower.

Fast ML version. Using Program 6-30, SYS 828,H,V takes in horizontal (H) and vertical (V) parameters and puts them into the Kernal routine at $FFF0.

Program 6-30. PRINT @

```
Ø DATA 32,155,215,138,72,32,155,215,1Ø4,17Ø,164,1Ø
  1,24,76,24Ø,255
1Ø FOR J=828 TO 843:READX:POKEJ,X:NEXT
```

To do the same thing in BASIC, POKE 781,V:POKE 782,H:POKE 783,0:SYS 65520:PRINT "HELLO!"

Another approach, valuable when the screen size has been changed (and therefore conflicts with the assumption that the screen has 22 columns and 23 rows) is given in Program 6-31.

Program 6-31. PRINT@ for Reformatted Screens

```
1ØØØØ S=768Ø: REM SCREEN START AT $1EØØ; $1ØØØ IF
      {SPACE}8K OR 16K ADDED
1ØØ1Ø SC=S+24*Y+X: REM EXAMPLE APPLIES TO 24 COLUM
      N SCREEN
1ØØ2Ø POKE2Ø9,SCAND255: REM POKE POINTERS TO RELEV
      ANT X,Y POSITION
1ØØ3Ø POKE21Ø,SC/255: REM IN SCREEN MEMORY
1ØØ4Ø PRINT "HELLO!";
```

Line 10010 calculates the position in screen RAM of horizontal and vertical parameters X and Y (assumed to start at 0), on the assumption of 24 columns. This could just as well be 25 or 10 columns. Chapter 12 has many examples of the use of subroutines like this, some of which use different pointers (notably 211 and 214) but with the same object. Such subroutines are invaluable with full-size screen displays.

PRINT USING

In business systems it's customary to specify the format in which numbers and characters are printed. Program 6-32 is a valuable routine which provides an easy way to output numeric data in a variety of formats (for example, rounded to two decimal places or with leading asterisks). The overall lengths, including leading characters, are selectable, so lining up decimals in columns is made very simple. Output can be directed to a printer if hard copy is wanted.

Program 6-32. PRINT USING

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
Ø DATA 1,8,2,32,162,Ø,221,Ø,1,24Ø,6,232,224,12,2Ø8
  ,246,24,96,169,69,32,-162                  :rem 72
1 DATA 176,9Ø,173,-166,24Ø,94,173,2,1,2Ø8,11,172,-
  165,169,48,153,2,1,136                    :rem 2Ø9
```

```
2 DATA 2Ø8,25Ø,169,46,32,-162,168,144,2,16Ø,48,169
  ,Ø,32,-162,152,157,Ø,1,169                 :rem 148
3 DATA 46,32,-162,172,-164,232,136,2Ø8,252,236,-16
  5,176,33,172,-165,169,Ø                      :rem 3
4 DATA 153,1,1,189,Ø,1,2Ø1,32,2Ø8,3,169,32,234,153
  ,Ø,1,2Ø2,16,6,173,-163,136                 :rem 114
5 DATA 16,244,136,16,231,169,Ø,133,97,16Ø,1,132,98
  ,96,169,Ø,32,-162,144,24Ø                  :rem 1Ø6
6 DATA 138,168,173,2,1,24Ø,9,169,46,32,-162,144,2,
  138,168,152,17Ø,2Ø2,16,181                 :rem 156
7 DATA Ø,32,158,2Ø5,32,221,221,32,-148,32,3Ø,2Ø3,9
  6                                          :rem 181
1Ø PRINT"{CLR}{RVS}VIC PRINT USING"          :rem 2Ø4
2Ø T=PEEK(55)+256*PEEK(56):REM T= TOP OF BASIC
                                              :rem 63
3Ø L=T-166:REM PROGRAM IS 166 BYTES IN LENGTH
                                              :rem 89
4Ø FOR J=LTOT-1:REM PLACE ROUTINE IN TOP OF AVAILA
   BLE RAM                                   :rem 156
5Ø READ X%:IF X%<ØTHENY=X%+T:X%=Y/256:Z=Y-X%*256:P
   OKEJ,Z:J=J+1                              :rem 197
6Ø POKE J,X%                                 :rem 136
7Ø NEXT                                      :rem 166
1ØØ X%=L/256:Z=L-X%*256:REM WILL BE HI & LO BYTES
    {SPACE}OF NEW TOP OF MEMORY              :rem 135
11Ø POKE 55,Z:POKE53,Z:POKE51,Z:REM SET NEW MEMORY
     TOP                                      :rem 29
12Ø POKE56,X%:POKE54,X%:POKE52,X%:REM WITHOUT USIN
    G CLR                                    :rem 131
125 REM**PRINT OUT INSTRUCTIONS AND ADDRESSES, ONT
    O SCREEN**                               :rem 251
13Ø PRINT "{DOWN}SYS (";L+153;")FOLLOWED BY ANY NU
    MERIC                                     :rem 96
132 PRINT"EXPRESSION PRINTS THE FORMATTED VALUE.":
    PRINT                                     :rem 8Ø
14Ø PRINT L "=DEC/INT FLAG"                  :rem 239
15Ø PRINT L+1 "=OUTPUT LENGTH"               :rem 255
16Ø PRINT L+2 "=DEC. PLACES"                   :rem Ø
17Ø PRINT L+3"=LEADING CHARS"                :rem 181
18Ø PRINT L+98"=+VE LEAD CHAR"               :rem 137
19Ø PRINT "{DOWN}SAVE FROM" L "TO" T-1;      :rem 255
2ØØ PRINT"($";:GOSUB5ØØ: PRINT" TO $";: L=T-1:GOSU
    B 5ØØ:PRINT")"                            :rem 62
21Ø PRINT"{DOWN}SET UP WITH LENGTH 9,2 DEC. PLACES
    , LEADING SPACES.                         :rem 56
25Ø END                                      :rem 11Ø
499 REM***ONE LINE DECIMAL TO HEX CONVERTER***
                                             :rem 191
5ØØ L=L/4Ø96:FORJ=1TO4:L%=L:L$=CHR$(48+L%-(L%>9)*7
    ):PRINTL$;:L=16*(L-L%):NEXT:RETURN       :rem 236
```

Enter this program, save it, and run it. You should get a screen display similar to that shown in Figure 6-11. The figures in the resulting screen depend on VIC's memory, because the ML is stored in the highest available RAM (where it is protected from corruption by BASIC strings). Note that the relocating loader puts 166 bytes into memory, while there are actually fewer bytes than this in the DATA statements. This is not a mistake; each negative value corresponds to an address to be relocated and contributes two bytes, not one.

The unexpanded VIC should give the display shown in Figure 6-11.

Figure 6-11. PRINT USING Display

```
SYS ( 7667 ) FOLLOWED
BY ANY NUMERIC
EXPRESSION PRINTS THE
FORMATTED VALUE.

  7514 = DEC/INT FLAG
  7515 = OUTPUT LENGTH
  7516 = DEC. PLACES
  7517 = LEADING CHARS
  7512 = + VE LEAD CHAR

SAVE FROM 7514 TO 7679
  ($1D5A TO $1DFF)

SET UP WITH LENGTH 9,
2 DEC. PLACES, LEADING
SPACES.
```

Using this routine is less complicated than it might appear. SYS 7667 (X), for example, will print the current value of X, formatted in accordance with the contents of the locations listed below.

Decimal/integer flag. A value of 0 in this location means the result will be treated as an integer (no decimal point symbol will be printed) while 1 means it is decimal.

Output length. This location specifies the total length of the output string −1. It allows tables of numbers to be constructed easily.

Decimal places. This controls the number of figures after the decimal point. If the number is an integer, it is ignored.

Leading characters. This location holds the ASCII character printed before the number begins. This enables printing in formats like ****100 or 000123,23. The usual leading character is the space character (32).

Positive symbol. Numerals are preceded by a space or minus sign with unmodified PRINT; this routine permits a substitute for space to be printed (for example $) so all positive numbers will appear preceded by $.

Note that X is truncated, not rounded off. If you wish to round to two decimal places, use SYS 7667 (X + .005).

Demonstration program. Program 6-33 demonstrates an application of PRINT USING by printing formatted columns of figures. Lines 20, 30/31, and 40 print the first, second, and third columns. It assumes an unexpanded VIC and uses meaningful variable names (insofar as reserved words permit) to make the POKEs more comprehensible.

Program 6-33. PRINT USING Demonstration

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
Ø PRNT=7667:SWITCH=7514:LNGTH=7515:DECPTS=7516:CHA
  R=7517:LDGCHAR=7612                       :rem 29
1Ø FORJ=-1ØTO1ØØ STEP 1Ø:PRINT             :rem 245
2Ø POKE SWITCH,Ø:POKE LNGTH,4:POKE CHAR,42: POKE L
   DGCHAR,42:SYS (PR) J                    :rem 132
3Ø POKE SWITCH,1:POKE LNGTH,7:POKE CHAR,32: POKE L
   DGCHAR,32                                :rem 17
31 POKE DECPTS,4: SYS (PR) 1/(1+J)          :rem 57
4Ø POKE LDGCHAR,ASC("$"): POKE DECPTS,2:SYS (PR) 1
   ØØ+J                                      :rem Ø
5Ø NEXT                                    :rem 164
```

The core of this routine is as follows:

JSR $CD9E	; Input and evaluate a BASIC numeric expression
JSR $DDDD	; Convert contents of FP accumulator 1 into a string
JSR $xxxx	; Special routine (address varies) to process $0100–$010C
JSR $CB1E	; Print the string using A (low), Y (high) pointers
RTS	; Return to BASIC without any other action.

The idea is to print exactly as normal, except that the number, after being prepared for printing as a string, is edited. The ML routines which edit are located earlier in memory, and the table of values is at the start. The first four values in the DATA statements, in fact, set the decimal mode, the length, the number of decimal points, and the leading character (which is a space).

When the relocating loader runs, the start and end addresses of the routine are printed. Thus the routine can be loaded directly at any time in the future without having to be relocated in memory. Remember to lower the pointers each time any ML routine coexisting in BASIC space is newly loaded, or string calculations will corrupt it.

Reconfiguring BASIC Memory

Chapter 5 explained how BASIC configures itself on switch-on, to allow for any expansion RAM which may be present. The position of a BASIC program in memory (as defined by pointers to its start and end) and the positions of the screen and character generator are set. Any pure BASIC program will run, provided there's enough RAM. But machine language, graphics requiring a fixed position in memory, or PEEKs and POKEs may fail to work.

If you're in the position of having software which runs with some but not other memory configurations, try Program 6-34. The most common problem (apart from simply not having enough RAM) is that a program designed to run on 3K added memory won't run with an 8K or 16K expander because the screen moves. Option 5 of Program 6-34 may help.

Program 6-34. Reconfiguring VIC's Memory Without Removing RAM or ROM Packs

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
0 REM REARRANGE POINTERS IN MEMORY AND MIMIC VARIO
  US BASIC SETUPS                             :rem 64
10 PRINT "{CLR}CONFIGURE MEMORY AS:-{DOWN} :rem 42
20 PRINT "1 UNEXPANDED VIC                    :rem 16
30 PRINT "2 VIC + 3K RAM PACK                :rem 206
40 PRINT "3 VIC + 8K RAM PACK                :rem 213
50 PRINT "4 VIC + 16K RAM PACK                 :rem 6
60 PRINT "5 VIC+8K & 3K PROGRAM              :rem 150
70 PRINT "6 VIC+3K & 8K PROGRAM              :rem 152
80 PRINT "7 DISCONNECT ROM PACK               :rem 69
90 PRINT "8 ANY OTHER{DOWN}                  :rem 203
92 PRINT "PRESS 1 TO 8"                      :rem 213
94 GET V: IFV=0 GOTO 94: REM SELECT OPTION 1-8.
                                              :rem 80
96 ON V GOTO100,200,300,400,500,600,700,800
                                             :rem 115
100 S=4096: E=7680: SC=7680: GOTO 1000: REM SET ST
    ART OF BASIC,                             :rem 86
200 S=1024: E=7680: SC=7680: GOTO 1000: REM END OF
     BASIC, AND                              :rem 123
300 S= 4608: E=16384:SC=4096: GOTO 1000: REM START
     OF SCREEN FOR                           :rem 179
400 S=4608: E=24576: SC=4096: GOTO 1000: REM EACH
    {SPACE}COMBINATION.                       :rem 94
500 S=8192: E=16384: SC=7680: GOTO 1000    :rem 209
600 S=1024: E=4096: SC=4096: GOTO 1000     :rem 144
700 POKE 783,181: SYS 64815: REM DISCONNECT $A000
    {SPACE}ROM                                :rem 90
800 INPUT " START OF BASIC";S               :rem 127
802 INPUT "{3 SPACES}END OF BASIC";E        :rem 188
804 INPUT "START OF SCREEN";SC               :rem 36
1000 POKE641,S-INT(S/256)*256: REM PUT LOW AND HIG
     H                                        :rem 85
1010 POKE642,S/256: REM BYTES OF START AND:rem 115
1020 POKE 643,E-INT(E/256)*256: REM END OF BASIC.
                                              :rem 91
1030 POKE644,E/256                           :rem 206
1040 POKE648,SC/256: REM HIGH BYTE OF SCREEN
                                             :rem 235
1050 SYS 64818                               :rem 208
```

If you have some standard application, you can simply pick out the relevant parts of the program. For example, if you use a ROM assembler or other utility (such as the *Super Expander*), only lines 700 and after are needed.

The program allows unexpanded VIC programs to run on VIC with 3K, 8K, or 16K expansion; it allows VIC with several RAM packs in an expansion board to be reconfigured as though only one were present.

Option 5 puts the screen in the 3K expansion position, so with 8K or 16K expansion a 3K program using POKEs to the screen can work. Also, unexpanded programs using screen POKEs, which have too many REMs in their fully commented forms, can be run in this way. Option 6, with 3K expansion, puts the screen in the 8K position, so programs developed with 8K may run (but they'll have to be short).

Note that programs can call the reset routine from within themselves, and if they do this your reconfiguration will be reset. There may be no choice but to use the correct memory packs.

Reconfiguring without resetting BASIC. The program listed above works in a completely general way, but it resets BASIC as though the VIC were just switched on. When can you reconfigure without losing the program? Lowering the top of BASIC's space (or raising it again) can be done within BASIC without any problems. Moving the bottom of BASIC is more difficult, and the easiest solution is to use a loader or boot program.

Lowering top-of-BASIC with an unexpanded VIC. The unexpanded VIC's pointers (at 43 and 44, and 55 and 56) show that BASIC occupies $1000 to $1E00. To lower the top of memory available to BASIC to $1800, pointers 55 and 56 must have high byte 24 ($18) and low byte 0. POKE 55,0:POKE 56,24 sets the upper limit. CLR will reset all the string pointers but lose the variables. POKE 51,0:POKE 52,24:POKE 53,0:POKE 54,24:POKE 55,0:POKE 56,24 has the same effect but does not lose variables' values. If only 55 or 56 is changed, the first string in memory will position itself above the new top of BASIC; subsequent strings will be below.

The graphics chapter contains many such examples, which set the VIC chip to coincide with BASIC. For example, POKE 36879,254 assigns the area starting at $1800 to character definitions.

Reconfiguring to move BASIC without resetting the VIC chip. Program 6-35 is an alternative form of RECONFIGURE; rather than call a reset routine, it alters pointers so the screen color and size are retained.

Program 6-35. Reconfiguring VIC Memory by Altering Pointers

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 POKE 43,STARTLO:POKE44,START HI: REM DESIRED ST
   ART OF BASIC+1                          :rem 103
20 POKE55,ENDLO : POKE56, ENDHI : REM DESIRED END
   {SPACE}OF BASIC+1                        :rem 73
30 POKE STARTLO+256*STARTHI-1,0: REM ZERO BYTE MUS
   T START BASIC                            :rem 81
40 POKE 648,SCREENHI: REM SCREEN START     :rem 30
50 POKE36866,ROWS+128: REM OMIT 128 WITH LARGE SCR
   EEN                                      :rem 55
```

```
60 POKE36869,VALUE FROM 192 TO 255: REM SET CHARAC
   TER GENERATOR POSN                        :rem 219
70 NEW : REM MAKES BASIC SELF-CONSISTENT    :rem 163
```

Notes on BASIC's zero byte. BASIC must have a zero byte at the position indicated by 43 and 44. If it hasn't, NEW or RUN will give ?SYNTAX ERROR. *VICMON* alters 43, so POKE 43,1 may be necessary after its use.

For example, to make BASIC start at $1200 and end at $1400, with the screen at $1600 and color RAM therefore at $9600, you should POKE 43,1: POKE 44,18: POKE 55,0: POKE 56,20: POKE 256*18,0: POKE 648,22: POKE 36866,150: POKE 36869,210: NEW. At that point, PRINT FRE(0) shows 509 free bytes, and POKE 5632,6: POKE 38400,5 prints a green F at the top left of the screen, showing that the POKEs have worked.

Loaders, or boot or bootstrap programs. So far, you have seen how to design any feasible BASIC arrangement you like. However, if the start of BASIC is moved during reconfiguration, the reconfigure program is lost. How can you load and run another program? The keyboard buffer, or input buffer, offers a solution. (Alternatively, commands can be printed to the screen and the keyboard queue can be used to input them; this, however, assumes that the position of the screen doesn't change.)

For tape booting, add line 65 POKE 631,131:POKE 198,1 to Program 6-36 below. These POKEs have the effect of typing SHIFT-RUN; the ASCII value is 131. Location 198 holds the number of characters in the keyboard queue, and 1 simulates a single key. Now the program reconfigures BASIC, loads the next tape program, and runs it.

For disk booting, a different approach is required. The keyboard queue can't easily hold more than ten characters, which is not enough to load a disk program since a name is usually needed. LOAD"*",8:RUN (which, in its short form, is L SHIFT-O "*",8:R SHIFT-U) just fits. A solution is to use the input buffer, by adding the lines in Program 6-36.

Program 6-36. Booting Disks

```
61 CLR : REM NEW NOT NEEDED AT END(AS NEW PROGRAM
   {SPACE}IS TO BE LOADED)
62 N$="LOAD" +CHR$(34) + "HELLO"+CHR$(34)+ ",8"+CH
   R$(0)
63 FORJ=1TOLEN(N$):POKE511+J,ASC (MID$(N$,J)):NEXT
64 POKE198,3:POKE631,82:POKE632,213:POKE633,13
65 POKE781,255:POKE782,1: REM POINTER TO $01FF
66 SYS 50310: REM INPUT LINE
```

Line 62 sets up a string ending with a null byte; this exactly mimics a line input from the keyboard, when line 63 POKEs it to the input buffer at 512 ($0200). Add PRINT N$:END if you're not sure of the quotes. Line 64 puts R SHIFT-U RETURN in the keyboard buffer, to enter RUN after LOAD. Lines 65 and 66 process the line in the buffer, loading the program called "HELLO."

REM

REM is part of VIC's normal set of commands, but it deserves a place here because of the unique status of REM statements outside the normal strict rules of BASIC syntax.

REM with SHIFT and quotes. SHIFTed characters have their high bit set and are interpreted as tokens, so LIST converts these into reserved words, expanding the line. Cursor control characters (HOME, etc.) can be inserted after an opening quote. So can DElete, by opening up space inside quotes with the Insert key. A hidden line can be created by following it by :REM"", expanding the space in quotes, and filling with DELetes, though this maneuver won't hide the line when it's listed on a printer. REM stores some characters (for example, *, which is a reserved word) differently inside quotes than outside. Thus, utilities which search for strings may not find them in REM statements.

Inserting characters into REMs. REM is tokenized as 143 in decimal. Program 6-37 puts two RETURN characters immediately after REM in a REM line, and also immediately before the end of the REM line, so (for example) 100 REM** REMINDER COMMENTS * will list remarks neatly onto new lines.

Program 6-37. Inserting Characters into REMs

```
63000 L=43
63010 L=PEEK(L)+256*PEEK(L+1):IFL=0THENEND:REM SKI
      P THRU LINKS.
63020 IF PEEK(L+4)<>143 GOTO63010: REM IF REM FOUN
      D, THEN:-
63030 POKEL+5,13:POKEL+6,13: REM POKE 2 RETURNS

63040 FOR J=L+5 TO 9E9: IF PEEK(J)>0THEN NEXT: REM
       FIND END-OF-LINE,
63050 POKEJ-1,13: GOTO 63010: REM AND POKE 1 RETUR
      N.
```

Other characters might include printer control characters (to give bold or reversed printing of REMs) or color characters (to list REMs in a different color on the screen).

REMs to store ML. BASIC can hold ML routines or DATA as REM statements. All that needs to be done is to POKE in the relevant bytes. However, there are three potential problems with this technique:

Zeros should not be used because they will be treated as end-of-line markers if the program is edited, thus corrupting the ML by inserting a spurious link address and line number. This behavior could be used, with care, as a security device. Generally, instead of LDX #0, you should use LDX #1/DEX.

The actual position in memory has to be known. It is easiest to put a REM statement at the very start of a program, so the sixth byte from the initial zero byte is the start position.

BASIC containing ML in REM must load back to the same RAM area (unless it relocates), and the SYS call takes account of the starting location. Allowing for memory expansion thus adds a slight amount of work.

RENUMBER

Renumbering, or resequencing, a BASIC program has some cosmetic advantages and is valuable where BASIC line numbers are too close to allow more BASIC to be added (although BASIC numbered 0,1,2,..., runs slightly faster than it would otherwise). Program 6-38 is a short BASIC subroutine that changes line numbers only, between a selected range, by POKEing in new values.

Program 6-38. RENUMBER

```
62000 A=1025:B=256:PRINT"LO/HI LINES, START & INCR
      EMENT": INPUT L,H,S,I
62010 FOR R=0 TO 9E9: IF PEEK(A+2)+B*PEEK(A+3)<L T
      HENA=PEEK(A)+B*PEEK(A+1):NEXT
62020 FOR R=0TO 9E9:X=S+R*I:IF A=0 OR PEEK(A+2)+B*
      PEEK(A+3)>H THEN END
62030 POKE A+3,X/B:POKE A+2,X-(INT(X/B)*B):A=PEEK(
      A)+B*PEEK(A+1):NEXT
```

A good renumbering utility should obviously change addresses (such as GOTO or GOSUB) within BASIC, since otherwise the program will no longer run. However, this is more difficult to program than you might think. BASIC holds line numbers as ASCII strings, so these may have to be changed in length (unlike the two-byte line number storage system) to accommodate the renumbered address. At least two passes are necessary, one to store current line numbers and one to change references.

Another desirable feature is to have four parameters, so that a range of lines can be numbered. The *Programmer's Aid* cartridge does not have this feature, so that the whole program must be renumbered and the numbering of standard subroutines cannot be retained.

Generally, renumbers are prone to various difficulties. All line numbers after IF-THEN, GOTO, GO TO, GOSUB, ON-GOTO, and ON-GOSUB must be located and changed. RUN and LIST can be followed by line numbers, too, and are often included in renumbers. The renumbering may cause lines to have impossibly high numbers, or, with four parameter utilities, to coincide with or overlap already existing numbers. References to nonexistent lines should be signaled as errors (for example, GOTO 100 where there was no line 100). Conceivably, graphics could be upset too since the whole program changes length.

SEARCH and REPLACE

Searching BASIC is reasonably straightforward, given an understanding of the way it is stored. Program 6-39, a VIC search routine, is an example; it is an ML search which hunts for a match with the contents of the first line of the BASIC program currently in memory. To use the program, load and run Program 6-39, then load the program you wish to search and add as the first line the characters for which you wish to search. Start the search with SYS 828. For instance, with 0 XX as the first line, SYS 828 prints all lines containing the variable XX. The ML relocates and can be stored elsewhere than in the tape buffer.

Program 6-39. VIC Search Routine

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
0 DATA 166,43,165,44,134,251,133,252,160,1,134,253
  ,133,254,177,253,240,73,72,136,177        :rem 41
1 DATA 253,72,160,4,132,142,132,143,177,251,201,34
  ,208,2,230,143,164,143,177,251,240        :rem 14
2 DATA 28,72,164,142,177,253,240,15,104,209,253,24
  0,6,230,142,160,4,208,222,230,142        :rem 223
3 DATA 208,226,104,104,170,104,208,193,160,2,177,2
  53,170,200,177,253,32,205,221,169        :rem 233
4 DATA 32,32,210,255,201,0,208,231,96        :rem 9
10 FOR J=828 TO 919: READ X: POKE J,X: NEXT:rem 21
```

Program 6-40, which lets you both search and replace, works along the same lines. It is a rather simple relocatable example, which ML programmers might like to examine; it changes single bytes, not strings, so that all X's could be changed to Y's, or all PRINTs to PRINT#s.

Program 6-40. VIC Search and Replace

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
0 DATA 165,43,133,252,198,252,165,44,133,253,169,0
  ,133,254,168,24,165                        :rem 80
1 DATA 252,105,4,133,252,165,253,105,0,133,253,230
  ,252,208,2,230,253                        :rem 244
2 DATA 177,252,240,26,201,34,208,6,165,254,73,255,
  133,254,165,254,240                        :rem 72
3 DATA 232,177,252,201,65,208,226,169,66,145,252,2
  08,220,200,200,177                         :rem 18
4 DATA 252,208,196,96                        :rem 20
10 FOR J=320 TO 390: READ X:POKE J,X:NEXT   :rem 1
```

You control the character searched for and the replacement character by changing the contents of locations 375 and 379 after the program has been loaded. Location 375 holds the value of the character to be searched for, and location 379 holds the character with which the search character will be replaced. This search and replace program has three modes, controlled by POKEing locations 370 and 371. To change only those characters which appear within quotes, POKE locations 370 and 371 with values of 240 and 232, respectively; to change only those occurrences of the specified character outside quotes, POKE the values 208 and 232 into those locations. To change both, POKE 234 and 234.

For example, suppose you want to change all occurrences of the variable Y (ASCII 89) to Z (ASCII 90). You wouldn't want to make the replacement where Y appears in quotes—you don't want to change things like PRINT"YES OR NO" to PRINT"ZES OR NO". First, load and run Program 6-40, then load the program to be modified, then type:

POKE 375,89: POKE 379,90: POKE 370,208: POKE 371,232

Start the search-and-replace process with SYS 320.

If the ML is put into an area other than locations 320–390, of course, these addresses will change.

SET

SET (and UNSET or RESET) are graphics commands in some BASICs which allow a point or small square to be drawn at any specified position on the screen. Chapter 12 contains an extensive discussion of this, including a high-resolution plotting routine and a space-saving routine to plot battenberg cake style quarter-size dots which provide 44 × 46 resolution. The same principle can be followed with larger screen displays. VIC's limitations on color mean that all four quadrants cannot be independently colored, even with 256 user-defined characters, including all combinations of character/background/border/auxiliary colors.

SORT

Sorting, in computer jargon, means arranging a list in order, usually alphabetically or numerically. Many types of sorts exist, but to save space only three will be discussed here: a machine language sort, which includes a demonstration to illustrate the syntax, and two BASIC sorts.

Machine language. The ML version is far faster than BASIC. Program 6-41 puts it securely at the top of BASIC memory, although it is relocatable and can be put anywhere in free RAM. It sorts string arrays in ascending order, using an ordering algorithm identical to the VIC-20's, and it is initiated using a simple SYS call. It lets you sort strings from the second, third, or any other character, and it works with any memory configuration.

Program 6-41. Machine Language Sort for VIC String Arrays

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
Ø DATA 32,115,Ø,133,97,169,128,133,98,32,115,Ø,24Ø
  ,7,9,128,133,98,32,115                    :rem 213
1 DATA Ø,165,47,133,99,165,48,133,1ØØ,16Ø,Ø,165,97
  ,2Ø9,99,2Ø8,7,2ØØ,165,98                   :rem 79
2 DATA 2Ø9,99,24Ø,2Ø,24,16Ø,2,177,99,1Ø1,99,72,2ØØ
  ,177,99,1Ø1,1ØØ,133                        :rem 71
3 DATA 1ØØ,1Ø4,133,99,144,221,16Ø,5,177,99,133,1Ø2
  ,2ØØ,177,99,133,1Ø1,2Ø8                     :rem 3
4 DATA 2,198,1Ø2,198,1Ø1,24,165,99,1Ø5,7,133,99,16
  5,1ØØ,1Ø5,Ø,133,1ØØ,165,1Ø1               :rem 192
5 DATA 2Ø8,2,198,1Ø2,198,1Ø1,2Ø8,4,165,1Ø2,24Ø,18,
  133,1Ø5,162,Ø,134,1Ø3,134                  :rem 82
6 DATA 1Ø4,165,99,133,1Ø6,165,1ØØ,133,1Ø7,24Ø,224,
  24Ø,114,24,165,1Ø6,1Ø5                    :rem 198
7 DATA 3,133,1Ø6,165,1Ø7,1Ø5,Ø,133,1Ø7,23Ø,1Ø3,2Ø8
  ,2,23Ø,1Ø4,16Ø,2,177,1Ø6                   :rem 17
8 DATA 153,1Ø9,Ø,136,16,248,16Ø,5,177,1Ø6,153,1Ø9,
  Ø,136,192,2,2Ø8,246,17Ø                     :rem 7
9 DATA 56,229,1Ø9,144,2,166,1Ø9,16Ø,255,232,2ØØ,2Ø
  2,2Ø8,8,165,112,197,1Ø9                    :rem 16
```

```
10 DATA 144,10,176,34,177,113,209,110,240,238,16,2
   6,160,2,185,112,0,145                    :rem 142
11 DATA 106,136,16,248,160,5,185,106,0,145,106,136
   ,192,2,208,246,169,0,133                  :rem 49
12 DATA 105,165,101,197,103,208,152,165,102,197,10
   4,208,146,165,105,240,138,96               :rem 1
18 REM ****** FIRST LOWER MEMORY BY 256 BYTES...
                                            :rem 124
19 REM ****** ... THEN POKE IN RELOCATING CODE, AN
   D GIVE ITS ENTRY ADDRESS                 :rem 126
20 POKE 56, PEEK(56)-1: CLR                 :rem 130
30 T=PEEK(55) + 256*PEEK(56)                :rem 167
100 FOR J=T TO T+242: READ X: POKE J,X: NEXT
                                            :rem 107
110 PRINT "USE SYS"T":X TO SORT ARRAY X$(), FOR EX
    AMPLE:-"                                 :rem 54
1000 INPUT "SIZE OF ARRAY";N                :rem 109
1010 DIM XY$(N)                              :rem 16
1020 FOR J=0 TO N: XY$(J)=LEFT$(STR$(RND(1)*100),6
     ): NEXT                                 :rem 67
1030 PRINT "SORTING..."                      :rem 69
1040 ML=PEEK(55) +256*PEEK(56)               :rem 78
1050 SYS ML:XY                               :rem 73
1060 FOR J=0 TO N: PRINT XY$(J): NEXT         :rem 6
```

This is a version of the Bubble Sort, which operates on the pointers of string arrays and produces no garbage collection delays. It operates in direct or program modes; to save space it doesn't include a validation routine, so don't try to sort an array that does not exist.

Speed is maximized if new items are added at the beginning of an array before sorting. The zeroth element isn't sorted; it can hold a title if desired. If the 255 in line 9 is changed to 1, strings are sorted from the second position; if it becomes 2, from the third; and so on.

Provided spaces pad out the strings correctly, it's possible to re-sort an array in different ways. For an example, see the disk drive sorting program, which sorts on the initial of each program or file.

Strings are sorted in ASCII order. This can produce apparent anomalies: 12.3 comes before 2.87, which comes before 29.67. HELLO! precedes HELLO; and strings 0 to 25 emerge as 0, 1, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 2, 20, 21, 22, 23, 24, 25, 3, 4, 5, 6, 7, 8, 9. Computer sorting always produces effects like these, but they should not pose too much of a problem in practice. In fact, programming can often be simplified by careful choice of the way in which items to be sorted are arranged. For instance, a date held as YYMMDD automatically sorts into the correct order. Similarly, the fact that the comma has a lower ASCII value than any letter insures that names held with commas sort correctly. Williams,P. will come before Williamson,A.

Lines 1000–1060 in Program 6-41 provide a demonstration of the sort. Lines 1000 and 1010 establish array XY$, and line 1020 fills the array with random numeric characters. Line 1040 calculates the address at the top of free RAM where the machine language sorting routine is stored, and line 1050 shows the proper syntax for initiating the ML sort. Note that you specify XY to sort array XY$; the $ is not used. After the array is sorted, line 1060 prints the results. If you wish to add this sorting routine to your own programs, lines 1000–1060 should be omitted.

BASIC sorts. The Shell-Metzner Sort is a fast sort, which is also quite easy and trouble-free to program. The version given in Program 6-42 sorts items 1 to N of an array dimensioned with A$(N). The sort is written as a subroutine to be added to your programs. It assumes that array A$ and number of elements N have both been established before you GOSUB to the routine. Upon return from the routine, the contents of array A$ will be arranged in ascending order.

Program 6-42. Shell-Metzner Sort

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
59000 REM SHELL-METZNER SORT                  :rem 244
59005 M=N                                     :rem 219
59010 M=INT(M/2):IFM=0THENRETURN: REM SORT COMPLET
      ED                                        :rem 8
59020 J=1:K=N-M                                :rem 66
59030 I=J                                     :rem 209
59040 L=I+M                                    :rem 76
59050 IF A$(I)>A$(L)THENTE$=A$(I):A$(I)=A$(L):A$(L
      )=TE$:I=I-M:IF I>0THEN59040             :rem 219
59060 J=J+1:IFJ>KGOTO59010                      :rem 5
59070 GOTO59030                                :rem 63
```

The Tournament Sort, so called because it pairs together items for comparison, starts to give answers almost immediately, rather than waiting for the entire array to be sorted. In addition, since numbers rather than strings are moved, garbage collection time (which can otherwise be horrendous with BASIC) is nil.

Program 6-43 illustrates the Tournament Sort. Lines 10 and 20 allow you to set up the array N$, which will be sorted. A numeric array I is also required, and it must be dimensioned for twice as many elements as N$. Lines 200–330 perform the sort, printing each element as it is sorted into its proper position and ending when the sort is complete.

Program 6-43. Tournament Sort

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 INPUT "SORT HOW MANY ITEMS";N:B=N-1:DIM N$(B),I
   (2*B)                                      :rem 141
20 FOR J=0TOB:INPUT N$(J):NEXT:REM SETS UP DEMONST
   RATION DATA                                 :rem 56
200 X=0:FORJ=0TOB:I(J)=J:NEXT: REM INDEX ARRAY SET
     UP WITH O,1,2,3,...                       :rem 51
210 FOR J=0 TO 2*N-3STEP2:B=B+1:REM ORDERS INDEX A
    RRAY IN PAIRS                             :rem 157
220 I(B)=I(J): IF N$(I(J+1))<N$(I(J)) THEN I(B)=I(
    J+1)                                      :rem 200
```

```
230 NEXT                                     :rem 212
250 X=X-1:C=I(B):IFC<0THEN END: REM SORT FINISHED
                                             :rem 166
260 PRINT N$(C) " ";: REM PRINT ONE SORTED ITEM OF
     DATA                                    :rem 230
270 I(C)=X: REM SORT LOOP IS HERE            :rem 107
280 J=2*INT(C/2):C=INT(C/2)+N:IF C>B GOTO 250
                                             :rem 204
300 IF I(J)<0 THEN I(C)=I(J+1):GOTO280         :rem 8
310 IF I(J+1)<0THEN I(C)=I(J): GOTO 280        :rem 9
320 I(C)=I(J): IF N$(I(J+1))<N$(I(J)) THEN I(C)=I(
    J+1)                                     :rem 203
330 GOTO 280                                 :rem 105
```

TRACE with SINGLE STEP

Program 6-44 presents a version of TRACE to help you understand the workings of BASIC. The whole current BASIC line is displayed at the screen top. The f1 key toggles the trace from on to off and vice versa, while f3 changes the speed of TRACE by accepting a number from 0–9. The program executes a true single-step whenever f5 is pressed, and f7 traces as fast as possible through BASIC.

The machine language program starts at $3000 (12288) and therefore assumes that an 8K or 16K memory expander is present. This is partly because the program occupies about 600 bytes. It works for any screen position and can easily be made to operate with programs for the unexpanded VIC. Program 6-44 erases itself after loading the machine language, so be sure to save a copy before you try to run the program.

Program 6-44. TRACE for the VIC

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
0 DATA 169,76,133,132,169,19,133,133,169,48,133,13
  4,96,255,0,254,15                          :rem 243
1 DATA 0,252,72,138,72,152,72,173,136,2,141,148,48
  ,166,197,224,39                            :rem 139
2 DATA 208,12,228,197,240,252,173,13,48,73,255,141
  ,13,48,173,13                               :rem 31
3 DATA 48,240,38,224,47,208,61,228,197,240,252,160
  ,0,140,14,48,132                           :rem 173
4 DATA 198,32,249,241,240,251,24,105,198,141,15,48
  ,165,57,164,58                              :rem 99
5 DATA 205,16,48,208,5,204,17,48,240,92,173,15,48,
  141,18,48,162                               :rem 36
6 DATA 128,160,25,165,197,201,63,240,22,201,47,240
  ,200,173,14,48                              :rem 66
7 DATA 208,162,208,74,202,208,236,136,208,233,238,
  18,48,208,228                               :rem 46
8 DATA 120,162,0,181,0,157,76,49,202,208,248,162,8
  7,169,160,157                               :rem 43
```

```
9 DATA Ø,16,2Ø2,2Ø8,25Ø,32,129,229,165,57,164,58,1
  41,16,48,14Ø,17                            :rem 131
1Ø DATA 48,133,2Ø,132,21,32,2Ø7,48,162,Ø,189,76,49
   ,149,Ø,2Ø2,2Ø8                             :rem 68
11 DATA 248,32,135,229,88,1Ø4,168,1Ø4,17Ø,1Ø4,76,1
   52,227,224,55                              :rem 37
12 DATA 2Ø8,137,142,14,48,228,197,24Ø,252,2Ø8,18Ø,
   32,19,198,16Ø                              :rem 41
13 DATA 1,132,15,177,95,24Ø,67,32,44,2ØØ,234,234,2
   34,2ØØ,177,95                              :rem 24
14 DATA 17Ø,2ØØ,177,95,197,21,2Ø8,4,228,2Ø,24Ø,2,1
   76,44,132,73,32                           :rem 121
15 DATA 2Ø5,221,169,32,164,73,41,127,32,71,2Ø3,2Ø1
   ,34,2Ø8,6,165                              :rem 17
16 DATA 15,73,255,133,15,2ØØ,24Ø,17,177,95,2Ø8,16,
   168,177,95,17Ø                             :rem 96
17 DATA 2ØØ,177,95,134,95,133,96,2Ø8,181,96,234,23
   4,234,234,234                              :rem 53
18 DATA 16,215,2Ø1,255,24Ø,211,36,15,48,2Ø7,56,233
   ,127,17Ø,132,73                           :rem 12Ø
19 DATA 16Ø,255,2Ø2,24Ø,8,2ØØ,185,158,192,16,25Ø,4
   8,245,2ØØ,185                              :rem 32
2Ø DATA 158,192,48,178,32,71,2Ø3,2Ø8,245,96:rem 84
1ØØ PRINT "{2 SPACES}*****{7 SPACES}VERSATILE TRAC
    E FOR VIC BASIC{12 SPACES}*****          :rem 45
1Ø1 PRINT "{2 SPACES}*{7 SPACES}ASSUMES 8K OR 16K
    {SPACE}EXPANSION MEMORY{15 SPACES}*      :rem 26
1Ø2 PRINT "{2 SPACES}* F1 TOGGLES ON/OFF; F3 + NUM
    BER SETS SPEED OF TRACE;{4 SPACES}*      :rem 96
1Ø3 PRINT "{2 SPACES}* F5 SINGLE STEPS; F7 QUICK T
    RACE{24 SPACES}*                          :rem 7
11Ø FOR J=12288 TO 12619: READ X: POKE J,X: NEXT
                                              :rem 9
12Ø SYS 12288                                :rem 151
13Ø POKE 55,Ø: POKE 56,48: NEW                :rem 25
```

To use TRACE with a program which works only on the unexpanded VIC, switch on VIC with the 8K or 16K RAM in place. Now type in POKE 642,16: POKE 644,30: POKE 648,30: SYS 64818. VIC will now have BASIC from $1000 to $1E00, and the screen will be at $1E00, exactly as in the unexpanded VIC-20. Load and run the TRACE program, which puts TRACE into memory at $3000 (12288), where it is protected from BASIC. Load your BASIC program. Then type SYS 12288 to initialize TRACE and run your program.

When f1 is pressed, the program should be traced. F7 will move rapidly through the program. F5 will single-step. F3 waits for a numeral key from 0 to 9 before continuing. Programs controlled with ordinary keystrokes (not the function keys) are best.

Programs which have user-defined graphics may produce partly or completely illegible BASIC lines. This is one of the problems of designing TRACE for VIC. Other problems include finding memory for it, trying to prevent its controls from conflicting with ordinary program operations, and trying to guarantee that the normal screen isn't disrupted too much by TRACE's own display.

TRACE intercepts BASIC with a wedge. It performs various operations before returning to BASIC, which as far as possible is untouched. In operation, first, the key f1 is checked; if it's pressed, a flag is reversed. Then, if the flag is off, program control is returned to BASIC. If the trace flag is on, however, f3 is checked, and, if pressed, a number key from 0 to 9 is accepted and put into a delay loop. F5 is also tested; if the single-step flag is on, the program loops indefinitely waiting for f5. However, when this key is found, the program runs BASIC until it finds a new line number. The new line is listed on the screen, and the loop reentered. If f7 is found to be pressed, the delay loop is bypassed and BASIC lines are listed as rapidly as possible. In this way, you have maximum keyboard control over the trace.

As it stands, the program is not relocatable since it includes a number of internal storage areas. However, it isn't difficult for an experienced ML programmer to move it. Note that the routine at $307F (which lists lines) saves the entire zero page (otherwise, LIST corrupts many locations), homes the cursor and blanks the first four lines of the screen, lists the line with a modification of LIST, and restores the zero page and cursor position.

UNLIST

This system command prevents LISTing of BASIC programs to reduce the risk of unauthorized copying or modification. Unlisting is successful only in proportion to the difficulty of learning about a system. No widely sold microcomputer offers a foolproof protection scheme; however, temporary and makeshift expedients may be better than nothing.

Several suggestions follow. Note that disabling STOP and STOP-RESTORE was discussed earlier in this chapter.

Machine-language routine to run BASIC. This method is given first because it is usable by anyone, works with any memory configuration, saves normally, and is very puzzling to the uninitiated. It also disables STOP and STOP-RESTORE, so if the program has no errors, no explicit or implicit END, and no STOP statement, it can't be stopped at all by a user with an unmodified VIC. BASIC runs normally but lists as 0 SYSPEEK(44)*256+23 without any further lines.

To use this routine, follow these steps:

Step 1. Be sure that the program has no line numbered 0 or 1. Change numbering if it has.
Step 2. Enter line 0, with no spaces, in exactly this way: 0SYSPEEK(44)*256+23
Step 3. Enter line 1 with exactly 21 asterisks (or any other character) and no spaces, like this: 1*********************
Step 4. List lines 0–1 and check them.
Step 5. Type in X=PEEK(44)*256+23. This is the starting address of the ML you will poke in; it varies with memory, so use the variable X.
Step 6. Enter the following 24 POKEs. They are written as a continuous string of POKEs, but only to save space. You should enter them one by one. Check by PEEKing (use PRINT PEEK(X)) before you run. All must be correct. POKE X, 169: POKE X+1,45: POKE X+2,133: POKE X+3,43: POKE X+4,169: POKE

X+5,109: POKE X+6,141: POKE X+7,40: POKE X+8,3: POKE X+9,160: POKE X+10,0: POKE X+11,169: POKE X+12,PEEK(X+22): POKE X+13,145: POKE X+14,43: POKE X+15,32: POKE X+16,89: POKE X+17,198: POKE X+18,76: POKE X+19,174: POKE X+20,199.

Finally, type in POKE X−4,0: POKE X−3,0: POKE X+22,0.

Step 7. Save the program, list it, and run it to be sure that UNLIST is working correctly. Now show the result to a friendly hacker and see if he can list it.

Rather than reveal how this method operates, I'll explain a shorter version. Enter a short working program into an unexpanded VIC and add lines 0 SYS4111 and 1********** (ten asterisks). Perform the following twelve POKEs: POKE 4111,169: POKE4112,26: POKE 4113,133: POKE 4114,43: POKE 4115,32: POKE 4116,89: POKE 4117,198: POKE 4118,76: POKE 4119,174: POKE 4120,199. This completes the ML.

Now POKE 4107,0 and POKE 4108,0 to put end-of-program bytes after line 0. Your program should now LIST as 0 SYS4111,but it should RUN as normal. After the program is run, it will then list as usual.

The ten ML bytes are as follows:

```
$100F  LDA  #$1A
$1011  STA  $2B     ;Moves start-of-BASIC to the true start after ML
$1013  JSR  $C659   ;CLR sets pointers
$1016  JMP  $C7AE   ;runs program from start
```

The effect is identical to POKE 43,31: RUN. All that's needed is to add some unlist features and disable STOP and STOP-RESTORE to get an effective UNLIST.

Special characters in REM statements or dummy lines. Since characters following a REM don't affect a program's performance, there is ample opportunity to POKE or otherwise enter confusing characters which will disrupt a listing. A dummy line is one which is never run, always branched past. Some of the characters you can put in REM or dummy lines to disrupt listing are given below.

SHIFT-L stops a listing with ?SYNTAX ERROR.
{**WHT**} makes a listing to screen invisible.
HOME or CLEAR confuses a screen listing.
DELetes can erase a line from the screen, making it invisible.
Printer commands can (for example) force page throws.

See REM in this section for the ways to actually construct these lines. For example, try this on an unexpanded VIC: Type a few program lines, run them to make sure they contain no syntax errors, then add this line at the beginning:

1 REM""L

(the underline indicates that you should hold down SHIFT when you type the L). Now, position the cursor on the right quote mark and insert six spaces (hold down the SHIFT key and hit the INST/DEL key six times). Type six DELete characters by hitting INST/DEL six times (without SHIFT). Each DEL should appear as a reverse-video T. Hit RETURN to enter the revised line, then try to list your program. You should see only a single set of quote marks and a ?SYNTAX ERROR message. However, the program should still run properly. To make it possible to list the program again, simply delete line 1.

Five leading tokens method. This method, once considered for commercial use, causes a program's line numbers to list, but nothing else. It is easy to use. Add five colons (or any five characters or tokens) at the start of every line of BASIC. Then add these lines to the program, choosing your own line numbers if 50000 to 50002 are taken:

50000:::::S=PEEK(43)+256*PEEK(44): FOR J=1 to 1E8
50001:::::IF PEEK(S+4)>0 THEN POKE S+4,0:S=PEEK(S)+256*PEEK(S+1): NEXT
50002:::::END

Now, RUN 50000 will put null bytes into the start of each line. The program lines will now LIST as line numbers only but will run normally! Type LIST; you should see a set of line numbers, and nothing else. Delete lines 50000 to 50002 and the process is complete.

The following line (omit spaces to make it fit) can put the colons back, so the lines will LIST again: S=PEEK(43)+256*PEEK(44): FOR J=1 TO 1E8: POKE S+4,58: S=PEEK(S)+256*PEEK(S+1): IF S THEN NEXT.

With this method, about the best you can hope for is that would-be listers of your programs haven't read this book. You can also set traps, like putting :::NEW: or ::::X before a variable, rather than five colons, before UNLISTing the program. If the program is made listable again but these entries pass unnoticed, the program will be NEWed on running, or variable A may be mysteriously converted into XA.

Overlong lines. A line longer than 250 characters cannot be listed after the first 250 or so bytes. LIST expects each line to be spanned by a single-byte pointer and will loop indefinitely if not. However, some other commands, like READ, also fail to work.

To combine lines, replace the null byte at the end of each line but the last one with a colon, then move the lines down in memory to overwrite the link addresses and line numbers. The very first link of the series must be set to span the completed giant line, and all the later link addresses (which are now wrong) must be corrected.

If the idea interests you, put the following routine at the beginning of a program and run. Type in two line numbers; when the program has finished they'll be joined together. Each line number is printed as its line joins onto the first line selected; this ends up as a composite line, so the lines listed on the screen disappear from the program.

Program 6-45. Combining BASIC Lines

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
0 INPUT "COMBINE LINES";L,U: REM LOWER AND UPPER L
  INES                                          :rem 5
1 C=PEEK(43)+256*PEEK(44):E=PEEK(45)+256*PEEK(46)-
  4                                            :rem 44
2 LT=PEEK(C+2)+256*PEEK(C+3)                   :rem 46
3 IF LT<L THEN C=PEEK(C)+256*PEEK(C+1):GOTO2
                                              :rem 254
4 IF LT>L THEN PRINT "LINE NOT FOUND": END:rem 147
5 S=C:C=C+4                                   :rem 100
6 IF PEEK(C)<>0 THEN C=C+1:GOTO6               :rem 31
```

```
7 IF PEEK(C+2)=Ø GOTO 13                           :rem 23Ø
8 LT=PEEK(C+3)+256*PEEK(C+4): IF LT<=U THEN PRINT
  {SPACE}LT                                        :rem 2Ø1
9 IF LT<=U THEN POKE C,58:FOR J=C+1 TO E: POKE J,P
  EEK(J+4): NEXT: GOTO 6                           :rem 191
1Ø C=C+1: POKE S,C AND 255: POKE S+1,C/256:S=C:C=C
   +4                                              :rem 214
11 IF PEEK(C)<>Ø THEN C=C+1:GOTO11                 :rem 119
12 IF PEEK(C+2)<>Ø GOTO 1Ø                          :rem 76
13 PRINT C+3:C=C+1:POKE S,C AND 255: POKE S+1,C/25
   6: CLR:END                                        :rem 4
```

When this program is run, line numbers are printed, as is a value (line 13) which is the new, lower end-of-BASIC. It isn't necessary to POKE this in, but if you wish to save memory, you can do so. If, for example, 4567 is printed, type in POKE 45, 4567 AND 255: POKE 46,4567/256:CLR. Be sure to type it correctly. Otherwise, there'll be problems; incorrectly linked BASIC behaves in odd ways and may refuse to accept new lines or delete old ones. Remember not to include lines referenced by GOTO or GOSUB, or lines with IF statements or REM statements which will cause later parts of the newly joined line to be bypassed.

Self-modifying BASIC. If a program has only a few GOTOs and GOSUBs, this is an excellent way to get simple list protection. LIST needs a correct link address for each line of the BASIC program. However, RUN doesn't, except to process GOSUB or to GOTO a lower destination line than the command (10000 GOTO 100).

You can make use of this to get another type of UNLIST. With an unexpanded VIC, type in a few lines of program. Now POKE 4097,255 or some other random value. LIST will probably show garbage, but RUN will be satisfactory.

Before a GOTO or GOSUB of the sort just described, you'll need to POKE 4097 with the correct value for the program, then afterwards POKE in the wrong value again.

VARPTR

VARPTR finds the location of any variable stored in RAM. Its main use is to investigate variables, exactly as in the first part of this chapter. Program 6-46 loads a machine language routine which will find the starting location of a variable name, whether simple or subscripted. To be conveniently usable with BASIC, it uses ROM routines not only to find the variable, but (with LET) to assign the resulting address to another variable.

Program 6-46. VARPTR

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
1ØØ DATA 32,115,Ø,32,139,2Ø8,164,95,165,96,32,145,
    211,32,115,Ø                                  :rem 218
11Ø DATA 32,139,2Ø8,133,73,132,74,165,14,72,165,13
    ,72,76,186,2Ø1                                 :rem 8Ø
12Ø FOR J= 828 TO 859: READ X: POKE J,X: NEXT
                                                   :rem 74
```

After this is typed in and run, to put the ML into memory, the syntax SYS 828:AB$:L (for example) assigns to variable L the value of the address where AB$'s seven-byte description starts in memory. Program 6-47 is an example that finds and prints the value of X.

Program 6-47. An Example of Using VARPTR

```
200 X=123
210 SYS 828:X:S : REM S NOW = START OF X DESCRIPTI
    ON
220 FOR J=S TO S+6: PRINT PEEK (J);: NEXT: REM PRI
    NT 7 BYTES
```

This routine can't find TI, TI$, or ST, which are not stored as conventional variables. The actual machine language for the VARPTR routine is given below:

```
JSR  GETCHR  ; JSR $0073 (ignores separating colon)
JSR  SEARCH  ; with JSR SEARCH finds the variable
LDY  $5F
LDA  $60
JSR  FXFLT   ; Convert pointer bytes to floating-point
JSR  GETCHR  ; Ignore colon
JSR  SEARCH  ; Finds second variable
STA  $46
STY  $47
LDA  $08
PHA          ; Two entries on stack needed
LDA  $07     ; to assign value to variable
PHA
JMP  ASSIGN
```

Programmer's Aid

Programmer's Aid is a Commodore cartridge (designated VIC-1212) which fits either the VIC-20 or an expansion memory board. Its function is to aid BASIC program writing. It need not be present in memory when the final program runs, and the commands it adds to BASIC are all intended to be used in direct mode.

Don't let anyone convince you that this utility package is necessary for good programming or that programs cannnot be written without it. Both statements are untrue. Nevertheless, the cartridge is widely available and does provide many useful functions, so an overview of its use is included here.

Technical Description

Programmer's Aid ROM routines occupy locations $7000–$7FFF and so cannot be initialized simply by switching on VIC-20. $7009 is the entry point, so SYS 7*4096+9 or SYS 28681 is needed to activate it.

Once activated, the title PROGRAMMER'S AID will appear on the screen. BASIC's top of RAM is lowered by 120 bytes the first time SYS 28681 is used, to store function key definitions.

The KILL command turns *Programmer's Aid* off, but the 120 bytes aren't recovered. POKE 783,181: SYS 64815 will reset the system and recover these bytes. KILL followed by POKE 55,0: POKE 56, PEEK(56)+1: CLR recovers them and retains BASIC in memory; use this if you unexpectedly run short of memory during program development or if you wish to check the runtime speed of the program (which will increase by a small amount when *Programmer's Aid* is disabled).

SYS 28681 and KILL leave BASIC programs unaltered, as they should. STOP-RESTORE leaves *Programmer's Aid* active and retains BASIC. Don't reinitialize it.

Programmer's Aid 16 commands are listed from $73A6, with the high bit set, as RUN, AUTO, STEP, TRACE, OFF, RENUMBER, DELETE, HELP, FIND, DUMP, PROG, EDIT, CHANGE, KEY, MERGE, and KILL. The CTRL key is usable too. Abbreviated entry of these commands (with two keys, such as R SHIFT-E for RENUMBER) always works, because a wedge intercepts them before normal BASIC processing begins. This wedge has limitations, however; it works only in direct mode, and only for one command, so TRACE:KEY which looks all right won't in fact perform KEY.

Programmer's Aid commands are similar to those found in earlier utility chips for the PET/CBM.

Programmer's Aid can coexist, though not in a very happy state, with *VICMON* in RAM, and with *Super Expander* too, on an expansion board. Most *Programmer's Aid* functions operate normally unless they come across a *Super Expander* keyword, in which case they hang up in an infinite loop. FIND, CHANGE, and HELP are risky because *Programmer's Aid* cannot list a line containing a *Super Expander* keyword.

Using *Programmer's Aid*

Debugging commands: HELP, DUMP, TRACE, STEP, and OFF

HELP after a syntax error is supposed to list the defective line with a reversed character or reserved word at the place where the error was detected. Unfortunately, the interpreter often detects errors only when its BASIC pointer is away from the actual error, so this command isn't as useful as it sounds. HELP's pointer is the same that CONT would use; it is soon lost, so if you want help, get it right away.

DUMP looks for nonarray variables at the end of BASIC and prints them out in sequence. Try entering X=12.34: Y%=2:Z$="STRASMA" in direct mode, then DUMP. Arrays cannot be listed, a rather silly limitation. DUMP works only in direct mode. If you wish to dump within a program, use SYS 30817; this lists the variables, then stops.

TRACE puts six line numbers in a six-by-six reverse window at the top-right of the screen (unless the numbers have been replaced with user-defined characters); the lines themselves aren't listed. However, this may not be much help in tracing the way a program runs (see TRACE elsewhere in this chapter). OFF turns TRACE off. STEP is activated by the CTRL, Commodore, or SHIFT keys, but it isn't a true single step. It merely stops tracing when it finds one of these keys isn't pressed.

TRACE, STEP, and OFF can all be activated while a program runs. POKE 127,2 puts TRACE on; POKE 127,3 sets STEP; and POKE 127,0 turns TRACE off.

Line number commands: AUTO, DELETE, RENUMBER

AUTO a,b outputs line numbers in sequence starting with *a* and incrementing by *b*. For example, AUTO 1000,10 prints 1000 then, after entry of some BASIC and RETURN, 1010, and so on. This saves the effort of typing in line numbers. AUTO alone defaults to AUTO 100,10 and starts with 100. However, from then on the default varies. Generally, it carries on where it left off. This of course allows you to exit AUTO (by pressing RETURN twice), change a line or two, and then return to AUTO. SYS 30728,*a,b* has the same effect as AUTO.

DELETE offers a number of syntax alternatives; for example, DELETE -999 leaves lines 1000 and above. SYS 30415,*a-b* works too. DELETE on its own isn't allowed. It assumes conventional BASIC and (not surprisingly) can't be relied on in the presence of several line 999's, which RENUMBER makes easily possible. It can also cause problems with APPENDed line numbers out of sequence.

RENUMBER a,b is a two-parameter renumber which yields BASIC line numbers starting with *a* and incremented by *b*. RENUMBER 0,1, for instance, leaves the numbering as 0,1,2,3, and produces some speed increase. An increment of 0 is allowed; all the lines are then numbered 0. There is no way to renumber program segments, so if you'd like to keep some standard subroutines at line 60000, RENUMBER is not very much help. If there's no program in memory, RENUMBER hangs; STOP-RESTORE will recover. SYS 28908,*a,b* works exactly like RENUMBER.

Destination lines in GOTO, GOSUB, and RUN are renumbered; so is the line number designated by *a* in LIST *a-b*. ON is also renumbered despite the error in logic. Nonexistent line numbers are set to 63999, so FIND 63999 after RENUMBER will detect any loose ends.

Search/replace commands: FIND, CHANGE

FIND is valuable for quickly checking whether a variable has already been used in a long program. It isn't easy to use successfully for other purposes, because of the way BASIC is tokenized and stored. FIND T will find T (as well as T%, T$, T(8), and TA), but not TAN or "STAY", which must be found by FIND TAN and FIND "T" respectively. Generally, try it with and without quotes, and watch out for BASIC reserved words and for REM.

Spaces can be a problem. For instance, FIND GOSUB 500 won't find GOSUB500, and *Programmer's Aid* has no command to erase spaces. Each line with the sought item is printed. With long programs the text quickly scrolls off the screen. The command has a range restriction to avoid this problem; it has the form FIND ITEM,*a-b* with the usual permutations of *a* and *b*.

CHANGE has FIND's drawbacks plus some of its own. Assume you decide to change all occurrences of variable D into DEVICE, to make a program more readable. CHANGE D,DEVICE will change the variable, but it will also change a variety of other things. REM ADDRESS will become REM ADEVICEDEVICERESS, and variables like MD become MDEVICE. Strings are expected to be changed into same-length strings. The best approach may be to FIND all occurrences and change them manually, retaining the option of rejecting spurious occurrences.

Function key commands: KEY, EDIT, PROG

KEY lists the current definitions of twelve function keys. Function keys f1–f8 are the usual unSHIFTed and SHIFTed function keys; f9–f12 are CTRL plus one of the four keys. The Commodore key used with the function keys has no effect.

Two sets of key definitions exist in *Programmer's Aid,* starting at $7A90. EDIT and PROG switch between the two sets, which are given in Table 6-7. KEY lists them exactly as shown, except that the back arrow (representing RETURN) appears in reverse video on the screen. The idea is apparently for the PROG set to be useful while programming, offering a convenient method to enter single-key BASIC commands, while EDIT should be useful with the Programmer's Aid cartridge itself.

Table 6-7. Function Commands Listed by KEY

KEY 1,"LIST "	KEY 1,"LIST "
KEY 2,"MID$("	KEY 2,"AUTO"
KEY 3,"RUN ← "	KEY 3,"RUN ← "
KEY 4,"LEFT$("	KEY 4,"DELETE"
KEY 5,"GOTO"	KEY 5,"FIND"
KEY 6,"RIGHT$("	KEY 6,"CHANGE"
KEY 7,"INPUT"	KEY 7,"TRACE ← "
KEY 8,"CHR$("	KEY 8,"STEP ← "
KEY 9,"EDIT ← "	KEY 9,"PROG ← "
KEY10,"GOSUB"	KEY10,"RENUMBER"
KEY11,"RETURN ← "	KEY11,"MERGE"
KEY12,"STR$("	KEY12,"OFF ← "

KEY x, String Expression sets new values for the function keys. For example, KEY 1, "LOAD" + CHR$(34) + "$" + CHR$(34) + ",8" + CHR$(13) prints LOAD "$",8 and loads the directory from disk whenever f1 is pressed. Note, however, that string expressions longer than ten characters aren't allowed (12 lots of 10 bytes fill 120 bytes at the top of memory), which considerably weakens their appeal. Nevertheless, it's possible to fit in useful expressions. If you have a printer, for instance, OPEN 4,4:CMD4 and PRINT#4:CLOSE 4 will fit into two key definitions if you abbreviate the keywords. Use KEY 1, "OP4,4:CM4" and KEY 2, "PR4:CLO4" + CHR$(13), where the underlining indicates that the designated characters should be SHIFTed.

Program manipulation command: MERGE

MERGE is a modified LOAD which retains the current program in memory. *Programmer's Aid*'s version of MERGE preserves duplicate line numbers if these exist. For example, if LOAD loads a BASIC program from tape, then MERGE "",1 produces a listing with each line listed twice.

MERGE "STANDARD SUBROUTINES", 1 illustrates the sort of thing for which this command was designed. It has its own error message, ?MERGE ERROR, which is given if the program doesn't exist.

Screen editing: the CTRL KEY The routine at $7BCB tests for the CTRL (Control) key and one of L, N, E, U, Q, OR A. About 500 bytes processing follows.

CTRL-Q and CTRL-A let you scroll back and forth through BASIC, an enormous improvement on LIST. The cursor doesn't go to the top/bottom of the screen at once, so there are delays while it moves. Forward scrolling leaves large spaces between program lines.

CTRL-L , CTRL-U, and CTRL-N clear parts of the screen without erasing the program. CTRL-L clears the program line (up to four screen lines) on and after the cursor. CTRL-U clears the whole program line. CTRL-N clears the screen after the cursor. As an example, if a line begins 100 PRINT, with the cursor on P, then CTRL-L leaves only 100 on the screen. However, if additional BASIC lines are added and RETURN is pressed, the new line 100 replaces the old. These commands seem to be modified from CBM's BASIC 4.

CTRL-E turns off quote mode.

***Programmer's Aid* Machine Language** Internal evidence (unused pointers and characters) suggest that *Programmer's Aid* was written to start at $A000. It also tests for ROM at $B000, where no Commodore cartridge fits. This explains why it starts at $7009 rather than at $7000. Initialization is similar to that of the *Super Expander*, but *Programmer's Aid* doesn't take more memory on repeatedly initializing and uses this wedge:

```
0073 INC  7A
0075 BNE  0079
0077 INC  7B
0079 LDA  BASIC
007C JMP  731C
```

This leaves room from $7C to $8A for other storage; for example, $7F holds the trace indicator. KILL of course reinserts the normal CHRGET routine to get a BASIC character and set significant flags.

The entry addresses of the commands (in two tables, starting at $73EE and $73FE) are as follows:

$70EC	RENUMBER	$7780	HELP
$740E	RUN	$7808	AUTO
$7423	CHANGE	$7861	DUMP
$742B	FIND	$7943	KEY
$7610	OFF	$7A31	KILL
$7614	TRACE	$7BB1	PROG
$7617	STEP	$7BBE	EDIT
$76CF	DELETE	$7E0E	MERGE

Machine language programmers might note an unusual feature of the code: JSR $72D5 has to be followed by two parameters, which often appear as ??? on disassembly. Subroutine $72D5 moves two bytes from the first parameter's address to the second parameter's address.

Chapter 7

6502 Machine Language

Chapter 7

6502 Machine Language

Most programmers feel that machine language programming is more difficult than programming in BASIC, but by the end of this chapter you should have a good grasp of ML techniques on the VIC-20.

In the following discussion, machine language instructions are presented (as far as possible) in sequence from the easiest to the trickiest. It is assumed that you're familiar with hex notation (Chapter 5) and that you have an ML monitor available (for example, *VICMON*). Readers without a monitor may type in the monitor at the end of this chapter. That monitor is written in BASIC and requires memory expansion to be fully functional.

Note that Chapter 10 is a complete reference section, with examples, on all 6502 instructions. It can be used to help you with your own ML programs.

There are four subdivisions of Chapter 7. First, you'll see actual examples of ML programming to help you start learning. That is followed by a full description of the 6502 chip. Next is a list of problem-solving techniques. Finally, there is a discussion of machine language monitors for the VIC-20.

Introduction to 6502 ML Programming

Examples

In this section, you'll write some short ML programs, using only the simplest ML instructions. Each example should be entered with a monitor. *VICMON* offers a fairly standard format, which the examples use. At this stage, only four monitor commands will be considered: A (Assemble) for entering ML; D (Disassemble) to convert ML bytes into readable form, so they appear as they did during assembly; M (the Memory Display command) which displays consecutive bytes; and G (Go or Go Run) which executes the program, much as RUN executes programs written in BASIC.

The monitor program presented with this chapter has the first two commands, provided your VIC has expanded memory, but not the third; however, bytes can be PEEKed from BASIC to check statements made about M.

Note that leading dollar signs ($) are not used to indicate hex arithmetic in the BASIC monitor, so type (for example) 033C instead of $033C when entering the example programs. Also, the example ML routines end with BRK; this is fine for *VICMON* and ML monitors, but BASIC monitors require that routines end with RTS, since they are called with SYS (typically SYS 828 rather than G 033C). RTS returns control to BASIC.

These programs all put characters into screen memory, so the effect of each program is instantly visible. Direct feedback like this will prove helpful in learning.

The VIC-20 has movable screen memory, and these programs assume a screen start of $1E00 (applicable to an unexpanded VIC). But the screen may be at $1000, because the full BASIC monitor can't be fitted into VIC's original small memory. In this case, substitute 1000 for 1E00 and 9400 for 9600 when color RAM is used. If you're not sure where the screen is, PRINT PEEK(648). A value of 30 means the screen starts at $1E00. Color RAM starts at $9600 when the screen is at $1E00; when the screen is at $1000 it starts at $9400.

Example 1: Placing a Single Character on the Screen

With VICMON. Activate *VICMON* with SYS 6*4096 (some *VICMON* cartridges require SYS 40960 instead), and type in the six-byte ML program exactly as shown, using either the M or A command. The two forms are exact equivalents; they are simply different ways of showing the same thing. For example, the byte A9 is treated as an LDA (LoaD Accumulator) instruction by the 6502, and the D or disassemble command simply expands A9 into LDA whenever it is found. This is analogous to BASIC's LIST command, which expands one-byte tokens into keywords.

```
.A 033C  LDA  #$00              .M 033C 0341
.A 033E  STA  $1E00       or    .:033C A9 00 8D 00 1E
.A 0341  BRK                    .:0341 00 --- any ---
```

You'll find that a disassembly command, for instance .D 033C 0341, disassembles the bytes so they appear exactly as they did when you typed them in.

Note that, looking at the six bytes in memory, the screen starting address $1E00 is held in reverse order, with 00 preceding 1E. This feature is common to all three-byte instructions of the 6502 and many other microprocessors.

The command .G 033C executes this short program, then returns to *VICMON.* Its effect is to print an @ symbol in the top left of the screen, unless the screen scrolls and loses it (or unless there was no character there already so color RAM is white, making the @ invisible).

Why does this work? You know that $1E00 is the first screen position and that POKEing 0 to the screen generates @. That should give a clue. The program loads the *accumulator* (also called the A register) with 0 (LDA #$00), stores the accumulator's content in $1E00 (STA $1E00), then breaks (BRK) to return to *VICMON.* The accumulator is an eight-bit location within the 6502 processor that can be loaded with any value from $00 to $FF. This example has the same effect as POKE 7680,0.

Can you do more with this? If you cursor up and alter the program to LDA #$01 (or 033C A9 01, etc.), then G 033C has the effect of POKEing 1 into the screen, so the letter A appears. In fact, you can put any character into any screen location, after a certain amount of calculation to determine the address, and after looking up the screen POKE value from the appendices.

With the BASIC monitor. Load and run the monitor, preferably the full version with both Assembly and Disassembly facilities, but otherwise the Assembly portion. Then assemble from address 033C, entering the following lines:

```
033C  LDA  #00
033E  STA  1000
0341  RTS
```

Use STA 1E00 if your VIC is unexpanded. The query and quotes generated by the INPUT statement have been omitted; they aren't central to the program. Now, press RETURN, which inputs nothing and exits to READY mode. SYS 828 returns to

READY, after putting @ at the top left of the screen, just as the similar ML of *VICMON* does. For J=828 TO 833:PRINT PEEK(P): NEXT prints the six bytes of ML, and is analogous to *VICMON's* M command. Machine language programs can be POKEd into memory by reversing this procedure; Chapter 9 includes a program which converts ML into BASIC DATA statements.

To underline the fact that BASIC can POKE in and use ML programs, type in FOR J=1 TO 255: POKE 829,J: SYS 828: NEXT which prints all 256 characters in quick succession at the top left of the screen. Each loop alters the ML program, then executes it in its new, slightly changed, form. Disassembly of the program in its final form gives the following:

```
033C LDA #FF
033E STA 1000
0341 RTS
```

showing how the second byte of the six in the sequence ended as $FF, or 255. Exactly the same BASIC will work from *VICMON*, after exit to BASIC with *.X*, but remember to end the ML with RTS, not BRK, so the SYS call will operate properly.

Example 2: Placing a Character on the Screen with Color

This program will put a character on the screen and place a byte into the same character's color RAM. The comments are for information only and are not intended to be typed in; *VICMON* won't accept them.

With *VICMON*. Enter the following with the A command:

```
.A 033C LDA #$00   ;LOAD ACCUMULATOR WITH ZERO
.A 033E STA $1E00  ;STORE ACCUMULATOR IN SCREEN
.A 0341 STA $9600  ;STORE ACCUMULATOR IN COLOR RAM
.A 0344 BRK        ;BREAK, BACK TO VICMON
```

This nine-byte program will disassemble, with D 033C 0344, into exactly the same form. If you use M 033C 0344, you will get this:

```
.: 033C A9 00 8D 00 1E
.: 0341 8D 00 96 00 --
```

This is similar to the first example, except for the introduction of 8D 00 96, bytes which decode to STA $9600 just as 8D 00 1E is STA $1E00.

The command G 033C executes the program, causing @ to appear in black at the top left of the screen. It is black because color RAM treats 0 as black. Cursor up and replace LDA #$00 by LDA #$02; G 033C then prints a red B.

With the BASIC monitor. Assemble this at 033C:

```
033C LDA #00
033E STA 1000
0341 STA 9400
0344 RTS
```

Use STA 1E00 and STA 9600 with an unexpanded VIC.

SYS 828 prints a black @. As before, FOR J=0 TO 255: POKE 829,J: SYS 828: NEXT will cycle through all VIC's characters, but this time the colors cycle too.

From now on, the examples will assume *VICMON*; this is to save space. You should have little difficulty making the small amendments which may be needed to convert one format into the other.

Example 3: Introducing an Index

This example introduces the X register and its use as an index. The X register is another eight-bit location within the 6502 processor, like the accumulator. Notice its use in the following program:

```
.A 033C  LDA  #$00     ;LOAD ACCUMULATOR WITH #0
.A 033E  TAX           ;TRANSFER ACCUMULATOR TO X
.A 033F  STA  $1E00,X  ;STORE ACCUMULATOR IN SCREEN + X
.A 0342  LDA  #$00     ;LOAD ACCUMULATOR WITH 0
.A 0344  STA  $9600,X  ;STORE IN COLOR RAM + X
.A 0347  BRK           ;BREAK
```

The TAX (Transfer Accumulator to X register) instruction simply copies the contents of the accumulator into the X register. The ,X in STA $1E00,X is a special notation. It refers, not to address $1E00, but to address $1E00 plus X's current contents. That is, whatever value is in X is added to $1E00, and the resulting address used in the instruction. Since X has eight bits, the range must be within $1E00–$1EFF in the example. Likewise, the address for STA $9600,X must be within the range $9600–$96FF.

If you execute this program with G 033C, it prints an @ in black at the top left corner of the screen, exactly like the previous program. The difference appears after cursoring up and altering LDA #$00 to (for example) LDA #$05. Executing this prints E in black, but prints it five characters over from the @ in the top left. Any other value in place of the initial $00 prints a character offset from the screen starting location.

Now change BRK to RTS, type X to exit to BASIC, and type in FOR J=0 TO 255: POKE 829,J: SYS 828: NEXT. This prints all 256 characters consecutively in black, filling the top half of the screen, and shows clearly how the index X operates. POKE 835 with another color value, say 2 for red, to watch the effect of the ML after $0342.

Example 4: Loops with Machine Language

You have just used BASIC to cause a loop, but how can you do this in ML? As in BASIC, you will need a counter to check the number of loops, plus a test for the end of the loop.

The example shows a standard way of doing this with the 6502; it is standard because the instructions are designed for this very purpose. The program has an increment instruction (INX) and a branch instruction (BNE). A decrement instruction can be equally useful, and sometimes better, but increments are easier for beginners.

```
.A 033C  LDX  #$00     ;LOAD X REGISTER WITH #0
.A 033E  TXA           ;TRANSFER X TO A (HAPPENS 256 TIMES IN LOOP)
.A 033F  STA  $1E00,X  ;STORE A IN SCREEN START + OFFSET X
.A 0342  LDA  #$02     ;SET COLOR RED
.A 0344  STA  $9600,X  ;STORE COLOR IN COLOR RAM + OFFSET X
.A 0347  INX           ;INCREMENT X REGISTER
```

```
.A 0348  BNE $033E    ;BRANCH IF X NOT EQUAL TO ZERO
.A 034A  BRK          ;BREAK WHEN X CYCLES THROUGH TO #0
```

With this ML in memory, G 033C prints 256 characters in red in the top half of the screen. It does this far faster than the equivalent BASIC version given in Example 3; in fact, it takes only about 1/200 second.

How does this work? First, both the accumulator and X are loaded with #0. (The TXA transfer uses one fewer byte than LDA #$00.) TXA (Transfer X register to Accumulator) has the property of insuring that the offset X corresponds to the character in the accumulator, so that after the branch at 0348, which is taken 255 times, the contents of the accumulator depends on the value of X. This is a short way to program the result and depends on the use of INX to increment X by 1. Note that the value stored in screen memory cycles from $00 to $FF, but the value stored in color RAM is always 2. Thus the color of each character stays constant.

To understand this program fully, note the values in the accumulator and X at each stage of the program. The value in X progressively increases, until eventually it reverts from $FF to $00, while the value in the accumulator alternates between the identical (increasing) value of X and 2.

Note too that the branch instruction only occupies two bytes, in spite of disassembling to three bytes. It uses relative addressing, meaning that the branch, if taken, goes to the address of the following instruction plus the byte just after the branch instruction. The example adds $F4 to $034A, treating $F4 as negative (or −$0C, since $F4+$0C=$00). In addition, $034A−$0C=$033E. Branch instructions can therefore reach only 127 bytes forward or 128 back. More on this later on.

Example 5: Comparisons and Subroutines in ML

You have seen how SYS calls can run ML as a subroutine, provided the RTS instruction ends the ML. This also operates in ML itself; RTS (ReTurn from Subroutine) is analogous to RETURN in BASIC. Try the following program after changing BRK to RTS in Example 4's ML:

```
.A 0350  JSR  $033C  ;CALL EXAMPLE 4'S LOOP AS A SUBROUTINE
.A 0353  INC  $0343  ;INCREMENT THE COLOR IN EXAMPLE 4
.A 0356  LDA  $0343  ;LOAD A WITH THE NEW COLOR
.A 0359  CMP  #$08   ;COMPARE THE NEW COLOR WITH 8
.A 035B  BNE  $0350  ;BRANCH IF NOT EQUAL TO 8
.A 035D  BRK         ;BREAK WHEN COLOR = 8
```

Now G 0350 cycles through the colors until the last color (yellow) is reached. Because the subroutine is changed by this program, G 0350 behaves differently the second time the program is executed. The point is that, as in BASIC, ML subroutines provide a powerful means of dividing programs into manageable chunks. You'll see later that comparisons can be followed by other types of branch than BNE or BEQ (Branch if EQual); the illustrations here are used for simplicity.

Because of the speed of ML, the colors on the screen are changed too fast to be visible. As an exercise, add a delay loop after 0350 JSR $033C, to use up time without performing significant processing work. Hint: Use the X and Y registers; Y is the other eight-bit register of the 6502. Construct two loops within each other, and use DEX (DEcrement X register) and DEY (DEcrement Y register), each followed by BNE, so X decrements 256 times for each decrement of Y. Remember that STOP-RESTORE generally returns you to BASIC if your program doesn't work.

Full Description of the 6502 Chip

This section describes the 6502 microprocessor by looking at addressing modes; the status register (N, V, B, D I, Z, and C flags); the program counter, zero page, and stack; NMI, RESET, and IRQ vectors; and opcodes.

Opcodes (machine language instructions) are introduced last because their use depends on prior knowledge of the other 6502 features. Chapter 10 has a guide, with notes, on all the opcodes; and the appendices have comprehensive tables, giving concise information on the 6502 for experienced ML programmers.

Addressing Modes

The 6502 has 13 addressing modes. Most are easy to understand, but a few are more difficult.

You've seen how disassembly always treats a given byte in the same way; 8D *xx yy* is always treated as STA *yyxx*. In other words, this is implicit in the chip: Whenever 8D is encountered as an instruction, the following pair of bytes is considered to be an address in low/high byte order. A disassembler therefore prints STA in place of 8D and follows it with a 16-bit address.

Most addressing modes process the contents of memory locations, rather than using explicit values. This is invaluable in dealing with RAM and ROM where the processor is often a sort of intermediary, mainly concerned with arranging blocks of RAM. However, it is a rather abstract property, which takes some time to grasp.

All 6502 instructions take one, two, or three bytes; each type will be examined in some detail.

Single-byte instructions. Single-byte instructions cannot reference either address or data, and operate only on features within the 6502. The phrase *addressing mode* doesn't really apply, but for consistency these instructions are described as possessing *implied* addressing. Instructions which shift or rotate bits in the accumulator like ASL (Arithmetic Shift Left) are sometimes distinguished as *accumulator* addressing. At other times, however, they are not, so you may encounter monitors which require ASL A rather than just ASL.

Two-byte instructions. These instructions consist of an instruction followed by a single byte. If this byte is treated as data, the instruction uses *immediate* mode. This is usually indicated by a number sign (#) before the data; you saw examples in the programs of the last chapter. Apart from loading the accumulator or X and Y registers with a value, this addressing mode is used in arithmetic operations, logical operations, and comparisons.

All other two-byte instructions refer to addresses, not data. There are six different types. You have already used one of them, branches, in the previous chapter. That addressing mode is usually called *relative* because of its use of an offset.

Zero page instructions. The five remaining two-byte modes all use *zero page* addressing. The zero page is not a feature of the chip itself; it is the section of RAM (or ROM) which is wired to addresses $0000–$00FF. However, the chip has the facility of enabling the most significant byte to be ignored, so that (for example) LDA

$34 can be written in place of LDA $0034. This saves a byte, which in turn shortens programs and increases their speed. For this reason, the first 256 bytes are usually in great demand in 6502 programs, so that machine language routines which coexist with BASIC must be careful to take into account BASIC's use of these locations.

In the simplest type, the second byte specifies the address in zero page. For example, LDA $55 loads the accumulator with the contents of address $0055; location $55 may hold any value from $00 to $FF. Note the difference between this and the immediate mode instruction LDA #$55 which loads the value $55 into the accumulator, and has no connection with location $55.

Zero page indexed by X. LDA $A0,X loads the accumulator from an address calculated by $A0 plus the contents of the X register at the time the instruction is carried out. Note that the total of $A0+X is itself treated as a zero page address; if there is overflow, it is ignored. For example, if X holds $60, $A0+$60 is treated as $00, not $0100, and the contents of address 0 are loaded into the accumulator.

Zero page indexed by Y. This is exactly analogous to the previous mode, but the chip is designed so that only two instructions can use this mode (LDX and STX).

Indexed indirect. An example of this type of instruction is LDA ($00,X). The parentheses are a convention which indicates that the accumulator is loaded from an *indirect* address. That is, the quantity in parentheses specifies the address of the first of two consecutive zero page bytes which form the address from which the data is taken. Let's assume for the moment that X contains 0, to simplify matters. In effect, LDA ($00,X) would then be equivalent to LDA ($00), since the indexing effect of X is zero.

Suppose the first few bytes in zero page are 01 80 84 02. LDA ($00) loads the accumulator from the address it finds in the bytes in locations $00 and $01, in this case $8001. So the instruction, in this instance, has the same effect as LDA $8001.

Pure zero page indirect addressing is not available on the 6502. Indexed indirect addressing, as the name implies, allows indexing of the indirect address. Thus, if X were loaded with $02, then LDA ($00,X) has the effect of loading the accumulator from the indirect address of $00+$02, or ($02). Therefore, with the figures above, the equivalent of LDA $0284 is executed. The instruction is useful when X is set to $00, as pure indirect addressing of the zero page, and when a table of pointers exists in the zero page. The BASIC pointers to the start of BASIC, its end, and its variables provide an example. This instruction is asymmetrical with respect to the X and Y registers; see STY in Chapter 10 for additional information.

Indirect indexed. An example of this type of addressing is LDA ($2A),Y. As with the previous mode, the address in parentheses specifies the location of the first of two consecutive bytes which together form an address. Apart from this special case, however, this mode is post-indexed by Y; that is, the indirect address is calculated, then the value in the Y register is added, and the resulting address is the object of the processing.

To show how this works, consider the data shown in the indexed indirect mode example above, with four possible bytes at the very start of RAM. LDA ($00),Y loads from $8001+Y, so the 256 bytes from $8001 to $8100 can all be accessed, depending on the value in Y.

Indirect indexing can only be used with the Y register. It is used for pure indirect addressing (when Y contains $00) for such purposes as following the link pointers from one BASIC line to the next and for processing blocks of data which aren't in the zero page.

Three-byte instructions. Three-byte instructions in the 6502 always consist of an instruction followed by a two-byte address. There are four interpretations of the address: absolute, absolute indexed by X, absolute indexed by Y, and absolute indirect.

Absolute. This mode is a simple reference to a two-byte address, as in LDA $1234 or LDA $8000 or LDA $0012.

Absolute indexed by X. The contents of X are added to the address to give the actual referenced address. Thus, if X holds $50, LDA $8000,X loads the accumulator from $8050. As with zero page indexing, the maximum value cannot exceed the legitimate range, so LDA $FFF0,X when X holds #$11 loads the accumulator from $0001, not from the nonexistent $10001.

Absolute indexed by Y. This is exactly analogous to the previous mode, except that it is indexed by Y.

Indirect. The 6502 has only one instruction with this mode, namely JMP (JuMP). An indirect jump transfers the program's flow of control to a new address; this address is found from the contents of the address indicated by the indirect instruction. An example is helpful: Suppose the first few bytes in zero page contain the values shown above in the example for indexed indirect mode. In that case, JMP ($0000) has the same effect as JMP $8001; JMP ($0001) jumps to $8480; and so on. This instruction is useful when a table of addresses (like the three vectors at the top of RAM) exists in a block. For example, the RESET vector at $FFFC/FFFD can be called by JMP ($FFFC) irrespective of BASIC ROM.

The Status Register

The status register (or processor status register), denoted SR by *VICMON*, is another eight-bit register. It contains seven individual status bits, or *flags*, all of which are automatically controlled by the 6502 chip as ML programs run. Bit 5 of the register isn't used and is permanently set at 1. Table 7-1 lists all possible bit-pattern combinations for the status register. Note that values of 0, 1, 4, 5, 8, 9, C, or D are not possible in the high nybble, since bit 5 is always set to 1. This means that the value in the status register will always be at least 32, $20, even when all flags are clear. For example, if the register contains $32, then B is set (high nybble $3), and Z set (low nybble $2).

Table 7-1. VIC Status Register

7	6	5	4	3	2	1	0
N	V	1	B	D	I	Z	C

High Nybble				
2			•	
3			•	B
6		V	•	
7		V	•	B
A	N		•	
B	N		•	B
E	N	V	•	
F	N	V	•	B

Low Nybble				
0				
1				C
2			Z	
3			Z	C
4		I		
5		I		C
6		I	Z	
7		I	Z	C
8	D			
9	D			C
A	D		Z	
B	D		Z	C
C	D	I		
D	D	I		C
E	D	I	Z	
F	D	I	Z	C

These flags don't change unless altered by an instruction. The decimal mode (D) bit, for example, typically remains off through all BASIC programs.

The table of 6502 opcodes in Appendix N shows which instructions alter which flags. LDA, for instance, alters (or rather conditions, because the new value may be the same as the existing value) the N and Z flags, but none of the others. This process is automatic; it's part of LDA and happens even if you have no need for it. However, a few instructions do explicitly set flags: CLC (CLear Carry) sets the C flag to 0, and SEC (SEt Carry) sets C to 1.

The logic behind the use of flags can be difficult to follow at first. The V and N flags, in particular, are tricky, while Z and I are much simpler. With practice, the programmer should find them easy enough (or at least be able to avoid the awkward ones). For instance, V is seldom used.

The N, or negative, flag (bit 7 of SR) is a direct copy of bit 7 of the result of some other operation. Thus, LDA #$D3 loads $D3 into the accumulator, and since $D3 is shorthand for binary 1101 0011 (which has bit 7 high), N is turned on by this instruction. Some hardware ports are wired up to bit 7, so LDA from the location sets or clears N to reflect the status of bit 7. N is used along with BMI (Branch on MInus) or BPL (Branch on PLus), the branches being taken if N is 1 or 0 respectively.

The negative idea is part of twos complement arithmetic, which is dealt with in the next section.

The V, or internal overflow, flag (bit 6 of SR) is seldom used. Like N, it's related to twos complement arithmetic and indicates typically that two numbers added together give a result outside the acceptable range. See the next section.

The B, or break, flag (bit 4 of SR) is usually set only on BRK. Its purpose is to enable a BRK instruction to be distinguished from an interrupt, since both jump to the same address. The address is fixed in ROM. This is a hardware feature of the 6502, discussed in greater detail later in this section.

The D, or decimal calculation mode, flag (bit 3 of SR) changes the mode of addition and subtraction performed by the 6502 to decimal, actually binary coded decimal) instead of the usual binary. The results resemble ordinary decimal arithmetic. Again, this concept is not a simple one. As an illustration, consider adding 35 to 97. In hex, the result is $CC; in decimal mode, it is 32 with the carry flag set, identical to the normal decimal outcome. The 6502 automatically adds 6 to either nybble if a result exceeds 9.

The I, or interrupt disable, flag (bit 2 of SR), when set with SEI (SEt Interrupt flag), prevents any IRQ interrupts from taking place. Chapter 8 explains these interrupts, with examples, but due to their importance in handling the keyboard, they are mentioned in several places in this book. The main point of disabling interrupts is to prevent them from disturbing ML routines which temporarily can't handle interrupts, often when a new piece of programming is inserted into the regular interrupt sequence.

The Z, or zero result, flag (bit 1 of SR) is set by most of the instructions which set N. Z ORs together all eight bits of a result; if this process gives a value of zero, the Z bit is set to show a zero result. Otherwise, when Z is off, the result is nonzero. The notes to BEQ and BNE in Chapter 10 expand on this.

The C, or carry, flag (bit 0 of SR) is primarily of use in addition or subtraction, where its function is similar to the carry which denotes overflow from a column of figures to a more significant column. BCC, BCS, CLC, and SEC are other instructions involving this flag.

The Program Counter, Zero Page, and Stack

The program counter (PC) is a 16-bit register within the chip that records the address of the current instruction being executed. The register can't be accessed directly; BRK, or an interrupt, saves its value on the stack, as does JSR. Thus, the value of the PC can be determined after BRK, for example, which is how *VICMON* is able to record the PC. Branch and jump instructions operate at chip level by loading new values into the PC, transferring control to some new program location.

The zero page, as you have seen, is the section of memory from $00 to $FF. Because many 6502 instructions can use zero page addressing modes, which are faster (and shorter) than addressing elsewhere, this region is important and invariably connected to RAM. It's called a page because blocks of 256 bytes form a natural subdivision of 6502 addressing; there are 256 pages in all.

The stack is a part-RAM, part-hardware feature of the 6502. It uses page 1 of RAM, from $100 to $1FF, and it can be difficult to understand for several reasons. First, page 1, although used by the processor as the stack, also doubles as normal RAM. Second, instructions like PHA (PusH Accumulator onto stack) and its opposite PLA (PuLl Accumulator from stack), which are used for temporary storage purposes, work in a fairly complex way, adding new bytes to the lower end of the stack and recovering old bytes from the lower end, under the control of another eight-bit register, the stack pointer. The process is explained in Chapter 10. Note that another complementary pair of instructions, PHP and PLP, operate on the status register, allowing it to be stored and examined at will. Four other instructions operate on the stack: JSR (Jump to SubRoutine), its converse RTS, BRK, and RTI (ReTurn from Interrupt). The stack pointer can be read or reset by copying values to or from the X register, using the TSX (Transfer Stack pointer to X register) or TXS (Transfer X register to Stack pointer) instructions respectively.

NMI, RESET, and IRQ Vectors

The 6502 has a group of reserved addresses, defined in hardware, at the top of its addressing area. The top of memory is therefore invariably ROM. Whenever the NMI, RESET, or IRQ pin of the 6502 is grounded, the processor sets the program counter to the address in location $FFFA/FFFB, $FFFC/FFFD, or $FFFE/FFFF respectively. For example, when the VIC is turned on, after a short delay, the RESET line of its 6502 is held low, causing the processor to look to locations $FFFC and $FFFD for the address for the standard switch-on sequence. If you check these addresses with PRINT PEEK(65532)+256*PEEK(65533), you'll see that the VIC's ROM reset routine begins at location 64802 ($FD22), as discussed in Chapter 5.

Briefly, the RESTORE key uses NMI, and NMI can also be programmed. RESET is valuable in program recovery, to restore programs which have crashed in otherwise unstoppable loops. IRQ is used by the VIC to read the keyboard. Chapters 5, 6, and 8 consider the software side of these hardware features.

6502 Instructions and Opcodes

An opcode (operation code) is a number, an eight-bit byte, that instructs the microprocessor to perform a particular action. Since humans find it easier to deal with letters rather than numbers, the opcodes are usually represented by *mnemonics*, character representations intended to make machine language relatively easy to read. All 6502 opcodes are three letters long, which makes for neat assembler and disassembler listings.

Although the mnemonics are standard, there is nothing to stop you from coming up with your own (for example, by modifying the BASIC monitor program in this chapter). This may in fact be helpful as a learning aid, although it would be unorthodox.

There are 56 distinct types of instructions (and hence 56 different standard mnemonics), some with one addressing mode, some with as many as eight, for a total of 151 valid opcodes. They can be grouped by function, as shown below.

Add/subtract. ADC (ADd with Carry) and SBC (SuBtract borrowing Carry) are the 6502's arithmetic functions. Both addition and subtraction are carried out on all eight bits, using the carry flag (C) for overflow. Twos complement arithmetic is not used, but flags are present which enable it to be implemented. A binary-coded decimal (BCD) arithmetic mode is also available.

Branches. The 6502 has eight branch instructions, all conditional on the status of a flag, and all having a single-byte twos-complement offset. The instructions are BCC and BCS, BNE and BEQ, BPL and BMI, BVC and BVS, and the branch is taken if the C, Z, N, or V flag is off (clear) or on (set) respectively.

Break. The BRK instruction causes an unconditional jump to the address in locations $FFFE/FFFF, having first saved both bytes of the program counter and the status register on the stack.

Comparisons. CPX, CPY, and CMP make it possible to compare the contents of X, Y, and A (the accumulator) with data or with memory contents. The data or memory is subtracted from X, Y, or A, and flags are set, without changing the value in the register. N, Z, and C are set, so a comparison may be followed by any branch (except BVC or BVS) to test the comparison.

Data transfers. Data can be loaded into the 6502 from RAM or ROM by LDA, LDX, or LDY; it can be stored in RAM by STA, STX, or STY. These few instructions are extended in power by being equipped with a large number of addressing modes.

Decrements/increments. These alter X, Y, or memory locations by subtracting or adding one bit, setting N and Z according to the result. The instructions are DEX, DEY, DEC, and INX, INY, INC.

Flag clear/set. These enable some status register flags to be altered at will. CLC, CLD, CLI, and CLV clear flags C, D, I, and V; SEC, SED, and SEI set flags C, D, and I.

Jumps. JMP acts like GOTO in BASIC. JSR acts like GOSUB, with RTS the equivalent of RETURN. JSR saves the current address plus two on the stack.

Logical operations. AND, EOR (exclusive OR), and ORA (inclusive OR) perform binary logical operations on the accumulator and data or memory, retaining the result in the accumulator, and setting the N and Z flags. BIT sets the Z flag just as AND would, but does not affect the contents of the accumulator.

No operation. NOP does nothing.

Return. RTS returns to the instruction following JSR by jumping to the address currently on the stack, plus one. RTI jumps to the address on the stack and also loads the status register from the stack.

Rotate/shift. ROL (ROtate Left) and ROR (ROtate Right) act on the accumulator and the C (carry) flag (a nine-bit rotation). For example, an ROL causes all bits in the accumulator to move one position to the left; the leftmost bit (bit 7) is pushed out into the carry flag, and the old contents of the carry flag wrap around into the rightmost bit of the accumulator (bit 0). ASL (Arithmetic Shift Left) and LSR (Logical Shift Right) also involve the accumulator and C (but do not rotate C) so that bit 0 with ASL and bit 7 with LSR are always set to zero. Flags N, Z, and C are set.

Stack operations. These are PHA, PHP, PLA, and PLP. These are explicit operations on the stack, but BRK, JSR, RTS, and RTI also use the stack. TSX and TXS allow the stack pointer to be found and set respectively.

Transfers between registers. Six instructions allow transfers between any two registers Y, A, X, and S. The opcodes are TYA and TAY, TAX and TXA, and TXS and TSX.

Note. Not all 6502 machine language is instructions; tables of data are a common, and necessary, feature, and these can usually be identified by the fact that they don't disassemble sensibly. Chapter 5 explains about such tables. BASIC ROM starts with tables, including address tables (that is, tables of 16-bit numbers), BASIC keywords, and BASIC messages.

Timing

All opcodes take a precise number of 6502 clock cycles; the faster the clock, the faster the ML executes. VIC's chip runs at about one million cycles per second. Table 7-2 summarizes timing in the 6502; most instructions are included in the first column, but a few exceptional instructions are listed in the other columns.

Table 7-2. 6502 Timing Reference Chart

Addressing Mode	Time	Exceptions	
		DEC, INC, Rotate, Shift	Others
Absolute	4	6	JMP=3 JSR=6
Abs,X and ABS,Y	4 (+1 over page)	7	STA=5
Zero Page	3	5	
ZP,X and ZP,Y	4	6	
Implied	2		Stack PH=3, PL=4 RTS=6 RTI=6 BRK=7
Immediate	2		
Relative	2 (if no branch) 3 (if branch taken +1 over page)		
Accumulator	2		
(Ind,X)	6		
(Ind),Y	5 (+1 over page)		STA=6
Indirect	5		

In practice, it is impossible to time long programs by timing individual instructions, since there are too many instructions to count. But it's helpful in speeding up slow ML routines, when you're trying to optimize functions like updating a screenful of information.

6502 Techniques and Methods

This section deals with the following topics:

- **Two-byte operations,** including incrementing, decrementing, adding, subtracting, multiplying, dividing, and comparing.
- **Testing for a range of data.**
- **Loops.**
- **Shift and rotate instructions.**
- **Logical instructions,** including AND, ORA, EOR, and BIT.
- **Twos complement arithmetic**.
- **Decimal arithmetic**.
- **Debugging ML.**

Two-Byte Operations

Incrementing two bytes. The best method to increment two bytes is illustrated by the following routine:

```
       INC LOBYTE
       BNE CONT   ;BRANCH UNLESS FF JUST BECAME 00
       INC HIBYTE ;ONLY NEEDED WHEN LOBYTE NOW IS 00
CONT ...
```

Decrementing two bytes. There's no test for decrement from $00 to $FF, so this is less simple than incrementing. However, the following routine will do it:

```
     LDA LOBYTE
     BNE DECL    ;BRANCH UNLESS LOBYTE IS 00
     DEC HIBYTE  ;ONLY NEEDED WHEN LOBYTE WAS 00
DECL DEC LOBYTE
```

Adding two-byte pairs. The carry flag carries from low to high bytes.

```
CLC      ;START BY CLEARING CARRY
LDA  LO1 ;GET FIRST LOW BYTE ...
ADC  LO2 ;...ADD IT TO OTHER LOW BYTE
STA  LO2 ;AND STORE RESULT
LDA  HI1 ;GET FIRST HIGH BYTE ...
ADC  HI2 ;...ADD IT AND CARRY TO OTHER HIGH BYTE,
STA  HI2 ;AND STORE RESULT
```

In this example, LO2 and HI2 end up with the contents of LO1 and HI1 added to them. Chapter 10 has another example.

Subtracting two-byte pairs. The carry flag is set before subtraction (if it's clear the result will be off by 1). If C is clear on exit, the result is negative—that is, the amount subtracted was larger than the original two-byte amount.

```
SEC      ;SET CARRY FLAG
LDA  LO1 ;GET FIRST LOW BYTE...
SBC  LO2 ;SUBTRACT OTHER LOW BYTE
```

```
STA  LO2 ;STORE RESULT'S LOW BYTE
LDA  HI1 ;GET FIRST HIGH BYTE...
SBC  HI2 ;SUBTRACT OTHER HIGH BYTE AND CARRY FLAG COMPLEMENT
STA  HI2 ;STORE HIGH BYTE OF RESULT.
```

Multiplying two single bytes to give a two-byte result. Multiply and divide instructions don't appear on the 6502. The example multiplies the contents of two zero page locations ($FC and $FD), leaving the result in the same two bytes. On average, about 6000 multiplications can be performed per second by this routine, which uses ROR (ROtate Right) to detect bits and to store the result in FC (low byte) and FD (high byte). This example is formatted for the BASIC monitor program presented with this chapter. With *VICMON* or a similar monitor, remember to add $ to all the numbers and to change the RTS to BRK.

```
033C  CLC
033D  LDA  #00
033F  LDX  #08
0341  ROR
0342  ROR  FC
0344  BCC  0349
0346  CLC
0347  ADC  FD
0349  DEX
034A  BPL  0341
034C  STA  FD
034E  RTS
```

The one-line program 10 INPUTX,Y: POKE 252,X: POKE 253,Y: SYS 828: PRINT PEEK(252)+256*PEEK(253) can be used to test this. It is possible to test this routine exhaustively, something practically unattainable with more complex ML.

Division of a two-byte number by a single byte. The next routine is roughly the opposite of the previous one. A 16-bit (two-byte) number in locations $FC (low byte) and $FD (high byte) is divided by the contents of $FE, and the result (assumed to be in the range $00–$FF) is left in $FC, with the remainder in $FD. The identical addresses and locations need not be retained in actual programs, of course. This example is formatted for the BASIC monitor program presented with this chapter. With *VICMON* or a similar monitor, remember to add $ to all the numbers and to change the RTS to BRK.

```
033C  CLC
033D  LDX  #08
033F  LDA  FD
0341  ROL  FC
0343  ROL
0344  BCS  034A
0346  CMP  FE
0348  BCC  034D
034A  SBC  FE
034C  SEC
034D  DEX
034E  BNE  0341
0350  ROL  FC
```

```
0352  STA  FD
0354  RTS
```

This can be tested from BASIC by POKEing locations 252 and 253 with low and high bytes of the numerator, POKEing 254 with the denominator, SYS 828, and printing PEEK(252) and PEEK(253) for the solution and remainder.

Comparing two-byte pairs. The trick is to avoid comparison instructions and use SBC instead, which retains results as well as setting flags. Use the following routine:

```
SEC
LDA  LO1
SBC  LO2
STA  TEMP   ;TEMPORARY STORE
LDA  HI1
SBC  HI2
ORA  TEMP   ;RESULT 0 ONLY IF A AND TEMP BOTH ZERO
```

Z is set if the contents of the first address equaled those of the second; C is clear if the contents of the first were less than the second. BEQ, BCC, and BCS test for =, <, and > respectively.

Other two-byte operations. It's often possible to write compact ML using the X and Y registers to store two bytes. Suppose locations $FD and $FE contain an address to be decremented, then stored in locations $0350 and $0351. You can use the following routine:

```
    LDY  $FE
    LDX  $FD
    BNE  NO
    DEY
NO  DEX
    STY  $0351
    STX  $0350
```

Testing That a Byte Is in the Correct Range

The following example tests whether the byte in the accumulator is within the range 5 to 9. A whole sequence of CMP instructions, with their immediate mode bytes in increasing order, can be tested with a succession of BCC instructions. It's not necessary that the second branch be BCS, as it is here.

```
    CMP  #$05
    BCC  SMALL  ;BRANCH TAKEN IF A=0,1,2,3, OR 4
    CMP  #$0A
    BCS  LARGE  ;BRANCH TAKEN IF A=0A,0B,0C,...,FF
OK  ...         ;CONTINUE WITH A IN DESIRED RANGE
```

Loops

Loops generally use X or Y as a counter and often as an offset too. There's some room for timesaving in the design of loops. Also, it's worth checking over their logic. It's easy to write loops which aren't quite correct, perhaps missing one of the values at one end of the loop.

Look first at a typical small loop. This one puts the five bytes for the letters of the word HELLO on the screen. There are two versions (shown here in the format of the BASIC monitor program):

```
      LDX   #0                LDX   #5
LOOP  LDA   TABLE,X     LOOP  LDA   TABLE-1,X
      STA   1E00,X            STA   1E00,X
      INX                     DEX
      CPX   #5                BNE   LOOP
      BNE   LOOP              RTS
      RTS               TABLE .BYTE 'HELLO'
TABLE .BYTE 'HELLO'
```

In the first version, X successively takes values 0, 1, 2, 3, and 4; in the second, the values taken are 5, 4, 3, 2, and 1. The second version is shorter; DEX counts down to zero, and CPX #$00 is redundant, since in effect the processor does this when it sets Z flag of the status register. For this reason, decrements are often more elegant than increments. However, you should take note that the decrementing version prints OLLEH instead of HELLO. That is, the bytes are read from right to left with this version. You'll need to take that into account when you set up your tables of data. Using decrements also adds an extra difficulty in that the bytes' start is one position away from the address in the LDA instruction, as a consequence of the fact that X is never zero. This explains why the decrementing version uses LDA TABLE−1,X instead of LDA TABLE,X.

Note that LDX #$04/.../BPL LOOP counts X down from 4 to 0, and the accumulator loads from the expected start point. However, X values larger than $7F won't cause a branch on BPL, so it's best to avoid BPL at first.

Looking at longer loops, there are again several possible methods. Suppose 512 bytes are to be moved into color RAM from $1D00. Different approaches are given below.

```
       (a)                 (b)                  (c)

     LDA #$00            LDY #$00             LDY #$00
     STA $FB        LOOP LDA $1D00,Y     LOOP LDA $1D00,Y
     STA $FD             STA $9600,Y          STA $9600,Y
     LDA #$1D            LDA $1E00,Y          INY
     STA $FC             STA $9700,Y          BNE LOOP
     LDA #$96            INY                  INC LOOP+2
     STA $FE             BNE LOOP             INC LOOP+5
     LDY $00                                  LDA LOOP+2
LOOP LDA ($FB),Y                              CMP #$1F
     STA ($FD),Y                              BNE LOOP
     INY
     BNE LOOP
     INC $FC
     INC $FE
     LDA $FE
     CMP #$1F
     BNE LOOP
```

Loop (b) is the shortest (and also the fastest). It moves bytes in pairs. The loop will obviously get longer if several thousand bytes are to be moved, perhaps when ML has been loaded into RAM from tape and needs to be put into its correct RAM area to run.

Loop (c) is basically similar but uses self-modifying ML, which you may find easier to understand than indirect addressing. In the example, the loop becomes LDA 1E00,Y/ STA 9700,Y the second time around, then LDA 1F00,Y/ STA 9800,Y, after which the CMP test terminates the loop. Although this is fairly straightforward, it has the drawback that the ML is different on exit from what it was at the start. Thus, a second call to the ML gives different results.

Loop (a) is a general-purpose version, suitable in most cases; it's longer than the others, because of the need to set up $FB/FC and $FD/FE with $1D00 and $9600.

In each case, these examples assume that the loop ends at an address like $1F00. Obviously both bytes in the address can be compared if this doesn't apply.

Saving the zero page is sometimes a useful trick, perhaps to optimize ML running with BASIC. *VICMON* can do this; TRACE (Chapter 6) does it, too, to allow LIST and BASIC to work together. The routines are simple enough but require an inviolable 256 bytes of RAM (usually at the top of BASIC). Use the following routine to save the area:

```
     LDX  #$00
LOOP LDA  $00,X
     STA  STORE,X
     INX
     BNE  LOOP
```

and use this one to restore it later:

```
     LDX  #$00
LOOP LDA  STORE,X
     STA  $00,X
     INX
     BNE  LOOP
```

Shift and Rotate Instructions

Shift instructions (ASL, LSR) and rotations (ROL, ROR) are useful whenever individual bits are important. For example, an easy way to print a byte as eight 0's or 1's is to shift the byte eight times, using BCC or BCS to determine whether 0 or 1 is correct. Parallel-to-serial interconversion, where a byte is sent as separate bits or put together from bits, uses the same idea.

Because rotations use nine bits, including C, they can be used to hold intermediate results during processing. The multiply and divide routines presented earlier use rotations like this; division, for example, repeatedly doubles the denominator and compares it with the numerator (to see which is bigger) while collecting the result.

Both types of instructions are valuable in calculations, because they multiply by 2. This example shows how to multiply by 22; at the start, $FB is assumed to hold a Y value from 0 to 22, representing a screen row. $FC is assumed to hold #$0F. After the ML executes, the screen position $1E00 + 22*Y is left in $FB/FC, and an X value can be added.

```
LDA   $FB  ;A HOLDS Y VALUE  (0–22)
ASL        ;A=TWICE Y        (0–44)
ASL        ;A=4*Y            (0–88)
ASL        ;A=8*Y            (0–196, C ALWAYS CLEAR)
ADC   $FB  ;A=9*Y            (0–198, C CLEAR)
ADC   $FB  ;A=10*Y           (0–220, C CLEAR)
ADC   $FB  ;A=11*Y           (0–242, C CLEAR)
ASL        ;A=22*Y           (0–484, C MAY BE SET)
ROL   $FC  ;FC NOW HOLDS 1E OR 1F, DEPENDING ON 22*Y
STA   $FB  ;FB HOLDS LOW BYTE OF ADDRESS
```

Logical Instructions

AND and ORA act like BASIC's AND and OR, except that only eight bits are involved. EOR (exclusive OR) doesn't exist in BASIC; the nearest thing is (A OR B) AND NOT (A AND B). As Chapter 11 shows, AND is used to mask out bits, ORA is used to force bits high, and EOR is used to reverse bits. In each case, any combination of bits can be chosen.

For example, assume you have a byte ($72) which you want to print as the digits 7 then 2. Store the byte, then shift it right four times. AND #$0F masks off (erases) the leftmost bytes, then ORA #$30 forces $30 into the byte to create the ASCII value of a numeral ready for printing. Recover the original byte and repeat. Both digits are correctly output.

An example of EOR may be helpful too. EOR combines bits in repeatable patterns (unlike AND and ORA, which eventually set all bits to 0 and 1 when operating on sequences of bytes). You can use this to generate a checksum for BASIC or ML programs, helpful in checking that a program is correct; the following version prints a number from 0 to 255 and will print the same number whenever the identical program loads into the identical memory area.

```
        LDA  $2B     ;COPY START-OF-PROGRAM POINTER
        STA  $FD     ; INTO FD AND FE
        LDA  $2C
        STA  $FE
        LDY  #$00    ;SET Y TO 0
LOOP    EOR  ($FD),Y
        INC  $FD     ;INCREMENT ADDRESS IN FD/FE
        BNE  NOINC
        INC  $FE
NOINC   LDX  $FE     ;TEST WHETHER FE/FF YET EQUALS 2E/2F
        CPX  $2E     ; 2E/2F, THE END-OF-PROGRAM POSITION
        BNE  LOOP
        LDX  $FD
        CPX  $2D
        BNE  LOOP
        TAX          ;END OF PROGRAM. NOW PRINT OUT A'S VALUE
        LDA  #$00
        JMP  $DDCD   ;USING THIS ROM ROUTINE
```

All logical instructions (like the arithmetic instructions ADC and SBC) use the accumulator in their logical expressions; this is what the accumulator is for. EOR

#$FF is the equivalent of NOT A, since all the bits in the accumulator are flipped.

The BIT instruction is rather different from the above three instructions; it sets flags but doesn't alter the accumulator or any address. It may be of use when some location is to be tested logically while the accumulator must remain unchanged. The Z flag is set if the accumulator and the operand of BIT together AND to zero, and bits 6 and 7 of the result are copied into the V and N flags respectively.

Twos Complement Arithmetic

Is it possible to arrange arithmetic in eight-bit bytes to allow for negative values? You've become used to $00–$FF representing 0–255, but you can, with a change in interpretation, allow negatives too. This is not a convention, in the strictest sense, but a consequence of the rules of binary arithmetic. Thus it must work on any microprocessor.

Bit 7 (the leftmost bit) can be regarded as a sign bit. It takes one of two values, with 0 designating a positive number and 1 designating a negative number. The N flag in the status register is wired to be consistent with this; when N=1, the number is deemed negative and BMI's branch is taken. If N=0, BLP is taken. These branches operate whether or not you're using negatives.

A number and its negative must add to zero. It follows that a pair of numbers (say +7 and −7) can be represented by $07 and $F9, because these add to $00, and because the second has its high bit set. The use of numbers like $F9 to represent negatives is called *twos complement arithmetic,* and $F9 is the twos complement of $07. You'll find with experiment that the largest possible one-byte twos complement number (%0111 1111) is 127, and the smallest (%1000 0000) is −128. These figures are identical to the range available to branch instructions and show how a branch's offset can be stored in just one byte.

Subtraction from 256 gives the twos complement. Another rule, which may be easier to use, is to flip all the bits in the byte, and add 1. So, the twos complement of %0101 0101 ($55) is %1010 1010 plus 1, or %1010 1011 ($AB). Again, $55 plus $AB adds to $00, ignoring the carry flag. Note that $00 has no negative complement; it equals its own twos complement.

You can generate twos complement numbers with the following routine:

```
LDA NUMBER
EOR #$FF
CLC
ADC #$01
```

Since the sign can be stored elsewhere, this type of arithmetic isn't particularly popular; however, VIC's BASIC integer variables (for example, X%) use a 16-bit version of twos complement arithmetic in which the highest bit stores the sign, so integers may range from −32768 to 32767. Their twos complements need EOR #$FF on both bytes, plus #$01.

The V flag is also associated with this type of arithmetic, showing that an overflow took place into the sign bit. Consider addition, where V is conditioned by ADC (and four other instructions). Numbers of opposite sign cannot overflow; even extreme values (0−128) must fall in the correct range. But if the signs are the same, overflow is possible. $44 + $33 gives $77, and V is clear; but $63 + $32 gives $95, which in twos complement arithmetic is considered negative, so this addition results in V being set. Similarly, two negative numbers can appear to add to a positive result, and if this is the case, V will also be set.

The condition of the V flag is in fact determined internally, by the chip, by reversing the EOR of the sign bits (giving 0 if they match, 1 otherwise). V is set, in other words, when the signs are the same; the result (incorrectly) shows a different sign.

To reiterate, twos complement is an interpretation. Many programmers may never use it, preferring to work in positive numbers. But you can't understand N and V flags without grasping the idea of negative bytes.

Decimal Arithmetic

Decimal mode arithmetic, with the D flag set, packs two digits into each byte and adds or subtracts in decimal. This example adds a four-digit number in locations $8B (high byte), $8C (low byte) to a six-digit number in locations $8D (high byte), $8E (middle byte), $8F (low byte), leaving the result in the three-byte number. Scoring in games often uses such subroutines; a score is stored in the smaller location, the subroutine called to total, and the result printed.

```
        SED           ;TURN ON BCD MODE
        CLC           ;CLEAR CARRY
        LDA  $8C      ;ADD LOW BYTES,
        ADC  $8F
        STA  $8F      ;STORE RESULT
        LDA  $8B      ;ADD MID BYTES
        ADC  $8E
        STA  $8E      ;AND STORE
        BCC  NOINC
        INC  $8D      ;HIGH BYTE
NOINC   CLD           ;RETURN TO NORMAL MODE
```

The six-digit number can be printed by looping three times to select a byte, then shifting it right, using AND #$0F followed by ORA #$30 to convert to ASCII, outputting with JSR $FFD2, and repeating with the same byte unshifted.

It is often simpler to use individual bytes for totals of this sort. This example uses the first five locations of the unexpanded VIC's screen. It starts by putting zeros there, and then puts scores into $8B through $8F as, for example, 00 00 01 00 00 (to represent 100). The result appears directly on the screen.

```
       LDX  #$04     ;SET COUNTER FOR 4,3,2,1,0
       CLC           ;CLEAR CARRY
LOOP   LDA  $1E00,X  ;LOAD BYTE FROM SCREEN
       ADC  $8B,X    ;ADD CORRESPONDING BYTE
       CMP  #$3A     ;IS RESULT 10 OR MORE?
       BCC  CLEAR    ;IF NOT, BRANCH,
       SBC  #$0A     ;IF SO, SUBTRACT 10, LEAVING CARRY SET
CLEAR  STA  $1E00,X  ;UPDATE SCREEN BYTE
       DEX           ;COUNT DOWN TO NEXT BYTE
       BPL  LOOP     ;BRANCH UNTIL X IS FF
```

This is very fast and efficient. Change the value of X for more or fewer than six digits. This method does not use BCD mode.

Debugging ML Programs

Listed here are many errors common in 6502 ML programming. Program design should be approached methodically, preferably from the top down, starting with the writing or reusing of standard subroutines. Careful analysis of the code, perhaps with flow charting, and testing with typical and abnormal data should insure a sound program.

Careless errors. Careless errors may remain undetected for a long time. Examples include transcription errors (entering 7038 for 703B) and immediate mode # errors (using LDA 00 instead of LDA #00). You might also use a wrong ROM address, perhaps one for a different computer, or make branch errors, especially with simple assemblers where forward addresses must be reentered. Yet another possibility is the use of a subroutine which alters A, X, or Y.

Addressing mode errors. These stem from confusion of order of low and high address bytes, failure to understand indirect addressing modes, or attempted use of indexed zero page addressing to extend above location $FF (LDA $AB,X always loads from zero page, for any X). Indirect jumps may also cause problems; JMP ($03FF) takes its address from $03FF and $0300.

Calculation errors. With addition, subtraction, and so on, do not forget to use the proper instruction (CLC before adding, SEC before subtracting, and SED and CLD with BCD math). Remember, too, that LDA #$02 followed by ADC $FD adds the contents of location $FD to the 2 in the accumulator, but leaves $FD unchanged. It's easy to forget that only the accumulator holds the result, and STA $FD may be needed to return the answer to the desired location.

You should also be careful to keep track of the carry bit with shifts and rotates. That can be tricky; C is easy to overwrite.

Status flag errors. The logic behind the flags' settings may cause difficulties for beginners, who may not realize (for example) that AND #$00 is identical to LDA #$00. Incrementing from a value of $7F to $80 sets the negative flag. The following routine

```
LDA  KEY
CMP  #$3A
BNE  ERROR
STA  LOCN
```

stores the contents of KEY in LOCN, but STA sets no flags. You might expect it to clear Z, but this is not the case. Z will remain set until cleared by the execution of some instruction which affects that flag.

Stack errors. Generally, the number of stack pushes should equal the number of pulls; additionally, the order should match. PHA/ TXA/ PHA, for example, usually requires PLA/ TAX/ PLA to retrieve A and X later on.

Errors in which RAM is overwritten. Programs or their data can be overwritten by BASIC strings or variables, by tape activity, or by subroutines which happen to access similar RAM areas or zero page locations (including utilities like *VICMON*), to name but a few. The program itself may be at fault: A loop may move some data it shouldn't, a pointer may be updated while still in use so it points temporarily to a wrong address, or a part of the stack may be used for storage but get filled by normal stack activity.

Machine Language Monitors with VIC-20

VICMON

Commodore's VIC-1213 cartridge, called *VICMON*, is a popular monitor. *VICMON* occupies 4K, usually from $6000 to $6FFF. Some *VICMON* cartridges were produced in which the ROM occupies locations $A000–$AFFF. SYS 6*4069 (or SYS 40960, in the case of ROM at $A000) turns it on. It inevitably uses RAM, almost all of it in the zero page.

Syntax of commands. *VICMON* command syntax is more involved than BASIC. First, each line starts with a period; the function of this is to verify that a line is intended as input to the monitor. Other prompts, for use internally by the monitor, appear after the period, the colon (when altering memory with M), the semicolon (when altering registers with R), the comma (when disassembling), and the single quote (when used with the interpret command). Generally, these are handled automatically.

VICMON commands usually consist of a single letter (A, B, C, D, E, F, G, H, I, L, M, N, Q, R, S, T, W, or X) followed by some byte pattern relevant to the command. The routine which searches for one of these letters is in ROM; there's no easy way to add extra commands, as you can see by disassembling. Two commands—RB and J—are patched into other routines and are exceptional.

As an example, .T 1800 1850 033C copies the contents of $1800 through $1850 into the area starting at $033C, therefore filling $033C through $038C. The pattern of addresses must be entered correctly. There is no error indication if a mistake is made, but nothing happens. Some mistakes, however, do indicate that there has been an error; one of them is Y, a nonexistent command.

The parameters to be input are accepted by absolute position. Punctuation is irrelevant. Thus, .T 1800,1850,033C produces exactly the same results as the earlier form, but commas aren't compulsory.

Most command inputs are accepted up to, but not beyond, a colon. For example, .A 033C CLC: GARBAGE is treated as .A 033C CLC and assembled correctly. This is often a timesaver.

Screen handling. Two things are noteworthy here. First, *VICMON* uses whatever line lengths already exist; therefore, spacing is liable to be erratic until a screenful of data has been processed. Second, many commands scroll back, if cursor up is held. The screen is assumed to be at $1E00. With memory expansion in place, this feature won't work. Scrolling up also affects BASIC.

Exit from ML programs. BRK as the last instruction returns to *VICMON;* *VICMON*'s X command returns to BASIC. Any BASIC in memory is prone to pointer alteration by *VICMON*. Type POKE 43,1 to correct this. You may need to POKE 44,16 and, usually, POKE 16*256,0 too if you've used trace commands. Alternatively, tuck away BASIC's zero page with .E 1D00 (for the unexpanded VIC when you aren't using that area); X restores BASIC, and the cursor position, to its entry position. It's also true that RTS returns to BASIC, but in an irregular way. Reserve RTS for subroutines to be tested from BASIC.

Running ML programs. .G 033C runs the ML program starting at $033C from within *VICMON*. This is usually satisfactory, but there may be zero page clashes between your routines and *VICMON*. For instance, JSR $DDCD (a ROM routine which outputs a number) won't give the correct result within *VICMON*, but when called by SYS from BASIC it runs fine.

Programs can be run from *VICMON* under trace control, so, in principle, each instruction can be carried out singly and the results checked.

What to do if ML crashes. If your ML program goes into some apparently endless loop, from which it looks like there is no escape short of turning off the computer, there are two things that you can try. STOP-RESTORE will often work. If it doesn't, the only recovery procedure is an external reset (see Chapter 5). Provided the ML is in RAM above $0400, it will be completely unaltered by this process.

To help avoid this, fill RAM with zero bytes (using the F command) to increase the chance that a wrong instruction will end on BRK. In addition, check that all branch instruction destinations are valid.

Using *VICMON*'s Walk and Quick Trace. The W command lets you "walk" (single step) through ML. It works reliably. However, it lists each line of ML onscreen, so graphics programs usually can't be walked.

To get around this problem, Quick Trace was introduced. But it is not reliable; if you have no reset switch, you should probably avoid Q altogether. Otherwise, you may lose your program. The reasons include zero page clashes, if your program uses locations $FD and $FE, for example.

Walk's J command (to skip subroutines, avoiding long detours) also has bugs. G, Q, and W share most of their ML; a flag ($00, $40, or $80) in location $12 distinguishes them.

Notes on using *VICMON*. When using *VICMON*, it's best to stick to tried and tested commands. Assemble and Disassemble (along with Load and Save) are probably the most useful.

Using *VICMON* with BASIC, although tricky, is perfectly feasible. A reset switch is useful; if you have one, put your ML above $0400 (not in $033C and following, which reset erases). You should also avoid working with BASIC in RAM, because you'll have to lower its memory top to preserve your ML. Note that SYS 64802 will clean everything up, provided the ML is located above $0400, so if things are going wrong, exit to BASIC and use the SYS statement.

If you're a beginner, use *VICMON* to assemble the programs at the start of this chapter. Run them with G, and disassemble again with D; you'll soon get the feel of it.

VICMON Commands

A (Assemble)

Assemble is a miniassembler, which converts instructions into the correct form (inferring addressing mode from format) and stores bytes into memory. There's a read-back check in case RAM isn't there. Labels and other features of true assemblers aren't accepted. Press RETURN to leave A mode. Note that errors also exit from A. The entry ??? is accepted and treated as byte #$02, so be careful when disassembling not to press RETURN over ??? or data may be corrupted.

Examples of the use of this command include .A 033C LDA #$00 and .A 033E STA #1E00.

B (Breakpoint)

Breakpoint allows ML to be stopped at a predetermined point, without running its full extent. This is a valuable feature when used with G (and Q). *VICMON* runs the ML and checks for equality with the breakpoint, which is stored in locations $10 and $11; a breakpoint therefore slows program running.

For example, B 0343 000A puts a breakpoint at 0343, ready for G 033C or Q 033C. The second parameter stores the number of times (10) which the breakpoint will be ignored; this is useful for testing loops. The default B 0343 breaks the first time round. The maximum number of waits is $FFFF, enough for most purposes.

A breakpoint must be set to coincide with an opcode, not with data or tables, or it will be missed. Note that only one breakpoint can be set at any one time.

RB (remove breakpoints) is not reliable. Use B 8000 instead, which relocates the breakpoint into ROM, where it will never apply.

C (Compare Memory)

This command reports any differences between two areas of memory. Syntax is identical to T. For example, C 1000 10FF 033C checks whether the contents of locations $1000–$10FF match the contents of $033C–$043B and prints the address of any discrepancies.

D (Disassemble)

Disassemble is a standard 6502 disassembler, including $ and # symbols, and compatible with the assembler. D 033C then RETURN starts disassembly; cursor down or up continues. D 033C 0344 disassembles a range. Bytes which cannot be interpreted as opcodes disassemble as ???; their actual contents can be read with the M command.

When scrolling back, disassembly gives priority to longer opcodes, so it can give results different from normal disassembly. However, this is impossible to avoid.

E (Enable Virtual Zero Page)

This command saves a copy of the current zero page. With BASIC, use of this just after entering *VICMON* allows X to operate correctly most of the time. The most common location for the virtual page on the unexpanded VIC is just below screen memory, so you would use E 1D00.

It's not necessary to use this command; with Walk or Quick Trace, this stored page is swapped back and forth provided your program doesn't use locations $FE and $FF. Thus, if your ML uses other zero page locations and you wish to try W or Q, use E 1D00. E 0000 has the effect of turning this command off.

F (Fill Memory)

As you might expect, this command fills a region of RAM with identical bytes. For example, F 033C 03FF 00 fills the tape buffer with zero bytes. This has no syntax or read-back checking, so if you enter it incorrectly nothing will happen. The byte $EA, equivalent to a NOP instruction, is another useful filler.

G (Go Run)

Go Run executes ML from *VICMON*. G DDF9, for example, goes to a BRK instruction in ROM and immediately stops. G alone goes to the address currently in the program counter (which you can change with the R command), but there's no advantage in using it.

Once started with G, execution continues until interrupted by BRK (which returns to *VICMON)* or RTS (a bad exit to BASIC), unless B has set a breakpoint, in which case it terminates there. The combination of G and B is a very useful one for program debugging.

H (Hunt Memory)

This command reports all instances of a byte combination or string of characters between two addresses. For example, H C000 FFFF 00 90 prints all occasions in ROM where address $9000 is referenced. Another example, H C000 FFFF 'BASIC, prints all appearances in ROM of the word BASIC.

Be careful when interpreting the results of H. For instance, the pattern 20 E4 FF is almost certainly JSR $FFE4, but a hunt for 0F 90 to find a reference to address $900F may yield nothing since 9000,X may be used to address that location. Addresses of branches cannot be picked up, and so on.

I (Interpret Memory)

The I command prints the contents of a range of memory locations, displaying the bytes as ASCII characters. The output lines are formatted as an address followed by 12 characters. For instance, I C09E C100 prints some BASIC keywords from ROM. To take another example, I C09E prints a single line; cursor down (or up) for more.

J

See W.

L (Load ML)

L has syntax L "NAME",01 from tape or L "NAME",08 from disk. Abbreviations are accepted. For example, L "",01 loads the next tape program; L "N*",08 loads the first disk program beginning with N. In either case, the program or data is loaded as an image. After loading it's not altered in any way.

M (Memory Display)

This command displays the contents of a range of memory locations, formatted as an address followed by five hex bytes. M 033C 0350 displays bytes from the tape buffer, while M 033C displays a single line of five bytes. Again, use cursor down or up for more.

N (Number Adjuster)

N is used after transferring ML with T. It's an intelligent routine which relocates numbers, adjusting subroutine calls and absolute addresses within the original ML so they also apply to the shifted ML. This command is not particularly user-friendly.

Chapter 9 shows how to use it to relocate *VICMON* itself. To move the contents of $6000–$6FFF to $2000–$2FFF, you could use T 6000 6FFF 2000, which transfers the bytes unchanged, provided there's RAM at $2000–$2FFF.

N 2000 2FFF C000 6000 6FFF searches $2000–$2FFF for all references to addresses between $6000 and $6FFF, which the original ML would have used. It alters them by adding $C000. This is because $6000+$C000=$2000 (overflow is ignored). You have to calculate the offset yourself. The first instruction in *VICMON* is 6000 JMP $600C. After Transfer, you have 2000 JMP $600C. Then N converts it to 2000 JMP $200C.

N 2E7F 2EB5 C000 6000 6FFF W is an alternative option. This searches for word tables, which have nothing to do with words but are simply tables of addresses (like *VICMON*'s at $6E7F) which store the addresses of each *VICMON* command. These tables are updated just as the instructions' addresses were.

N works best with straightforward ML, starting at the beginning, with tables of addresses collected at the end. Embedded ASCII and other data, which won't disassemble meaningfully, should be avoided.

P (Printer)

VICMON has no special printer commands. Use OPEN 4,4: CMD 4: SYS 6*4096 and type commands blind.

Q (Quick Trace)

Despite its name, Q is a relatively slow way to run ML; apart from this it's nearly identical to G. Both use breakpoints. Q, however, can be stopped (by hitting STOP and X) during its run. In addition, Q enters Walk mode, rather than displaying registers with R, when it finishes.

Always use E before Q. STOP-X will not work with an X2 crash; only RESET can help.

Suppose you have some ML starting at $033C, and ending at $036B with BRK, which prints text to the screen. If G displays it too fast, and W spoils the layout, try E 1D00 (to set up the virtual zero page), B 036B (to set a breakpoint at the end), and Q 033C (to quick trace).

The program runs more slowly than usual, ending with BRK as Walk mode is entered. STOP now exits; cursor down would continue walking through ML.

R (Registers)

R displays the contents of the program counter, status register, A, X, Y, and stack pointer as they were on entry to the monitor. Only PC can be altered; if it is, G on its own has the same effect as G followed by PC. R is mainly used to check that the registers hold correct values; G 033C with 033C LDX #99/ BRK should show that the X register contains 99.

RB (Remove Breakpoints)
See B.

S (Save ML)
S has syntax SAVE "ML",01,6000,7000 (for tape) and SAVE "ML",08,6000,7000 (for disk). It is essential to specify the range of addresses to be saved; these examples save from $6000 through $6FFF, without saving the byte at $7000. Note there's no error warning when saving to disk if the drive is off, so check to be certain that your drive is turned on.

T (Transfer Memory)
T copies a block of memory; the syntax is identical to C. For example, T 1E00 2000 1E01 moves a screenful of bytes by one position. The endpoint of the new block is implicit in the three parameters. The original block is not altered unless the transfer overlays part of the original. See also N for relocation of programs.

V (Verify)
VICMON has no built-in VERIFY, but BASIC's VERIFY can be used as follows: S "TEST",08,A000,C000, then X to BASIC, then VERIFY "TEST",8,1.

W (Walk)
W single-steps through ML, listing each ML instruction onscreen. It does this by setting a timer to interrupt just as the next instruction starts; the interrupt routine disassembles a single instruction from the program counter's address so ML cannot run more than one instruction.

For example, W 033C single-steps from 033C. RETURN and several other keys single step; the space bar and other keys repeat. STOP exits to normal *VICMON* mode.

When a subroutine (for example, JSR $FFE4) is encountered, pressing the J key is intended to execute it without the tedious detour of listing it line by line. However, this feature isn't entirely reliable.

X (Exit to BASIC)
Used without parameters, X recovers a virtual zero page and returns to BASIC. See the earlier notes.

Other Monitors Written in ML
"Tinymon" fits any VIC; it is modified from CBM monitors and has R, M, G, X, L, and S commands, without A or D.

"Super VIC Mon" is CBM's "Supermon" modified for VIC. It fits any VIC and has most of *VICMON*'s commands, excluding W, Q, and N.

"VIC Micromon" requires memory expansion. It offers more commands than *VICMON*, including number conversion. However, its mnemonics are much like those of *VICMON*, so it's similar to *VICMON* in use.

Monitors in BASIC

BASIC monitors, although slow, have some advantages. They are easier for beginners to use, since they have familiar INPUT commands, and they can be loaded, run, and listed without difficulty. They are also much easier to modify than ML monitors; if you want printer disassembly, the necessary programming is relatively easy to put in. And they make the processes of disassembly and assembly easier to understand; studying the BASIC program is far easier than deciphering ML.

Program 7-1 is a BASIC monitor. The complete program will not fit an unexpanded VIC. However, either the assembly or the disassembly part will fit, so it's possible to run either part alone. The program runs on the 64 too. To use just the assembly feature, type only lines 1, 100, 300, 800–950, and 4000–8000; also, add a new line: 5 GOTO 4000. For disassembly only, use lines 6–680, 2000–3090, and 5000–8000.

Disassembly requires that an opcode be PEEKed and printed out, followed perhaps by an address, before disassembling the next instruction. For example, a decimal value of 32 is the JSR instruction, so it's followed by the next two bytes in reverse order. A loop in BASIC finds the opcode (or prints ??? if the number found doesn't match any valid opcode) and its addressing mode and outputs the result.

Assembly is trickier. This version infers the addressing mode from the format—for example, treating LDA 1234,X as absolute indexed while rejecting PQR or LDA #1234. Some programs use nonstandard opcodes like LDAIM to help ease this problem.

Disassembly to a printer is more complete than disassembly to the screen, because there's no limit of 22 characters when using a printer. Figure 7-1 shows a typical printer output.

Program 7-1. A BASIC Monitor

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
0 REM *** VIC-20/C64 TINY ASSEMBLER/ DISASSEMBLER
  {SPACE}***                                  :rem 236
1 GOSUB 5500 : REM READ OPCODE DATA           :rem 137
2 PRINT"{CLR}":PRINT"{3 DOWN}{2 RIGHT}{RVS}A{OFF}S
  SEMBLE":PRINT"{3 DOWN}{2 RIGHT}{RVS}D{OFF}ISASSE
  MBLE":PRINT"{3 DOWN}"                       :rem 132
3 INPUTA$:PRINT"{CLR}"                        :rem 198
4 IF A$="A" GOTO 4000                         :rem 231
6 GOTO 2000: REM DISASSEMBLE IS DEFAULT        :rem 28
100 L=L/2↑(4*N-4):FORJ=1TON:L%=L:PRINTCHR$(48+L%-(
    L%>9)*7);:L=16*(L-L%):NEXT:RETURN         :rem 198
300 L=0:FORJ=1TOLEN(L$):L%=ASC(MID$(L$,J)):L=16*L+
    L%-48+(L%>64)*7:NEXT:RETURN                :rem 11
400 FORK=2TO1STEP-1:L=PEEK(CA+K):N=2:GOSUB100:NEXT
    :RETURN                                   :rem 171
500 L=PEEK(CA+1):N=2:GOTO100                  :rem 111
600 PRINT"(";:GOSUB400:PRINT")";:RETURN       :rem 161
610 GOSUB500:PRINT",Y";:RETURN                :rem 145
620 PRINT"(";:GOSUB500:PRINT",X)";:RETURN      :rem 40
```

```
630 PRINT"(";:GOSUB500:PRINT"),Y";:RETURN   :rem 42
640 L=PEEK(CA+1):IFL>127THENL=L-256         :rem 52
642 L=CA+2+L:N=4:GOTO100                   :rem 122
650 GOSUB400:PRINT",Y";:RETURN             :rem 148
660 PRINT"#";:GOTO500                      :rem 211
670 GOSUB400:PRINT",X";:RETURN             :rem 149
680 GOSUB500:PRINT",X";:RETURN             :rem 151
800 ONLEN(AS$)-2GOTO810,4015,4015,820,830,840,4015
    ,850                                    :rem 78
810 L$="L":RETURN                          :rem 240
820 L$="J":RETURN                          :rem 239
830 L$="G":RETURN                          :rem 237
840 IFRIGHT$(AS$,1)="X"THENL$="I":RETURN   :rem 143
842 IFRIGHT$(AS$,1)="Y"THENL$="B":RETURN   :rem 139
844 IFLEFT$(AS$,1)="B"ANDMID$(AS$,2,1)<>"I"THENL$=
    "E":RETURN                             :rem 194
846 L$="K":RETURN                          :rem 248
850 IFRIGHT$(AS$,1)="X"THENL$="H":RETURN   :rem 143
852 IFRIGHT$(AS$,1)=")"GOTO860             :rem 183
854 IFMID$(AS$,8,1)=")"THENL$="D":RETURN    :rem 32
856 L$="F":RETURN                          :rem 244
860 IF MID$(AS$,9,1)="X"THENL$="C":RETURN   :rem 76
862 L$="A":RETURN                          :rem 236
900 P=6:L=4:GOTO950                        :rem 100
905 P=5:L=2:GOTO950                        :rem 102
910 P=6:L=2:GOTO950                         :rem 99
920 P=5:L=4:GOSUB950:N=4:GOSUB300          :rem 242
921 L=L-CA-2:IFL>127ORL<-128GOTO4015        :rem 98
922 IFL<0THENL=L+256                       :rem 176
923 RETURN                                 :rem 126
925 P=5:L=4:GOTO950                        :rem 106
950 L$=MID$(AS$,P,L):RETURN                 :rem 96
2000 GOSUB5500                              :rem 12
2010 INPUT"DISASSEMBLE AT";L$                :rem 3
2020 INPUT"DEVICE#";D:IFD<>3THENOPEND,D:CMDD
                                            :rem 11
2030 GOSUB300:CA=L                          :rem 31
3000 L=CA:N=4:IFD=4THENPRINT"{9 SPACES}";  :rem 130
3005 GOSUB100:PRINT" ";:P=PEEK(CA):RESTORE
     {5 SPACES}                             :rem 62
3015 FORJ=0TO150:READOP:IFOP<PTHENNEXT     :rem 247
3020 IFOP=PGOTO3030                        :rem 127
3025 NB=1:O$="???":M=0:GOTO3053            :rem 229
3030 IFOP<100THENM=ASC(MID$(OP$,4*J+4,1))-64:O$=MI
     D$(OP$,4*J+1,3):GOTO3045              :rem 165
3035 IFOP<200THENM=ASC(MID$(OQ$,4*J-220,1))-64:O$=
     MID$(OQ$,4*J-223,3):GOTO3045          :rem 119
3040 M=ASC(MID$(OS$,4*J-468,1))-64:O$=MID$(OS$,4*J
     -471,3)                                :rem 32
3045 NB=2:IFM=1ORM=6ORM=8ORM=11THENNB=3    :rem 207
3050 IFM=12THENNB=1                        :rem 113
```

```
3053 IFD=3GOTO3080                              :rem 18
3055 FORK=0TONB-1:PRINT" ";:L=PEEK(CA+K):N=2:GOSUB
     100:NEXT                                   :rem 49
3060 PRINTLEFT$("{8 SPACES}",10-3*NB);         :rem 28
3065 FORK=0TONB-1:L=PEEK(CA+K):IFL<32OR(L>127ANDL<
     160)THENL=32                               :rem 12
3070 PRINTCHR$(L);:NEXT                        :rem 169
3075 FORJ=NBTO3:PRINT" ";:NEXT                  :rem 98
3080 PRINTO$" ";                                :rem 74
3085 ONMGOSUB600,610,620,630,640,650,660,670,680,5
     00,400                                    :rem 133
3090 PRINT:CA=CA+NB:GOTO3000                   :rem 201
4000 INPUT"ASSEMBLE FROM";L$                   :rem 195
4005 N=LEN(L$):GOSUB300:CA=L                   :rem 136
4010 L=CA:N=4:GOSUB100                          :rem 22
4015 POKE631,34:POKE198,1:INPUTAS$              :rem 16
4020 CO$=LEFT$(AS$,3)                          :rem 112
4030 GOSUB800:CO$=CO$+L$:RESTORE               :rem 187
4040 FORJ=0TO55:IFMID$(OP$,4*J+1,4)<>CO$THEN NEXT:
     GOTO4050                                  :rem 184
4045 FORK=0TOJ:READOP:NEXT:GOTO4500              :rem 3
4050 FORJ=0TO61:IFMID$(OQ$,4*J+1,4)<>CO$THEN NEXT:
     GOTO4060                                  :rem 184
4055 FORK=0TO56+J:READOP:NEXT:GOTO4500         :rem 154
4060 FORJ=0TO61:IFMID$(OS$,4*J+1,4)<>CO$THEN NEXT:
     GOTO4015                                  :rem 187
4065 FORK=0TO118+J:READOP:NEXT                 :rem 142
4499 REM *** OP IS OPCODE VALUE; ADDRESSING MODE I
     S HELD IN RIGHT OF CO$                     :rem 11
4500 NB=2:L$=RIGHT$(CO$,1)                     :rem 181
4510 IFL$="A"ORL$="F"ORL$="H"ORL$="K"THEN NB=3
                                                :rem 73
4520 IFL$="L"THEN NB=1                         :rem 196
4530 POKECA,OP:IF NB=1 GOTO 4900                :rem 23
4540 IFL$="E"THENGOSUB920:POKECA+1,L:GOTO 4900
                                               :rem 221
4550 ONASC(L$)-64GOSUB900,905,910,910,920,925,910,
     925,905,905,925                           :rem 142
4560 IF NB=2 THEN N=2:GOSUB300:POKECA+1,L   :rem 87
4570 IF NB=3 THEN N=4:GOSUB300:POKECA+1,L-INT(L/25
     6)*256:POKECA+2,L/256                      :rem 49
4900 CA=CA+NB: GOTO 4010                         :rem 5
4999 REM *** DATA HOLDS DECIMAL VALUES OF LEGAL OP
     CODES                                     :rem 155
5000 DATA0,1,5,6,8,9,10,13,14,16,17,21,22,24
                                               :rem 126
5010 DATA25,29,30,32,33,36,37,38,40,41,42,44,45
                                                :rem 46
5020 DATA46,48,49,53,54,56,57,61,62,64,65,69,70,72
                                               :rem 235
```

```
5030 DATA73,74,76,77,78,80,81,85,86,88,89,93
                                         :rem 222
5040 DATA94,96,97,101,102,104,105,106,108,109,110
                                         :rem 147
5050 DATA112,113,117,118,120,121,125,126,129,132,1
     33,134                              :rem 221
5060 DATA136,138,140,141,142,144,145,148,149,150,1
     52,153                              :rem 246
5070 DATA154,157,160,161,162,164,165,166,168,169,1
     70,172,173,174                      :rem 157
5080 DATA176,177,180,181,182,184,185,186,188,189,1
     90,192                              :rem 41
5090 DATA193,196,197,198,200,201,202,204,205,206,2
     08,209                              :rem 255
5100 DATA213,214,216,217,221,222,224,225,228,229,2
     30,232                              :rem 229
5110 DATA233,234,236,237,238,240,241,245,246,248,2
     49,253,254                          :rem 204
5499 REM *** STRINGS FOLLOWING ARE SET UP AT START
      OF PROGRAM                         :rem 81
5500 OP$="BRKLORACORAJASLJPHPLORAGASLLORAKASLKBPLE
     ORADORAIASLICLCL                    :rem 46
5510 OP$=OP$+"ORAFORAHASLHJSRKANDCBITJANDJROLJPLPL
     ANDGROLLBITKANDK                    :rem 245
5520 OP$=OP$+"ROLKBMIEANDDANDIROLISECLANDFANDHROLH
     RTILEORCEORJLSRJPHAL                :rem 30
5530 OP$=OP$+"EORGLSRLJMPKEORKLSRKBVCEEORDEORILSRI
     CLILEORFEORH                        :rem 23
5540 OP$=OP$+"LSRHRTSLADCC               :rem 103
6000 OQ$="ADCJRORJPLALADCGRORLJMPAADCKRORKBVSEADCD
     ADCIRORISEILADCF                    :rem 236
6010 OQ$=OQ$+"ADCHRORHSTACSTYJSTAJSTXJDEYLTXALSTYK
     STAKSTXKBCCE                        :rem 80
6020 OQ$=OQ$+"STADSTYISTAISTXBTYALSTAFTXSLSTAHLDYG
     LDAC                                :rem 232
6030 OQ$=OQ$+"LDXGLDYJLDAJLDXJTAYLLDAGTAXLLDYKLDAK
     LDXK                                :rem 152
6040 OQ$=OQ$+"BCSELDADLDYILDAILDXBCLVLLDAFTSXLLDYH
     LDAHLDXFCPYG                        :rem 194
6050 OQ$=OQ$+"CMPCCPYJCMPJDECJ           :rem 118
7000 OS$="INYLCMPGDEXLCPYKCMPKDECKBNEECMPDCMPIDECI
     CLDLCMPFCMPHDECH                    :rem 242
7010 OS$=OS$+"CPXGSBCCCPXJSBCJINCJINXLSBCGNOPLCPXK
     SBCKINCKBEQE                        :rem 212
7020 OS$=OS$+"SBCDSBCIINCISEDLSBCFSBCHINCH:rem 199
8000 RETURN                              :rem 168
```

Figure 7-1. Disassembly to a Printer

```
D367  Ø6 71      ●   ASL 71
D369  26 72     &_   ROL 72
D36B  9Ø ØB         BCC D378
D36D  18            CLC
D36E  8A            TXA
D36F  65 28     ¯ (  ADC 28
D371  AA         ▌   TAX
D372  98            TYA
D373  65 29     ¯ )  ADC 29
D375  A8        ░    TAY
D376  BØ 93      ┌   BCS D3ØB
D378  C6 5D     _ ]   DEC 5D
D37A  DØ E3     ┐ ¯   BNE D35F
D37C  6Ø        —    RTS
D37D  A5 ØD     |    LDA ØD
D37F  FØ Ø3      ┌   BEQ D384
```

Symbolic Assemblers

You have seen how much easier it is to use mnemonics than the fundamental eight-bit system of the 6502. A full assembler carries this process further, allowing ML to be represented in a completely symbolic, algebraic way. In fact, you have seen symbolic notation in some of the examples; LOOP STA SCREEN,X/INX/BNE LOOP illustrates this, where LOOP is a label and SCREEN is an address or word.

ML in this form is called *source code*. A program called the assembler converts source code into object code, which is identical to ML generated without an assembler.

Source code must include *equates* commands, like SCREEN=$1E00, in the program, and typically also has a comment after each instruction, to document the program. Therefore, source code is usually much longer than object code, at least 20 times as long.

But the great advantage of source code is the fact that it can be edited. There's no problem inserting extra instructions in the middle of a program, because assembly simply recalculates all the addresses and branches. In contrast, *VICMON* users have to shift parts of the program, alter addresses, and generally rewrite and recheck.

VIC's small memory makes it an unlikely machine for assemblers. The closest available programs are on ROMs and use BASIC's editing facilities, so a line of source code might be 100 LOOP JSR GETCHR; GET FROM KEYBOARD. Most such assemblers assume ML will be put into the tape buffer and start assembling at $033C unless specifically told otherwise.

There is no standard assembler for VIC, but the interested user should try his or her hand with an assembler using BASIC line-editing. You will probably find that the information supplied with these systems is not very complete; you may also discover that the assemblers themselves may have quirks. Finally, note that it is possible to assemble for VIC on the 64, though you can only test the object code on VIC.

Chapter 8

ML Methods Specific to the VIC-20

ML Methods Specific to the VIC-20

This chapter shows how to program using the built-in features of VIC BASIC. It covers five areas of application for ML to the VIC.

Kernal routines. These are versatile and useful ML routines for input and output handling.

BASIC ROM routines. Less well-known than Kernal routines, these allow powerful results to be achieved with minimum effort. The section should be read in conjunction with Chapter 11's comprehensive breakdown of ROM and will let you construct your own programs from ROM subroutines.

Modifying BASIC. Wedges allow BASIC to perform entirely new commands; the technique for altering RAM is explained fully.

Vectors. Some BASIC commands are vectored through RAM; these too can be changed to add features to BASIC.

Interrupts. Both NMI and IRQ interrupts are possible with the VIC. This section gives examples of each, allowing (for example) music to be played while BASIC runs.

Other chapters discussing ML are Chapter 6 (keyboard, screen, etc.), Chapter 12 (graphics), and Chapters 13 through 17 (sound, tape, disks, peripherals).

Kernal Routines

The Kernal is a jump table giving access to routines which are supposed to be fairly constant between Commodore machines. In practice, this consistency can be realized only slightly, because so many hardware and software differences exist between machines.

Even so, the Kernal does serve a useful purpose. Its routines are arranged in ROM order in the following list. The Kernal appears less formidable if you note that more than half is concerned with opening and closing of files and input/output of characters. Table 8-1 lists input/output errors that may be returned by Kernal routines.

Table 8-1. Kernal Routine I/O Errors

JMP to	ERROR#	Description	Example
F77E	1	TOO MANY FILES	OPEN when ten files open already
F780	2	FILE OPEN	OPEN 1,3: OPEN 1,4
F783	3	FILE NOT OPEN	PRINT#5 without OPEN 5
F786	4	FILE NOT FOUND	LOAD "NONEXISTENT",8
F789	5	DEVICE NOT PRESENT	OPEN 11,11: PRINT#11
F78C	6	NOT INPUT FILE	OPEN 8,8,8,"SEQ,S,W": GET#8,X$
F78F	7	NOT OUTPUT FILE	OPEN 1,0: PRINT#1
F792	8	MISSING FILENAME	LOAD "",8
F795	9	ILLEGAL DEVICE NO.	LOAD "PROGRAM",3

Kernal routines with error-trapping return 0–9 in the accumulator.
Note: POKE 157,64 makes these I/O messages visible; try POKE 157,64: PRINT#55,X$ which prints I/O ERROR #3

Kernal Address	Routine Location	Name	Description
FF8A	**FD52**	**RESTOR**	**Set Default Vectors** Sets 16 vectors ($0314 to $0333) from a ROM table; used on power-up and RUN/STOP–RESTORE. Alters A, X, Y, and SR. No error returns.
FF8D	**FD57**	**VECTOR**	**Save/Set User Vectors** C flag set: Moves table from $0314–$0334 to X low, Y high address, saving current vectors. C clear: Moves table from X low, Y high back to $0314–$0334. Alters A, Y, and SR. No error returns.
FF90	**FE66**	**SETMSG**	**Control Screen Messages** Puts A into $9D to control messages. A has bit 7 on for direct mode, off for program mode. Bit 6 (not used by VIC) causes I/O errors to appear, as Table 8-1 shows. Alters A and SP. No error returns.
FF93	**EEC0**	**SECOND**	**Send Secondary Address After LISTEN** Can be used to send a secondary address to the serial bus after LISTEN. A holds the address, which is used unchanged, and therefore needs to be ORAd with #$60. After this subroutine, ATN is brought low so data output from the VIC-20 can begin (see Chapter 17). Alters A and SR, and probably X and Y. Errors returned in ST byte at $90.
FF96	**EECE**	**TKSA**	**Send Secondary Address After TALK** Can be used to send a secondary address after TALK on the serial bus; A needs to be ORAd with #$60. The routine checks for a return clock pulse. Alters A, and probably X and Y, and sets C flag. Errors returned in ST byte at $90.

FF99	**FE73**	**MEMTOP**	**BASIC RAM Top** C flag set: Loads X low, Y high from ($0283). C clear: Stores X low, Y high into ($0283). Note that ($0283) is not the normal top of memory, which is ($37), but that it holds the top of memory as detected by the VIC when power is applied. Alters X, Y, and SR. No error returns.
FF9C	**FE82**	**MEMBOT**	**BASIC RAM Bottom** Identical to MEMTOP, except that ($0281) is the relevant address.
FF9F	**EB1E**	**SCNKEY**	**Read Keyboard** Reads the keyboard and puts key, if any, into the keyboard buffer, where GETIN can recover it. The normal keyboard locations are used too, so $028D holds the SHIFT key indicator (see Chapter 6). Normally executed at each interrupt, this subroutine is useful for interrupt-driven routines where IRQ is moved or when the interrupt is disabled. Alters A, X, Y, and SR. C set on return means the buffer was full and the character wasn't accepted.
FFA2	**FE6F**	**SETTMO**	**Set Timeout** Not used by the VIC-20. Stores A into $0285, but that location is also never used. Intention is to set a timeout value, after which a serial device is assumed not present.
FFA5	**EF19**	**ACPTR**	**Input a Character from Serial Bus** Gets a byte from device number 4 or higher, typically disk. A file must be opened or the device made to talk. This routine is virtually identical to CHRIN (FFE4); the reason it has a Kernal address at all is because it allows GET from a device without a file necessarily being open. The character returns in A. Errors are returned in the status byte $90. Alters A, X, and SR.
FFA8	**EEE4**	**CIOUT**	**Output a Character to Serial Bus** Exactly analogous to ACPTR, this routine transmits the contents of A to device number 4 or higher, provided a file is open and ready or the device is a listener. CHROUT, FFD2, calls this routine. Errors return in the status byte $90. Alters A and SR.
FFAB	**EEF6**	**UNTALK**	**Untalk Serial Devices** Untalks devices on the serial bus, sending IEEE standard UNTALK command. Alters A, X, SR, and probably Y. Errors return in status byte $90.
FFAE	**EF04**	**UNLSN**	**Unlisten Serial Devices** Exactly analogous to UNTALK, this command unlistens devices numbered 4 or higher. Alters A, X, SR, and probably Y. Errors return in status byte $90.
FFB1	**EE17**	**LISTEN**	**Make Device a Listener** Converts a device on the serial bus to a listener. A holds the device number (4–30). ATN is held low to send the command byte, which is the device number ORAd with #$20. Alters A, X, SR, and probably Y. Errors return in status byte $90.

FFB4	**EE14**	**TALK**	**Make Device a Talker** Exactly analogous to LISTEN, this converts a device into a talker. The device number in A is ORAd with #$40. Alters A, X, R, and probably Y. Errors return in status byte $90.
FFB7	**FE57**	**READST**	**Read a Status Byte** Reads the status byte into A. Serial bus devices have $90 and RS-232 devices have $0297 for their respective status bytes. Note that the VIC has a bug. $0297 is converted to zero, and so is A. It's therefore necessary to use LDA $0297 with RS-232 devices to check ST. LDA $90 applies equally with serial bus devices but may not transfer between Commodore machines.
FFBA	**FE50**	**SETLFS**	**Set File Number Device, Secondary Address** This and the following routine are preliminaries to opening a file. They are, in effect, used by OPEN 4,4 and all other OPEN statements. There are three routines because the 6502 has only A, X, and Y registers. SETLFS puts the contents of A into file number storage in RAM, X into device number, and Y into secondary address. To mimic OPEN 1,4 in ML, load A, X, and Y with 1, 4, and 0 respectively, then JSR $FFBA. No error returns.
FFBD	**FE49**	**SETNAM**	**Set Filename** A is the length of the filename; X low, Y high points to the start of the name. If A is zero (not acceptable for disks, but OK for tape), X and Y become irrelevant. No error returns.
FFC0	**(031A) Usually F40A**	**OPEN**	**Open One File** Opens a file, assuming that the filename and other parameters have been set. Entry values of A, X, and Y are thus irrelevant. On exit, carry set indicates an error; the error number 1, 2, 4, 5, or 8 (see Table 8-1) returns in A. Alters A, X, Y, and SR.
FFC3	**(031C) Usually F34A**	**CLOSE**	**Close One File** A holds the file number on entry to this routine, which closes that file only, deleting its parameters from the file tables and decrementing the number-of-files-open location. On exit, C is clear. No errors are reported. Alters A, X, Y, and SR.
FFC6	**(031E) Usually F2C7**	**CHKIN**	**Prepare Open File for Input** Opens an input channel in preparation to read characters. This is analogous, with (for example) GET#4,X$, to converting GET to file 4 in place of the usual keyboard. The method here is to load X with the file number, call CHKIN with JSR FFC6, then typically use JSR FFE4 to get characters. After the characters are read, CLRCHN returns files and devices to normal, untalking them. On return from CHKIN, C set indicates an error; A holds the error number (3, 5, or 6). Alters A, X, Y and SR.
FFC9	**(0320) Usually F309**	**CHKOUT**	**Prepare Open File for Output** Exactly analogous to CHKIN, this prepares output to be directed to a file specified by CHKOUT, in the same way

			PRINT# commands operate. Load X with the file number, call CHKOUT with JSR FFC9, output characters with CHROUT, and close files with JSR CLALL, is a typical sequence. C set on return from CHKOUT indicates an error; A holds the error number (3, 5, or 7). Alters A, X, Y, and SR.
FFCC	**(0322) Usually F3F3**	**CLRCHN**	**Set I/O Devices to Normal** Sets output device to screen and input device to keyboard, and unlistens or untalks active devices. Leaves open files open, so CHKIN and CHKOUT still operate when wanted without further OPENs being needed. Compare CLALL, which is virtually identical but also closes all files. JSR FFCC is all that's needed. Alters A, X, and SR. No error returns. Note that $9A holds current output device number; $99 holds input device number.
FFCF	**(0324) Usually F20E**	**CHRIN**	**Input a Character** Gets a single byte from the current input device (stored in $99). This routine is identical to GETIN, FFE4, except for two factors. For keyboard characters, CHRIN is designed for use with INPUT statements and gets characters from the screen even when the keyboard is the nominal input device. Second, CHRIN with RS-232 loops until a non-null character is found. In all other cases (tape, disk), CHRIN and GETIN are identical. See the examples for use of CHRIN. JSR CHRIN returns the byte in A. Alters A, X, Y, and SR. Errors returned in ST byte $90.
FFD2	**(0326) Usually F27A**	**CHROUT**	**Output a Character** Outputs a single character to the current output device(s). A character may be sent to tape, RS-232, screen, or the serial bus, where any listener will receive the character. Generally there is only one listener. To use CHROUT, load A with the character, then JSR FFD2 to output it. A retains its value on entry; X and Y are unaltered. Errors return in status byte $90.
FFD5	**F542**	**LOAD**	**Load to RAM** Kernal LOAD is used by BASIC LOAD to load from tape or disk into RAM. The result is not rechained; this is a feature of BASIC LOAD. Kernal LOAD loads RAM from the device without any changes. Keyboard, RS-232, and screen return ILLEGAL DEVICE. Since commands like LOAD "*filename*",1,1 use a device number and name, SETLFS and SETNAM or the equivalent POKEs have to be called before LOAD. Before entering LOAD, A holds 0 for LOAD, 1 (or some nonzero value) for VERIFY. LOAD and BASIC's VERIFY use almost identical routines, except that VERIFY compares bytes rather than storing them in memory. X low, Y high points to the address ($C3) into which LOAD will start, provided that the secondary address is 0. If it isn't, ($C3) will be overwritten by the start address stored with the file to be loaded.

	(0330) Usually F549		LOAD has a vector after X and Y are stored. The routine branches to F563 (disk LOAD), and F5D1 (tape LOAD).
			On exit, C set denotes an error. A holds the error number (4, 5, 8, or 9). A, X, 4, and SR are all altered by LOAD. X low, Y high points to the end address, one byte after the final loaded byte, following LOAD.
FFD8	**F675**	**SAVE**	**Save to Device** Kernal SAVE is similar to LOAD. It dumps memory unchanged to tape or disk, has the same illegal devices, and requires SETLFS and SETNAM or their equivalents to be called first. There is no equivalent to a LOAD/VERIFY flag. But SAVE has to specify two addresses, the start and end addresses; it uses A, X, and Y for this. X low, Y high defines the end address (one byte past the relevant data's end). A is used rather clumsily as a pointer. If A holds #$2A, for instance, the contents of $2A (low) and $2B (high) define the start address.
	(0332) Usually F685		SAVE has a vector after the addresses that are stored in ($C1) and ($AE). After this the routine branches to F692 (disk) and F6F8 (tape) SAVEs.
			On exit, C set denotes an error. In this case A holds 5, 8, or 9. However, with disks there may be a disk error (FILE EXISTS) which has to be read from disk. Alters A, X, Y, and SR.
FFDB	**F767**	**SETTIM**	**Set Jiffy Clock** Stores Y (highest), X (high), A (low) into three RAM locations which store the jiffy clock. If TI$ is greater than 240000, the next interrupt resets to a normal time range. This is a rather trivial routine. The main problem with the TI clock is printing it in ML.
FFDE	**F760**	**RDTIM**	**Read Clock** The converse of the previous routine, RDTIM loads Y (highest), X (high), and A (low) from the TI clock's bytes. Again, the result usually needs some conversion to be useful.
FFE1	**(0328) Usually F770**	**STOP**	**Test STOP Key** Easy way to check if STOP is pressed, so STOP can be used to break into ML programs as an exit mechanism. JSR FFE1 then BEQ will branch if the STOP key is pressed. Note that seven other keys—left SHIFT (not right SHIFT), X, V, N, the comma, /, and cursor up/down—return unique values in A. A returns with #$FD if left SHIFT. A returns with #$FF if none of these keys is pressed.
			If STOP is pressed, CLRCHN is called. If you don't want this, use LDA $91, then CMP #$FE, which gives the same results as STOP.
			Alters A and SR and, if CLRCHN is called, Y. No errors returned.

FFE4	(032A) Usually F20E	GETIN	**Get One Character** Almost identical to CHRIN, except that keyboard input is taken directly from the keyboard buffer, like BASIC GET. Character is returned in A. Zero byte means no character found in keyboard; RETURN means no more disk characters; and space means no more tape characters. The other alternatives apply only when the input device number in $99 is changed from 0. Alters A, X, Y, and SR. No errors returned if keyboard GET; otherwise, errors returned in status byte $90.
FFE7	(032C) Usually F3EF	CLALL	**Abort All I/O** Sets number of open files to zero and unlistens and untalks all devices. Does not close files. Compare to CLRCHN, which is almost identical. Alters A, X, and SR. No error returns. Note: Because files aren't closed, this command may give problems with write files. It is always best to close them.
FFEA	F734	UDTIM	**Update Timer, Read STOP Key** Increments TI clock; if the result is 24 hours, returns to zero. (To keep correct time, increments must be made regularly by interrupt.) Location $91 is updated to hold the STOP key register, so the Kernal STOP routine can be used after this. Alters A, X, and SR. No error returns.
FFED	E505	SCREEN	**Return 22 and 23** SCREEN is a rather pointless routine, returning 22 in X and 23 in Y regardless of the true screen dimensions (which VIC can change). Alters X, Y, and SR. No errors.
FFF0	E50A	PLOT	**Cursor Position** C flag set: Reads $D6 into X and $D3 into Y, cursor positions down (0–22) and across (0–21) respectively. C flag clear: Acts the other way, putting X down, Y across. See examples of its use in practice. PLOT adjusts the screen links. Alters A, X, and SR. No errors.
FFF3	E500	BASE	**Return $9110** Loads X low, Y high with $9110, the start of the VIA chips. Not much use with VIC-20; could be useful if machine architecture were more variable.
(FFFA)	FEA9	NMI	**Non-Maskable Interrupt**
(FFFC)	FD22	RESET	**Reset**
(FFFE)	FF72	IRQ	**Interrupt Request and Break**

Using the Kernal

Using CHROUT to print to screen. CHROUT (FFD2) prints to screen as PRINT does, using the same VIC characters. This makes it an easy command to use, and produces a notable increase over BASIC's speed. Typically, a table of characters beginning with 147 ($93), clear screen, and ending with 0 is set up, including color and cursor characters; a Kernal loop prints these far faster than BASIC. The zero byte is used to indicate the end. Chapter 12 includes several graphics routines using CHROUT.

Try the following:

```
       033C  LDX  #$00
LOOP   033E  LDA  TABLE,X   ;TABLE COULD START 034A
       0341  BEQ  EXIT
       0343  JSR  FFD2
       0346  INX
       0347  BNE  LOOP
EXIT   0349  RTS or BRK
```

Using PLOT to position cursor. PLOT tends to need further routines to make it work properly; this example is typical. The non-Kernal routine is part of the power-on sequence, where the cursor is positioned at the screen start.

```
CLC          ; SET CURSOR. EXAMPLE VALUES:
LDX   #$09   ; 10TH LINE DOWN
LDY   #$03   ; 4TH COLUMN ACROSS
JSR   FFF0
JSR   E587   ; NON-KERNAL ROUTINE
```

Using GETIN to fetch keyboard characters. The following listing shows how this is done. With the unexpanded VIC, keypresses are echoed by a POKE to the screen top. In practice, more constructive uses are likely.

Note the loop branching back to JSR FFE4; this is exactly similar to the GET loop waiting for a character. Note also the test for * pressed; it avoids an infinite loop.

```
LOOP JSR   FFE4
     BEQ   LOOP  ;AWAIT KEY
     CMP   #$2A  ;A HOLDS BYTE. COMPARE WITH *
     BEQ   EXIT  ;EXIT ON *
     STA   $1E00
     BNE   LOOP  ;BRANCHES ALWAYS
EXIT RTS or BRK
```

FFE4 alters X and Y registers, unlike FFD2. Using FFD2 in a loop with X is therefore acceptable. But this will not work with FFE4; some storage location must be used.

Using CHRIN to fetch characters. The routine below shows how a loop inputs successive characters. If you precede this short program by the cursor position routine, you can simulate INPUT. The cursor will flash at the selected position onscreen. The demo prints the characters at the top of the screen (on the unexpanded VIC-20) to show how CHRIN works. Note the use of a temporary store for the current offset; X or Y is altered by CHRIN. As with BASIC INPUT, if you wish to validate a string being typed, GETIN is best, but CHRIN is easier to use.

```
;POSITION CURSOR
     LDA  #$00
     STA  $FE
LOOP JSR  FFCF
     CMP  #$0D  ;RETURN IS LAST CHARACTER
     BEQ  EXIT
     LDX  $FE
     INC  $FE
```

```
        STA  $0400,X ;POKE CHARACTER
        LDA  #$00
        STA  $9600,X ;AND COLOR RAM
        BEQ  LOOP
EXIT    RTS or BRK
```

Using LOAD and SAVE. Examples are in Chapter 6 (Block LOAD and SAVE) and in the disk and tape chapters. If the precise mechanism of these commands interests you, disassemble the routines, following the branches to tape or disk. Tape LOAD, at F5D1, prints SEARCHING, loads a header, computes the start and end addresses, prints LOADING, and continues with the data LOAD. Disk LOAD reads the first two bytes for its LOAD address.

Using OPEN and CLOSE. Chapter 15 contains disk examples.

Using READST. JSR FFB7 loads A with the status byte, either RS-232 or otherwise, depending on which device is used. VIC's RS-232 clears the byte. Because of this bug, $0297 has to be loaded without READST. This is a simple routine, saving you the effort of remembering ST's RAM address.

Using SCNKEY. Chapter 6's PAUSE is an example of how this can be used. The IRQ vector is redirected by altering (0314) to point somewhere other than EABF, its usual destination. The new routine sets the interrupt disable flag, so no further interrupts are allowed, and repeatedly reads the keyboard until some predetermined keypress occurs. At that time, CLI then JMP EABF carries on as though nothing had happened.

Using STOP. JSR FFE1 then BEQ EXIT is an easy way to stop ML from the keyboard. Without it, the STOP key is generally inactive. STOP is called after each BASIC statement is executed in a normal RUN, which is why STOP works with BASIC.

Using SETTIM and RDTIM. Both these commands are very simple. What's usually more important is converting the result into a readable form. This ML routine (non-Kernal) converts the clock's contents into a form exactly like TI$ (a string of exactly six numerals, with leading zeros where needed), so that a quarter after seven is 071500. The string is left in locations $FF–$0104, as the demo shows by POKEing it into the screen top. The six bytes can of course be edited and printed (for example) as 07:15:00.

```
       JSR   CF84
       STY   $5E
       DEY
       STY   $71
       LDY   #6
       STY   $5D
       LDY   #$24
       JSR   $DE68  ; NOW TI$ IS SET UP IN 00FF-0104
       LDX   #5     ; POKE 6 BYTES INTO SCREEN
LOOP   LDA   $00FF,X; NOT $FF,X
       STA   $1E00,X; INCLUDE COLOR IF YOU WANT
       DEX
       BPL   LOOP
       RTS or BRK
```

INPUT/OUTPUT. There's a relatively complex example at the end of Chapter 15 which includes eight Kernal calls.

Using BASIC's ROM Routines

BASIC has an enormous number of built-in routines, many of them having a recognizably BASIC feel about them. This section will show you how RUN can be performed from ML and will give you an easy way to input data from the screen. You'll see how numbers and strings can be input by ML. Finally, you'll look at calculation in ML, which is not as difficult as it might seem. Examples include the USR function, a hex-to-decimal converter, and a random number generator.

Executing RUN from ML

When there is BASIC in memory, JMP $C871 (or the SYS equivalent, SYS 51313) will run the program, provided it has a line 0. Any line of BASIC can be run from ML with this equivalent of RUN 100:

```
JSR   C660   ;CLR
LDA   #$LO   ;LOW BYTE OF LINE NUMBER
STA   $14
LDA   #$HI   ;HIGH BYTE OF LINE NUMBER
STA   $15
JSR   $C8A3  ;FIND LINE
JMP   $C7AE  ;GOTO LINE
```

This can be useful when ML calls BASIC; see UNLIST for an example in Chapter 6. Also, it's sometimes easier to include some BASIC along with ML, particularly with tricky programming involving arrays or file handling, which can be more trouble to convert to ML than they're worth.

Receiving Lines from the Keyboard

JSR $C560 prints a flashing cursor, then inputs the screen line into the 88-byte buffer starting at $200. This is easier to use than the Kernal CHRIN routine. The end of line is marked by a zero byte (replacing the carriage return character actually entered). Once input, the line can be processed in any way you want; normally, VIC tokenizes the buffer and treats it as BASIC. To get the feel of this, load and output characters from $200 onwards with CHROUT.

Processing BASIC Variables

INSTRING$ (Chapter 6) is ML with notes, showing how strings can be manipulated, using their three-byte parameters of length and pointer to start. VARPTR (also Chapter 6) uses JSR D08B to input a variable name and search for it in BASIC RAM. The address returns in Y/A.

Printing strings and numerals. A cluster of routines around $CB1E outputs strings without the need to repeatedly call FFD2 to print individual characters. For example, JSR DDDD then JSR CB1E prints the contents of FAC1. JSR DDDD converts the accumulator to an ASCII string, setting pointers ready for JSR CB1E to print.

CB1E is generally useful and will print any ASCII string up to a zero byte (or double quote mark), provided A (low) and Y (high) were set correctly on entry.

Inputting parameters for SYS calls from BASIC. PRINT@ and COMPUTED GOTO in Chapter 6 are examples which take in numbers, in the first case in the range 0–255, and in the second in the two-byte range 0–65535. The entire range isn't used in either example, of course. JSR D79B and JSR CD8A fetch the numbers. Other examples (for instance, Chapter 12's double-density graphics) take in data in the same way. There's generally a choice of registers and memory locations for use in transferring data between ROM routines. D79B returns the value in both $65 and X; CD8A evaluates numeric expressions (for instance, VAL(X$)+6*X) and leaves the result in FAC1, so there's less choice with this. COMPUTED GOTO shows one continuation with FAC1, namely conversion to integer format using only two bytes.

Calculations

This section explains, in some detail, how to carry out calculations in ML. With the help of Chapter 11, you'll see that useful results are relatively easy to achieve, so you should not be held back by problems needing arithmetic.

Floating Point Accumulator 1 (FAC1 for short) is a major location for number work. Occupying six bytes from $61 to $66, the format is slightly different from the five-byte variable storage of BASIC. Conversion from FAC1 to the memory format (MFLPT for short) rounds off the extra bit.

FAC storage can be summarized as EMMMMS, having an exponent byte, four entire bytes of data (mantissas), and a sign. If E is set to zero, the number is treated as zero regardless of M's contents.

Some math routines (like negation) operate only on FAC1. However, many use FAC2, including all the binary operations. For example, when adding, FAC1 and FAC2 are each loaded with a value; when the addition subroutine is called, the numbers are totaled and the result left in FAC1.

FAC1 can be stored in RAM either by copying the six bytes for later use or by using one of the routines around DBC7. You'll see an example in the ML hex-to-decimal converter later on.

Storing FAC1 in MFLPT format is of course part of BASIC, and many of Chapter 11's routines are relevant to BASIC. As an example, DD7E adds the contents of A to FAC1, and DAE2 multiplies FAC1 by 10. Between them, these routines allow ordinary decimal numbers to be input and stored in FAC1 as each digit is entered.

ROM routine D391 is an easy way to put integers from -32768 to 32767 into FAC1 as floating-point numbers. The following routine loads 1 into FAC1.

```
LDY  #1
LDA  #0
JSR  D391
```

The USR Function

USR is helpful with ML calculation programming. It is less often used with BASIC, because function definitions are much easier to write than USR. However, USR is a function; it is always followed by a value in parentheses, like PEEK.

To understand what it does, consider PRINT USR(6). When BASIC finds this, the value in parentheses is computed (to allow expressions like PEEK(J)) and the value is put into FAC1. Then BASIC executes JMP $0000. If it finds 0000 JMP 033C, for example, it continues to 033C with whatever program is there; eventually, when it finds RTS, the value then in FAC1 is the value printed by PRINT USR(6).

Thus, POKE 0,96 puts RTS at 0000, so USR returns without any alteration to FAC1. PRINT USR(6) is 6.

Program 8-1 is a more elaborate example. Load and run the program; then enter any number (say 1234) and five bytes will be output in MFLPT format.

Program 8-1. USR Demonstration

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 FOR J=828 TO 841: READ X: POKE J,X: NEXT:rem 15
20 POKE 0,76: POKE 1,60: POKE 2,3: REM USR VECTOR
   {SPACE}NOW $033C                            :rem 92
30 INPUT X                                     :rem 75
40 X=USR(X)                                   :rem 156
50 FOR J=842 TO 846: PRINT PEEK(J);: NEXT :rem 241
100 DATA 32,199,219,162,4,181,92               :rem 41
110 DATA 157,74,3,202,16,248,96               :rem 247
```

The first byte controls the magnitude of the number. The others determine its value, except for the high bit of the first data byte, which handles the sign. This is handy if you wish to store floating-point numbers in memory. It works by directing USR to the following:

```
033C  JSR   DBC7    ;FAC1 INTO MFLPT FORMAT AT $5C
033F  LDX   #$04    ;MOVE TO MORE PERMANENT RAM AREA
0341  LDA   5C,X
0343  STA   034A,X  ;WHERE PEEKS CAN RECOVER
0346  DEX
0347  BPL   0341
0349  RTS           ;BACK TO BASIC AFTER USR
```

Line 20's POKEs direct USR to 033C. Line 40 executes a USR command. First, whatever number was input is converted to FAC1 format. Then BASIC jumps to 0000, where it finds JMP 033C. Here, FAC1 is rearranged in RAM, and its five bytes are moved from their temporary storage (which would soon be overwritten) into the tape buffer. After RTS, BASIC resumes and MFLPT can be PEEKed.

For example, suppose you want to evaluate −10*X*X. Enter the following at 033C:

```
033C  JSR   DC0C  ;COPY FAC1 INTO FAC2
033F  JSR   DA30  ;MULTIPLY FAC1 BY FAC2. RESULT IN FAC1
0342  JSR   DFB4  ;NEGATE FAC1
0345  JSR   DAE2  ;MULTIPLY FAC1 BY 10. RESULT IN FAC1
0348  RTS
```

Return to BASIC. Then POKE 0,76:POKE1,60:POKE2,3 and PRINT USR(8). You'll get −640, and so on. If you have no ML facilities, POKE the following numbers from 828 to 839: 32, 12, 220, 32, 48, 218, 32, 180, 223, 76, 226, and 218.

Routines can be strung together like this in many ways, though it's helpful to know ML well enough to appreciate potential problems. For instance, add JSR DEFD to calculate EXP of FAC1. Alternately, use temporary storage areas. For instance, the following routine puts FAC1 into MFLPT form beginning at $57, then multiplies FAC1 by the MFLTP number it finds starting at $57. In effect, it is simply another way of multiplying a number by itself.

```
JSR  DBCA
LDA  #$57
LDY  #$00
JSR  DA28
```

USR is not a very important function, but as these examples show, it can be useful in testing ML calculation routines.

Hex-to-Decimal Conversion: A More Complex Example of ML Math

Program 8-2 is a longer program that illustrates several points. INIT sets FAC1 to zero and stores 16 in MFLPT form in spare RAM (in fact, in the random number storage area). GET not only fetches an individual character, but also flashes the cursor and tests for the STOP key. PROC is the processing part; each digit is converted from ASCII (#$30 to 0, #$41 to 10, and so on), added to FAC1, and, if a further digit is wanted, multiplied by 16. PRINT outputs the result.

Program 8-2. Hex-to-Decimal Conversion

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 DATA 169,4,133,254,169,0,133,97,162      :rem 81
11 DATA 4,149,139,202,208,251,169,133,133 :rem 223
12 DATA 139,169,0,133,204,133,207,32,225   :rem 165
13 DATA 255,208,5,169,1,133,204,96,32,228 :rem 227
14 DATA 255,240,241,72,32,210,255,104,201 :rem 205
15 DATA 65,144,2,233,8,233,47,32,126,221  :rem 168
16 DATA 198,254,240,11,169,139,160,0,32   :rem 127
17 DATA 40,218,240,212,208,210,32,221,221 :rem 200
18 DATA 32,30,203,169,13,32,210,255,240   :rem 110
19 DATA 172,208,170                        :rem 168
100 FOR J=828 TO 913: READ X: POKE J,X: NEXT
                                            :rem 63
```

SYS828 accepts four-digit hex numbers and continues until the STOP key is pressed. The routine is relocatable. For binary-to-decimal conversion, POKE 829,8: POKE 844,130 after running.

The complete source code is given below.

```
INIT    033C  LDA  #$04
        033E  STA  $FE      ;COUNT 4 DIGITS
        0340  LDA  #$00
        0342  STA  $61      ;FAC1 NOW ZERO
        0344  LDX  #$04
        0346  STA  $8B,X    ;LOOP PUTS 16 IN
        0348  DEX          ;MFLPT FORM INTO
        0349  BNE  $0346   ;8B-8F (RND AREA)
        034B  LDA  #$85    ;FOR REPEATED USE
        034D  STA  $8B
        034F  LDA  #$00
        0351  STA  $CC
        0353  STA  $CF     ;CONTROL CURSOR
GET     0355  JSR  $FFE1   ;TEST STOP KEY
        0358  BNE  $035F
        035A  LDA  #$01    ;IF STOP PRESSED,
        035C  STA  $CC     ;FLASH CURSOR
EXIT    035E  RTS          ;RETURN
        035F  JSR  $FFE4   ;GET CHARACTER FROM KEYBOARD
        0362  BEQ  $0355   ;WAIT FOR NON-NULL CHR
PROC    0364  PHA          ;SAVE CHARACTER
        0365  JSR  $FFD2   ;ECHO TO SCREEN
        0368  PLA          ;RECOVER
        0369  CMP  #$41    ;COMPARE WITH A
        036B  BCC  $036F   ;BRANCH IF LESS THAN A
        036D  SBC  #$08
        036F  SBC  #$2F    ;CONVERT ASC 0-F TO 0-15
        0371  JSR  $DD7E   ;ADD A TO FAC1
        0374  DEC  $FE     ;REDUCE COUNTER
        0376  BEQ  $0383   ;EXIT AFTER 4 DIGITS
        0378  LDA  #$8B
        037A  LDY  #$00    ;SET POINTERS TO $8B
        037C  JSR  $DA28   ;MULTIPLY FAC1 BY MFLPT AT $8B (IE BY 16)
        037F  BEQ  $0355   ;BRANCH BACK (RELOCATABLE) FOR
        0381  BNE  $0355   ;NEXT DIGITS
PRINT   0383  JSR  $DDDD   ;CONVERT FAC1 INTO STRING AT $100
        0386  JSR  $CB1E   ;PRINT STRING
        0389  LDA  #$0D    ;PRINT RETURN TO GO TO
        038B  JSR  $FFD2   ;NEXT LINE
        038E  BEQ  $033C   ;BRANCH BACK (RELOCATABLE) FOR
        0390  BNE  $033C   ;NEXT HEX NUMBER
```

Random Numbers

Random numbers are used in simulations and in games. From ML, the easiest method is to call ROM routines, which have the advantage of being repeatable if you want them to be. JSR E0D0 is equivalent to RND(−X) and seeds the random number storage area with a value dependent on FAC1. The reason RND of negative integers is always very small is that the FLPT bytes are simply switched around.

E0D0 can be used to seed a constant value. However, with ML it's quicker to store your own seed value directly in 8B–8F.

JSR E0BB uses a formula to calculate a new random number from the previous one, leaving the result in both FAC1 and 8B–8F. The sequence is completely predictable.

JSR E09B uses VIA timers to generate a true random number, except in the sense that very short ML loops may start to show regularities.

Typically, during testing, a seed is chosen. Then E0BB is used to give a repeatable sequence (this eases debugging). The seed is replaced by E098 for use.

One- or Two-Byte Random Numbers

Often more useful in ML, you could use the following routine:

```
JSR  $E0BB ;NEW RND NUMBER FROM OLD
LDA  $8C
EOR  $8D    ;COMBINE DATA BYTES
EOR  $8E    ;INTO COMPOSITE BYTE
EOR  $8F
```

It uses all four bytes, excluding the exponent, presumably increasing the result's randomness. Suppose you want something to happen 10 times in 256. All you need is CMP #$0A, then BCC to branch when the accumulator holds 0–9.

If you need a random number in ML within a fixed range, say 0–20, the easiest method is to use repeated subtraction (rather than to get a decimal, multiply by 20, take an integer, and add 1):

```
RANGE   CMP  #$15    ;COMPARE WITH DECIMAL 21
        BCC  FOUND   ;NUMBER IN RANGE 0-20
        SBC  #$15    ;SUBTRACT DECIMAL 21
        JMP  RANGE   ;COMPARE AGAIN
FOUND   CONTINUE     ;A HOLDS 0-20 DECIMAL
```

Note that a random number from 48 to 57 is 48 plus a random number from 0 to 9.

If you need random numbers in quantity, it's faster to generate your own. All you need is one RAM location (or two for a 16-bit number). The following routine uses a single byte, LO:

```
LDA  LO
ASL
ASL
CLC
ADC  #$odd
ADC  LO
STA  LO
```

Any odd number can be selected (#$81 for example). The contents of LO now cycle through 256 different values in sequence. The method uses 5 times the previous value plus an odd number, ignoring overflow above 255; in other words, x becomes $5x+c$ (mod 256). Five is easy to program, but 9, 11, 21, or other numbers can also be used.

Each call to this routine loads A with the next number; this is not necessarily suitable as a random number, since the series repeats, but EOR with a timer (for example, EOR $9115) will scramble LO into an unpredictable form.

For a two-byte random number, use the following:

```
CLC
LDA  LO
ADC  HI
STA  HI
CLC
```

```
LDA  #$odd
ADC  LO
STA  LO
LDA  #$any
ADC  HI
STA  HI
```

In this case, x becomes 257*X+c (mod 65536) where c is odd. Any series generated from this repeats at 65536 cycles.

Sequences generated by this method always produce alternate odd and even values, and internal subseries are common, so the guarantee of a very long repeat interval doesn't insure success in any actual application. However, details like this are unlikely to trouble VIC users.

Series Calculations

All of the VIC's mathematical functions are evaluated by series summation. Briefly, the value to be converted is first put into a smaller range. Trig functions, for instance, repeat regularly, so their input values can be reduced (if large) by subtracting multiples of π . Then a series evaluation works out the function's value, and finally an allowance is made for the initial scaling-down process.

In the VIC-20, the ROM routine at E056 sums the series. The following short example shows how:

```
LDY  #$03
LDA  #$40
JSR  $E056
RTS
```

Location 0340 must contain 2, and locations 0341–0345, 0346–034A, and 034B–034F each must contain a number in MFLPT format. If we designate these N1, N2, and N3, calling the routine replaces FAC1's value with N3 + N2*X + N1*X*X. Working out the actual series parameters is beyond this book's scope.

Integer to Floating-Point Conversion and Multiply/Divide by 2, 4, 8, etc.

Although conversion of two-byte integers into floating-point form is often useful, the standard ROM routine at D391 converts A (high) and Y (low) into the range −32768–32767, the range of integer variables.

The following routine puts A (high) and Y (low) into FAC1 in the range 0–32767. There is no shorter way to do this. Note that the VIC-20 has vectors near the start of RAM which can be changed to allow for just such modifications.

```
LDX  #$00
STX  $0D
STA  $62    ;HIGH
STY  $63    ;LOW (NOTE REVERSE ORDER)
LDX  #$90   ;EXPONENT (#$91 DOUBLES, #$94 MULTIPLIES BY 16)
SEC
JSR  $DC49 ;CONVERT TO FAC1
```

Modifying BASIC

VIC BASIC has a large number of vectors in RAM; these are addresses which BASIC uses as it runs. If these vectors are altered, BASIC can be intercepted and new commands tested for and executed. Alternately, old commands can be modified slightly (or completely) as required.

The techniques are simple, but plenty of small problems await the programmer. In particular, when you alter BASIC, any errors in the added ML are likely to prevent BASIC from working normally. Thus it's important to save programs as they are written, or to use a reset switch for emergency recovery, in order to avoid tedious retyping. All the methods use ML, but this need not be too daunting, since the VIC can do most of the work.

Vectors and Wedge Methods

RAM contains blocks of vectors as indirect addresses. LIST, for example, has a command JMP ($0308) within it, so the contents of $0308 and $0309 determine what LIST does.

There are about 20 such vectors. In addition, the CHRGET routine at $0073 (which fetches BASIC characters as it runs) is accessible for programming; as you'll see, this allows access to BASIC as it runs, so new commands can be added.

Alterations to CHRGET or to vectors called from BASIC are semipermanent. Once in place, SYS 64802 or switch off and on will remove them, but otherwise even STOP-RESTORE leaves them untouched. This is intentional. On the other hand, STOP-RESTORE sets vectors used by the Kernal routines to the default values.

First, consider examples involving vector alterations and use of wedges. Generally, wedges are probably easier to program; there's only one subroutine to worry about, and commands can be added almost indefinitely. However, tokenization isn't possible, so short commands like @X or @Y are generally used.

BASIC vectors allow some effects to be achieved which aren't possible with wedges, for example, modified LIST. The *Super Expander* uses most of these vectors, which allow it to insert its own tokens, translate them, LIST them properly, and so on. Such large-scale modifications take prior planning and are not for inexperienced programmers. Kernal vectors are easier to deal with, in the sense that they give convenient access to commands but are not often used. There are not that many occasions when you would want to reprogram OPEN, LOAD, or SAVE. Kernal vectors, like the three interrupt vectors, are all set to normal by STOP-RESTORE.

The Wedge

To understand the wedge, first look at CHRGET, the RAM routine starting at $0073, which fetches every BASIC character while BASIC runs:

```
CHRGET  0073  INC  $7A      ;ADDS 1 TO CURRENT ADDRESS
        0075  BNE  $0079    ;WITH THIS STANDARD 2-BYTE
        0077  INC  $7B      ;INCREMENT
CHRGOT  0079  LDA  CURRENT
        007C  CMP  #$3A     ;COLON (OR GREATER) EXITS
        007E  BCS  $008A
        0080  CMP  #$20     ;SKIPS SPACE CHARACTERS
```

```
0082  BEQ  $0073
0084  SEC                ;ANYTHING FROM #$30 TO #$39
0085  SBC  #$30          ;CLEARS C FLAG
0087  SEC                ;ELSE C IS SET
0088  SBC  #$D0
008A  RTS
```

CHRGET is stored in ROM at $E387; SYS 58276 from BASIC moves it back to RAM. This may be useful if you've altered CHRGET, but note that it NEWs BASIC. A call to CHRGET returns with A holding the next BASIC character, C clear if an ASCII numeral was found, and the zero flag set if either a colon or null byte was found. JSR $0073 followed by BCC or BEQ is common in ROM, and BCC applies when a line number (made of ASCII numerals) is read from a GOTO or GOSUB statement.

ROM also uses JMP $0073. In this case, RTS uses the address it finds on the stack, and in fact BASIC keywords are executed in this way. The 6502 requires that the return address−1 be pushed on the stack.

CHRGET can be changed. Try POKE129,234:POKE128,234:POKE131,234: POKE130,234 (without spaces between POKEs). This deletes the test for space characters, replacing their ML by NOP commands, and BASIC runs exactly as normal except that spaces outside quotes cause ?SYNTAX ERROR. The first SEC becomes redundant, and SBC #$2F corrects for it. CHRGET shortened like this runs BASIC faster than normal, as expected; but only by an unexciting one-half percent.

Before seeing how to insert a wedge, note the difference between CHRGET and CHRGOT. CHRGET always increases the current address; it's normally called only once per character. CHRGOT rereads the current BASIC character and sets the relevant flags; therefore, whenever processing loses track of the current BASIC character in some way, CHRGOT is always available to recover it.

Wedge Demonstration

If you replace 0073 INC 7A by 0073 JMP 033C, or some other jump address, all ROM calls to BASIC characters can be intercepted before they are executed. This allows us to test for and use completely new commands in BASIC. A wedge, once inserted, is quite durable; STOP-RESTORE, for example, leaves it unaltered. That can be important. If your routine has an error, it may be impossible to POKE in the correct values or enter a SYS call to replace the wedge, since BASIC itself is behaving differently from usual.

Programmer's Aid (see end of Chapter 6) and the disk wedge (Chapter 15), as well as the TRACE version of Chapter 6, all use wedges. This example puts JMP at 0073; note that 0073 or subsequent addresses can be used, and are sometimes better, since they may allow another wedge to be used simultaneously. Some wedges test for JMP at 0073 and allow for them. They also allow zero page RAM (typically 007F–008A) to be used in programs.

Program 8-3 adds the single command ! to BASIC. When it executes, the screen characters are reversed. When naming new commands, it's easiest to use a character which doesn't appear in ordinary BASIC (like !, @, or &) as an identifier. If desired, it is easy to add further commands, such as !R or !P (with specific functions of their own) by getting the following BASIC character with JSR $0073 whenever ! is found. However, to keep the example shorter, it adds only a single command.

Program 8-3. BASIC Wedge Demonstration

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 DATA 169,76,133,115,169,73,133,116,169 :rem 239
11 DATA 3,133,117,96,230,122,208,2,230     :rem 57
12 DATA 123,32,121,0,201,33,240,3,76,121  :rem 139
13 DATA 0,165,123,201,2,240,247,152,72     :rem 57
14 DATA 138,72,162,0,189,0,30,73,128,157  :rem 176
15 DATA 0,30,189,0,31,73,128,157,0,31,202 :rem 203
16 DATA 208,237,104,170,104,168,76,115     :rem 80
17 DATA 0,169,230,133,115,169,122,133,116 :rem 217
18 DATA 169,208,133,117                    :rem 113
100 FOR J=828 TO 906: READ X: POKE J,X: NEXT
                                           :rem 65
```

Note that ! is accepted only in program mode. A line like 100! works perfectly, but ! on its own is an error. This is deliberate. It avoids commands being executed while a program is being written, when they may not be wanted, although you could obviously create a wedge (like the disk wedge) that only works in direct mode. There are several tests for direct mode; TSX then LDA $0102,X to recover the return address is one; another is location $9D, which usually holds #$80 in direct mode. The test in the example simply checks the current address used by CHRGOT; if it's around $0200, it must be a direct mode line.

The only peculiarity of BASIC syntax with wedges is the IF statement. IF X=1 THEN: PRINT "ONE" is correct as far as BASIC is concerned, but the colon can be omitted. With wedges, the colon can't be left out.

How the wedge works. Program 8-3 loads the following ML into the VIC:

```
SETUP  033C LDA #$4C   ;PUTS JMP $0349 INTO CHRGET
       033E STA $73
       0340 LDA #$49
       0342 STA $74
       0344 LDA #$03
       0346 STA $75
       0348 RTS
WEDGE  0349 INC $7A    ;MIMIC CHRGET
       034B BNE $034F
       034D INC $7B
       034F JSR $0079  ;A NOW HOLDS BASIC CHARACTER
       0352 CMP #$21   ;IS IT ! ?
       0354 BEQ $0359
       0356 JMP $0079  ;NO—JMP BACK TO CHRGOT. WEDGE UNUSED
       0359 LDA $7B    ;YES—CHECK FOR DIRECT MODE
       035B CMP #$02
       035D BEQ $0356  ;DIRECT MODE—DON'T USE WEDGE
       035F TYA        ;PROGRAM MODE—USE WEDGE
```

```
0360  PHA          ;SAVE X AND Y
0361  TXA
0362  PHA
0363  LDX  #$00    ;EXECUTE ! COMMAND TO
0365  LDA  $1E00,X ;REVERSE UNEXPANDED VIC CHARACTERS
0368  EOR  #$80
036A  STA  $1E00,X
036D  LDA  $1F00,X
0370  EOR  #$80
0372  STA  $1F00,X
0375  DEX
0376  BNE  $0365
0378  PLA          ;PROCESSING OVER. RECOVER X AND Y
0379  TAX
037A  PLA
037B  TAY
037C  JMP  $0073   ;AND CONTINUE WITH NEXT BASIC COMMAND
037F  LDA  #$E6    ;DEACTIVATE WEDGE
0381  STA  $73
0383  LDA  #$7A
0385  STA  $74
0387  LDA  #$D0
0389  STA  $75
038B  BRK
```

SYS 828 activates the wedge. SYS 895 turns it off. Note that the entire routine is relocatable, apart from the address in SETUP; long routines won't fit the tape buffer but can be put at the top of BASIC memory.

Note that 035D jumps to CHRGOT, not CHRGET. This means that ! in direct mode is treated as normal, generating ?SYNTAX ERROR if entered as a command, but included in BASIC otherwise. If 035D jumps to CHRGET there are no syntax errors, but the command becomes difficult to include in BASIC. Note as well that 037C jumps to 0073. Of course it immediately jumps back to 0349, but 0073 always relocates.

Using Vectors in RAM to Modify BASIC

The main blocks of vectors start at $300. (0300) through (030A) are vectors from BASIC. (0314), (0316), and (0318) are vectors from IRQ, BRK, and NMI. (031A) through (0332) are vectors from Kernal routines, except (032E) which is unused.

Earlier RAM has a sprinkling of vectors, including (028F) (used by SCNKEY, the keyboard-reading routine, which allows keys to be intercepted). USR's JMP instruction is at the start of RAM, and floating-to-fixed (and vice versa) number conversion is routed through (0003) and (0005).

There are six BASIC vectors, each called from the address just before its normal destination. For example, C437 JMP (0300) is set to C43A. They will be examined here in order.

(0300)

This is the error-message vector. X holds the number of the error; for instance, decimal 10 means NEXT WITHOUT FOR. Unless altered, this prints an error message, then READY. See Chapter 6 for ONERR...GOTO, a command included in some BASICs, which allows an error routine to be specified at some line number.

Program 8-4 lists a single BASIC line which has been found incorrect; the text is reversed at about the place the error is to be found. For example, 10 PRINT ((8) is listed with the third parenthesis reversed. There's no way to point up the exact position of the error, but an indication is often useful.

The modification will remain in effect as long as the POKEs stay in place, so a new program can be loaded and tested. Because it has been kept short, the error message is replaced by READY. If LIST is moved to RAM, the error can be printed too.

Program 8-4. Error Detection

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 DATA 165,57,133,20,165,58,133,21,32      :rem 68
11 DATA 19,198,165,122,24,229,95,133,254   :rem 188
12 DATA 56,76,189,198,72,196,254,208,9     :rem 112
13 DATA 169,18,32,210,255,169,255,133,254 :rem 233
14 DATA 104,76,26,199,                      :rem 60
100 FOR J=828 TO 868: READ X: POKE J,X: NEXT
                                            :rem 72
110 POKE 768,60: POKE 769,3: REM ALTER ERROR VECTO
    R                                       :rem 89
120 POKE 774,82: POKE 775,3: REM ALTER LIST VECTOR
                                            :rem 10
```

Here is how the program works.

```
ERR  033C LDA $39
     033E STA $14
     0340 LDA $3A
     0342 STA $15
     0344 JSR $C613  ;LISTS CURRENT LINE
     0347 LDA $7A    ;LOW BYTE OF CHRGET
     0349 CLC
     034A SBC $5F    ;LESS LOW BYTE OF LINE START
     034C STA $FE    ;IS DISPLACEMENT OF ERROR. SAVE IT
     034E SEC        ;INDICATES LINE FOUND
     034F JMP $C6BD  ;JUMP TO LIST ROUTINE
LIST 0352 PHA        ;SAVE CHARACTER TO BE LISTED
     0353 CPY $FE    ;COMPARE LIST POINTER WITH
     0355 BNE $0360  ;ERROR DISPLACEMENT.
     0357 LDA #$12   ;IF EQUAL, OUTPUT REVERSE CHR
     0359 JSR $FFD2  ;SO LIST REVERSES TEXT
     035C LDA #$FF   ;RESET DISPLACEMENT HIGH
     035E STA $FE
     0360 PLA        ;RECOVER CHARACTER
     0361 JMP $C71A  ;CONTINUE LIST
```

Two vectors must be changed: (0300) from C43A to 033C, and (0306) from C71A to 0352.

(0302)

This vector, sometimes called IMAIN, usually points to C483 and is called just after READY prints, before input from the keyboard. Try POKEing these values into 828–835: 169, 42, 32, 210, 255, 76, 131, and 196. Now POKE 770,60:POKE 771,3. The effect is to move the vector here:

```
LDA  #$2A  ;LOAD WITH *
JSR  FFD2  ;OUTPUT IT
JMP  C483  ;CONTINUE AS USUAL
```

Now, the cursor expecting input is preceded by *. In fact, you can tell when the routine is called by its presence.

IMAIN allows several possibilities, including automatic BASIC line numbering, output of some message or prompt, and automatic LOAD and RUN, as Chapter 14 shows.

(0304)

This vector, sometimes called ICRNCH, tokenizes BASIC. It is scanned while in the buffer from $200 and has keywords converted to tokens. This vector could be diverted, so new keywords can be recognized and converted into tokens. If this is done, (0308) and (0306) have to be altered too.

(0306)

(0306) is part of LIST. You have seen an example with (0300); here's another. POKE these values from 828 to 846: 72, 201, 58, 208, 10, 169, 13, 32, 210, 255, 169, 32, 32, 210, 255, 104, 76, 26, and 199. Now POKE 774,60:POKE 775,3. This simple routine compares the character to be listed with a colon; if it finds one, it starts a new line and prints a space. In effect, it makes LIST separate individual lines. (It doesn't test for colons within strings.) This sort of thing is useful with printers and could include a test for output device (= 4).

(0308)

(0308) is analogous to CHRGET but points only at tokens. It is used just before a statement is executed. If there's no token, LET is assumed. Bytes outside the range of valid tokens trigger a ?SYNTAX ERROR. You can intercept the routine and process your own tokens, or (as in Program 8-5) redefine a standard token.

Program 8-5. Redefining LET

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 DATA 32,115,0,201,136,240,6,32,121,0     :rem 85
11 DATA 76,231,199,32,155,215,142,15,144   :rem 176
12 DATA 76,234,199                         :rem 126
20 DATA 169,3,141,9,3,169,60,141,8,3,96    :rem 127
100 FOR J=828 TO 860: READ X: POKE J,X: NEXT
                                            :rem 64
```

```
110 SYS 850: REM CHANGES VECTOR IN ($0308) TO POIN
    T TO NEW ROUTINE                          :rem 207
```

Note that a SYS call is needed to alter the vector, because POKE is processed using this and gets confused if one byte of the vector changes.

After RUN, LET is redefined so LET 123 sets the border cyan and background yellow. This in effect POKEs $900F with 123, but it is much faster than POKE. FOR J=0 TO 255: NEXT cycles the colors at great speed. LET X OR 8 cuts out the reverse bit; the extra ML is given below:

```
HERE     JSR   0073      ;GET NEXT BASIC CHR
         CMP   #$88      ;LOOK FOR LET TOKEN
         BEQ   LETFND    ;BRANCH IF FOUND
         JSR   0079      ;CHRGOT SETS FLAGS
         JMP   C7E7      ;CONTINUE NORMALLY
LETFND   JSR   D79B      ;CALCULATE 1-BYTE BASIC PARAMETER
         STX   900F      ;PUT IT IN VIC REGISTER
         JMP   C7EA      ;CONTINUE, AFTER EXECUTION POINT
```

LET (and GO) and many mathematical functions lend themselves to this treatment, which may be helpful in dealing with some of the more tiresome commands needing POKEs and PEEKs. Chapter 13 has examples involving sound.

(030A)

(030A), normally CE86, points to the subexpression evaluator routine, which gets and evaluates single terms of expressions at runtime (for example, the values of X and 123 in the statement PRINT X+123). The reason for its inclusion in the vectors is to allow nonstandard terms, either string or numeric, to be defined. Thus, hex numbers beginning & can be introduced into BASIC; so can binary numbers beginning %. Use the following:

```
HERE   JSR  $0073   ;GET FIRST CHR OF TERM
       CMP  #$26    ;IS IT &?
       BEQ  YES     ;IF YES, BRANCH
       LDA  #$00    ;IF NO, SIMULATE
       STA  $0D     ;NORMAL BEHAVIOR
       JSR  $0079
       JMP  $CE8D
YES    JSR  $0073   ;GET 1ST CHR AFTER &
       ......       ;PROCESS. PUT IN FAC1
       JMP  $0073
```

Set (030A) (that is, 778 and 779) to point to HERE.

Program 8-6 adds hex numbers to BASIC, using the principles just explained. For example, POKE $900F,1 works correctly. PRINT $1234−$0055 subtracts one number from another, printing the result in decimal.

Program 8-6. Adding Hex Numbers to BASIC

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 DATA 169,71,141,10,3,169,3,141,11,3      :rem 55
11 DATA 96,169,0,133,13,32,115,0,201,36    :rem 106
```

```
12 DATA 240,6,32,121,0,76,141,206,162,2    :rem 100
13 DATA 32,115,0,201,64,144,2,105,8,10      :rem 45
14 DATA 10,10,10,133,254,32,115,0,201,64   :rem 137
15 DATA 144,2,105,8,41,15,5,254,72,202      :rem 61
16 DATA 208,224,104,168,104,133,98,132      :rem 78
17 DATA 99,162,144,56,32,73,220,76,115,0   :rem 180
100 FOR J=828 TO 905: READ X: POKE J,X: NEXT
                                            :rem 64
110 SYS 828 :REM ALTERS (030A). THIS MUST BE DONE
    {SPACE}FROM ML, NOT BASIC               :rem 233
```

To compute results in the range 0–65535, a modified fixed-to-floating point routine has to be used, as explained earlier.

IRQ, BRK, and NMI Vectors

IRQ interrupts are timed by a VIA (see Chapter 5 for notes on changing the frequency). If the interrupt isn't masked by SEI, a jump to $FF72 always occurs, and from there the status register distinguishes IRQ and BRK interrupts. The latter aren't of great interest; in fact, they're relevant only to ML monitors, where BRK typically reenters the monitor. *VICMON*, for example, sets this vector automatically when it's initialized, and future uses of BRK reenter *VICMON*.

IRQ interrupts are vectored through ($0314) to $EABF, where they process the keyboard and a few other things. It is fairly simple to add ML elsewhere, ending with JMP $EABF, then alter ($0314) to point to the new code. A complication is that these interrupts are continually taking place, so the code must all be set up before the vector is changed. Moreover, the vector can't safely be changed from BASIC in case another interrupt occurs while only one byte in ($0314) is changed. This is why the interrupts must be turned off as follows:

```
033C  SEI
033D  LDA  #$03
033F  STA  $0315
0342  LDA  #$49
0345  STA  $0314
0348  RTS
0349  ML ending JMP $EABF
```

SYS 828 changes the vector to $0349; it remains changed until STOP-RESTORE or reset.

NMI interrupts are unmaskable and can also be timed by a VIA. Thus they offer the possibility of absolutely regular processing, unlike IRQs which may be masked off at intervals. This is why they are used by the RS-232 software. They do have some drawbacks: Disks and tape are made inactive while they operate, and the code has to be a little longer. On the other hand, the vector is easy to move, and interrupts can be turned off with a simple POKE.

To use NMIV at ($0318), redirect the vector as follows:

```
PHA
TXA
PHA
```

```
TYA
PHA
ML   PROCESSING
JMP  $FEB2
```

Unlike IRQ, saving A, X, and Y is not built in ROM before the vector. The easiest way to generate interrupts is to POKE $911E (37150) with #$C0 (192), and to set the timer in $9114 and $9115 (37140, 37141). The ML processing must reset this timer, or interrupts will stop.

Kernal Vectors

All of these, except for ILOAD and ISAVE, are called immediately on entering the Kernal jump table. LOAD and SAVE first store address parameters.

As a fairly simple example, intercept CHROUT and cause it to output an extra asterisk. POKE these values into 828 to 837: 72, 169, 42, 32, 122, 242, 104, 76, 122, and 242. Now POKE 806,60: POKE 807,3 to alter the vector to $033C. The ML is given below:

```
PHA         ;SAVE OUTPUT CHARACTER
LDA  #$2A   ;LOAD ASTERISK
JSR  $F27A  ;OUTPUT ASTERISK, NOT WITH FFD2 OF COURSE
PLA         ;RECOVER CHARACTER
JMP  $F27A  ;OUTPUT IT, CONTINUE
```

Now, any use of CHROUT prints an asterisk. This includes all BASIC messages like READY, which appears as *R*E*A*D*Y*.

Using Interrupts

As you have seen, two types of interrupts are vectored through RAM, and either can be intercepted for your own background programs. Generally, IRQs are used. However, Program 8-7 is a demonstration with NMIs.

Program 8-7. Using Interrupts with NMIs

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
Ø DATA 169,86,141,24,3,169,3,141,25,3,169 :rem 226
1 DATA 1Ø2,141,2Ø,145,169,136,141,21,145  :rem 157
2 DATA 169,192,141,3Ø,145,96,72,138,72     :rem 88
3 DATA 152,72,169,1Ø2,141,2Ø,145,169,136  :rem 173
4 DATA 141,21,145,238,Ø,3Ø,76,178,254      :rem 25
1Ø FOR J=828 TO 874: READ X: POKE J,X: NEXT:rem 21
```

After RUN, SYS 828 diverts the NMI vector and the background program runs. It increments $1E00, the top left screen position (with the unexpanded VIC). Locations 860 (low) and 865 (high) control the interrupt rate. As with IRQ, if the number of interrupts is large, most of the computer's time is spent on them, and LIST works very slowly. For an example using IRQs, see Chapter 13's interrupt-driven music program. It illustrates most of the aspects of interrupt programming.

It is helpful to summarize the steps needed to create interrupt-driven programs. First, find a safe area of RAM to store the ML. Long routines are best stored at the top of BASIC memory; the music program does this, lowering BASIC's top of memory so pure BASIC can never touch the ML.

Second, write the initialization routine. Typically, it will be as follows:

```
SEI
LDA  #HI
STA  $0315
LDA  #LO
STA  $0314
RTS
```

A SYS call here alters the IRQ vector to point to ML; note that CLI isn't necessary.

Third, write the interrupt routine, keeping in mind a few important differences from normal programming:

- A, X, and Y can be used completely independent of BASIC. The BASIC values at the time the interrupt happened are saved automatically and restored on return from interrupt, so your ML is self-contained.
- The simplest termination of an IRQ routine is JMP EABF. This is the usual address in (0314), and exit to it means that BASIC behaves exactly as normal apart from the introduced ML. All exits must return properly, or BASIC will crash. It's not essential to exit via EABF. For instance, EAC2 exits omitting the time-updating routine with STOP-key test, and EB18 exits omitting all of BASIC's normal interrupt operations. But EABF is simplest.
- Keep in mind the effects of repeats. Time dependency is a little hard to get used to. A command like DEC $FE in normal programs decrements the contents of $FE just once, from (for example) 9 to 8. But in an interrupt-driven program, this command decrements whenever interrupts occur, typically 60 times per second. This is how the TI clock works (except that it increments). Clearly this is a valuable feature.

The music program relies on several counters. One, controlling tempo, counts down at each interrupt. When zero, it's replaced by its original value, and a new note is taken from a table and stored in a VIC sound register. If the constant in the counter is 10, the note will be changed every 1/6 second at most. Longer notes simply have the same note repeated in the note table, allowing durations of various lengths.

The VIC polls, or tests, various locations during the interrupt (for example, the keypress location $C5 (197)) and acts on the results. Chapter 16 has examples which automatically read the joystick while BASIC runs, using this principle. This is analogous to the use of a counter, except that the value is altered externally, not by increasing or decreasing a value from within the interrupt routine.

Flags can be used to vary the interrupt sequence. As a simple example, consider the following:

```
LDA  #0
BNE  ML
JMP  EAFB
```

As it stands, at each interrupt the branch will never be taken, and the effect is negligible. But a POKE into the location holding #0 will activate whatever ML has been put in. Using this approach, a change of background color or a screen reversal is easy to implement, and if the ML resets the flag to zero, the interrupt routine will be called just once.

The following is a simplified version of the music program in Chapter 13.

```
INT    LDA  $FB     ;FLAG TO PLAY SOUND
       BNE  PLAY    ;0=OFF
EXIT   JMP  $EABF   ;CONTINUE NORMAL IRQ PROCESSING
PLAY   STA  $900E   ;SET VOLUME = VALUE IN FLAG
       DEC  $F9     ;TEMPO CONTROL. COUNTS DOWN
       BNE  EXIT    ;TO ZERO
       LDA  $FA     ;REPLACE TEMPO COUNTER
       STA  $F9     ;WHEN IT REACHES ZERO
       LDX  $F8     ;POINTER TO NOTE TABLE
       LDA  $1D00,X ;LOAD NOTE
       STA  $900C   ;STORE IT IN SOUND REGISTER
       INC  $F8     ;INCREMENT POINTER TO NOTE
       JMP  $EABF   ;EXIT
```

This routine illustrates the important points of interrupt programming—except for polling, which might be used in a program to play notes only when a key is pressed. (See Chapter 13.)

$FB adjusts the volume of sound. When nonzero, $F9 counts down, controlling the tempo. When it times out, $FA replaces $F9, so $FA controls the tempo at which the notes are played. $F8 points to notes from a 256-byte note table; it simply cycles through the table, starting over whenever it exceeds 255. The routine exits to $EABF. The initializing routine, omitted here, simply points ($0314) to whatever address INT is, perhaps $1C00, after BASIC.

So far, we have discussed fast ML routines processed by interrupts. But suppose you happen to have a relatively slow routine (perhaps to fill a screen with graphics or perform a lot of calculations) which takes more time than about 1/60 second. Isn't there a chance that the interrupt might itself be interrupted, and the program therefore crash? In fact, the interrupt disable flag is set (in effect, SEI is executed) whenever an IRQ-type interrupt occurs, so this should not be a problem; on RTI, which returns to the preinterrupt state, the stored processor status flags are recovered, so there's no need to CLI on exit.

Chapter 9

Mixing BASIC with Machine Language

Mixing BASIC with Machine Language

While most programmers are happy to use BASIC, subroutines and utilities in ML offer increased speed and power. This short chapter explains the ways in which ML can be combined with BASIC, to retain the convenience of BASIC's LOAD, SAVE, and RUN commands.

BASIC Programs with Short ML Subroutines

Two fixed blocks of RAM (673–767 and 820–1023) are available for BASIC. Free zero page RAM includes 246–254, assuming RS-232 isn't used.

The second block of RAM (used primarily by tape) is large enough for significant ML routines. Examples include the vector-processing techniques described in the previous chapter; for consistency, all were written to start at 828. But all are relocatable and can be put anywhere in the buffer. Provided the vector points to their start, or SYS calls the correct address, they will still work.

The easiest way to mix ML routines with BASIC is to POKE them, reading the values to be POKEd from DATA statements. Program 9-1 is designed to convert ML into DATA statements and bypasses a lot of tedious typing.

Program 9-1. DATA Maker

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
100 INPUT "START";A                         :rem 111
110 INPUT "{2 SPACES}END";E                 :rem 189
120 INPUT "FIRST LINE#";L                   :rem 193
130 INPUT "LINE LENGTH";LL                   :rem 37
140 PRINT "{CLR}                            :rem 215
150 PRINT "{HOME}" L "DATA ";                :rem 95
160 PRINT MID$(STR$(PEEK(A)),2) ",";        :rem 161
170 A=A+1: IF A>E THEN END                   :rem 70
180 IF POS(0)<LL THEN GOTO 160              :rem 110
200 PRINT "{LEFT} {HOME}{3 DOWN}L="L"+1:A="A"
    {LEFT}:E="E"{LEFT}:LL="LL":GOTO140"     :rem 106
210 POKE 198,5: POKE 631,19: POKE 632,13: POKE 633
    ,13: POKE 634,13: POKE 635,13           :rem 180
```

To use "DATA Maker," suppose you have designed ML from 828 to about 900. Load and run this program, enter 828 and 910 as start and end addresses, select an initial line number (perhaps 0) and a suitable line length (so program listings can have lines of length suited to you), and watch the program write its own lines of DATA. If you weren't sure of the precise end address, edit the last few lines to remove extra data.

Delete the program manually, and the lines left store your ML in portable form. To POKE it back, you need FOR J=828 TO 900: READ X: POKE J,X: NEXT or something similar.

Several sets of ML (for example, PRINT USING and a joystick reader) can be put in the buffer together and converted to data. This method has the great advantage of being independent of memory changes; regardless of where BASIC starts and ends, SYS 828 remains fixed.

Other Methods

DATA statements take up space, and use of the tape buffer is impossible if a program is loaded from tape after the buffer was filled with ML. Thus, it's sometimes convenient to store data more economically within the program, or as a string of bytes, which occupy less space than DATA. But generally it's easier to put ML in the top of memory. There's also spare RAM from 1024 to 4095, if *both* 3K and 8K or 16K RAM cartridges are connected to the VIC-20. If you have this configuration, you'll have plenty of room for ML.

To store ML as a program line, use 0 REMXXXXXXXXXX and POKE the ML into the line itself. A SYS call will be able to run it as usual. There are a few complications; one is that the ML must not contain any zero byte, or BASIC will tend to treat it as an end-of-line and spoil the ML. Use LDX #1 then DEX in place of LDX #0, and so on.

Another complication is that the position of BASIC varies with expansion memory, so to make BASIC work for any VIC, you'll need to use SYS PEEK(43)+256* PEEK(44)+5 to call the routine. This finds the position of what was the first X after REM, to insure that the ML relocates.

A line treated in this way will list oddly, of course, and cannot be edited. You may prefer to include double quotes after REM so that the line lists without keywords.

Storing ML as a string (0 DATA 4C48D2AAD191D3 is typical) requires a few modifications to DATA Maker. Add these lines:

```
160 P=PEEK(A): Q%=P/16: P=P-16*Q%: REM Q% HI, P LO
162 Q%=Q%+48: IF Q>57 THEN Q%=Q%+7
164 PRINT CHR$(Q%);
166 P=P+48: IF P>57 THEN P=P+7
168 PRINT CHR$(P);
190 PRINT
```

Decoding such data needs a similar loop, where (typically) the line Q%=ASC(MID$(ML$,J)): P=ASC(MID$(ML$,J+1) recovers values from the string before each is converted to the range 0–15.

BASIC Programs with Long ML Subroutines

Long routines need a lot of RAM, and the only place to get it is usually at the top of BASIC's RAM. A few pointer changes are all that's necessary to convince VIC that its memory is less than it was. Then, ML DATA can be POKEd into the safe space after BASIC, or loaded directly as a block. The top of memory can vary, so to make your routines work with any VIC, you'll need the routine shown in Program 9-2.

Program 9-2. Changing VIC's RAM Pointers

```
100 T=PEEK(55) + 256*PEEK(56)
110 S=T-N
120 POKE 56,S/256:POKE 55,S-INT(S/256)*256:CLR
130 S=PEEK(55) + 256*PEEK(56)
```

The variable N in line 110 represents the number of bytes to lower the top of memory. Note that CLR sets all the pointers consistently. But it also deletes variables, so line 130 is needed to recover S.

Simpler, but less general, versions are given in Program 9-3 and Program 9-4.

Program 9-3. Lowering Memory by 256 Bytes

```
100 POKE 56,PEEK(56)-1:CLR
110 S=PEEK(55) + 256*PEEK(56)
```

Program 9-4. Lowering Memory in the Unexpanded VIC by 256 Bytes

```
100 POKE 56,29:CLR:S=7424
```

POKEing ML data into the lowered memory is straightforward. A new program can be loaded, leaving the ML intact. This is fine for ML which has been written to be relocatable (that is, ML using only branches and fixed locations). For example, JSR FFE4, followed by BEQ back to JSR FFE4, will work anywhere. But JSR $1EA0 or LDA $1E00,X can work only if they're put into the correct RAM position. If such commands are moved around in RAM, the absolute addresses no longer apply.

Relocating Loaders

To solve the problem, use a modified loader which alters addresses to fully relocate the ML. This is not too difficult and is worth doing if several utilities or ML programs are likely to be in RAM simultaneously. In that case, each can be stored before the previous one, and the relocatable format insures that each can run. But it's always simpler to make ML relocatable, if possible.

The loader is given in Program 9-5.

Program 9-5. Relocating Loader

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
100 T=PEEK(55) + 256* PEEK(56)                :rem 213
110 L=T-N                                     :rem 234
120 FOR J=L TO T-1:READ X%                    :rem 117
130 IF X%<0 THEN Y=X%+T: X%=Y/256:Z=Y-X%*256:POKE
    {SPACE}J,Z: J=J+1                          :rem 33
140 POKE J,X%:NEXT                             :rem 48
150 POKE 55,L-INT(L/256)*256:POKE 56,L/256:CLR
                                               :rem 34
```

To convert code into DATA which the relocating loader can use, follow these steps:

Enter the code into RAM and preferably test it.

Print (or write out) the disassembled version. A disassembler giving decimal values of locations is helpful.

Mark all the absolute addresses which need changing during relocation.

Replace each of them with its offset from the end of the program. Count from the end of program plus one backwards; the result will be a negative number from −1 to −30000 or so. The example shows how this is done; it is easier than you might think.

Convert the bytes into DATA statements and enter them. Note that, as a rule, each new negative value replaces two bytes.

Enter the value of N in line 110.

Test the loader. Run it several times, and check that each routine is independent and correctly set up.

The example shown in Figure 9-1 has a subroutine call, a table of byte values, and a branch. The branch, because of its relative addressing mode, relocates; so do the table, the implied-mode instructions, and the immediate-mode instruction. The only addresses to be relocated are those circled.

Figure 9-1. Relocatable Loader Example

```
 32 126 2  027A  JSR   027E
 96        027D  RTS
162   2    027E  LDX    #2
221 134 2  0280  CMP   0286,X
202        0283  DEX
208 250    0284  BNE    0280
 96        0286  RTS
 65  66    0287  .BYTE  $41, $42
```

Counting back from the end, note that 027E is the eleventh byte and 0286 is the third. Thus, −11 and −3 respectively replace all occurrences of those two addresses. The DATA statement is therefore

```
0 DATA 32,−11,96,162,2,221,−3,202,208,250,96,65,66
```

The number of bytes in the program is 15, so line 110 becomes

```
110 L=T-15
```

BASIC Which Is Mainly Machine Language

This section looks at programs which load normally but list as something like 0 SYS4624. Such programs (sometimes called hybrids) contain ML, packaged to imitate BASIC so they can be run in the usual way. They are structured with a line of BASIC followed by ML and are saved with pointers altered to straddle the whole program, BASIC and ML. The ML isn't visible on LIST because zero bytes after the SYS command cause LIST to act as though the end of the program had been found.

Before seeing how this is done, consider how it works in several examples.

Unlisted programs. The methods described in Chapter 6 allow BASIC to be converted to appear as a single SYS call. In these examples, ML is not of course the bulk of the program, but for security purposes it looks as though it might be. There's no way to tell without PEEKing around.

VICTERM. This pure ML program sits in memory at one place, just after the unexpanded VIC's LOAD address; it is not designed to be moved in memory. Many game programs are like this; LOAD simply puts the ML in memory, where it runs.

Disk Wedge. This is also pure ML, but like some BASIC utilities, it moves itself to the top end of memory, lowering pointers for security, and adding a wedge to BASIC to allow some disk commands to be simplified. After running, it returns to BASIC, with the utility installed.

"Tinymon" and "Super VICMON." These utilities move into the top of memory, too, and look exactly similar in operation to *Disk Wedge*. In fact, though they are not relocatable with simple POKEs, the necessary programming technique is more subtle. The hybrid program is BASIC with its SYS, followed by a loader, followed by the ML to be moved.

An Example of Mostly ML BASIC

In the following example, you'll see how *VICMON* itself can be saved as a BASIC program. The result needs some adaptations to work satisfactorily, and may not leave much RAM, but it shows the principles involved.

First, you need expansion RAM, say 8K, which makes BASIC start at $1200. Then enter the monitor and use the following steps

```
.T 6000 7000 1210
.N 1210 2210 B210 6000 7000
.N 208F 20D0 B210 6000 7000
W
```

to transfer the ML into the region $1210–$220F, altering addresses and tables where necessary. Now, a call to $1210, or SYS 4624 in BASIC, produces results like those you get with the monitor.

The next step is to put 0 SYS4624 in BASIC. Enter the following

```
.M    1200       1210
.1200 00 0A 12 00 00
.1205 9E 34  36 32 34
.120A 00 00  00 00 00
.120F 00 ---ML------
```

which represents a zero byte, the link address (120A), the line number (0), and SYS4624 as it is stored in BASIC. 9E is the token for SYS. This is followed by zero bytes, to simulate end-of-program, and the ML itself begins at $1210, ready to be called by SYS4624.

The only remaining step is to save this correctly, and the trick here is *not* to save the zero byte at the start. The format .S "MLM IN BASIC", 01,1201,2210 is correct; note the start address of $1201.

The same principles hold with any ML: Store it in memory, preferably near the start of BASIC, then add the BASIC line. If you have no monitor, you can enter BASIC, then alter locations 45 and 46 to include ML.

The method assumes that BASIC will be loaded back into the region it came from. Different memory arrangements may not work without a reconfiguring loader. Alternatively, SYS PEEK(43)+256*PEEK(44)+30 (or a similar expression) will find the ML whatever the LOAD address of BASIC, so if the ML is relocatable, it will always work.

Relocating Loaders

ML utilities can be moved in RAM just as BASIC ones can, and with negligible loss of time. The technique used in Program 9-6 is similar to that for BASIC. First, mark the absolute addresses needing relocation. Then add a zero byte immediately after each such address, and also after every genuine zero byte. This is much easier to do with an assembler.

Third, replace the addresses by their displacement from the end of the program (they convert to commands like LDA $FFD6 or JSR $FF65). Finally, put the BASIC call, the relocater (Program 9-6 gives the data for this), and the ML (preceded by a unique marker byte not found anywhere in the modified ML and which terminates the program) together in RAM, and save.

Program 9-6. Relocating ML

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
2 REM LINE 16'S 255 MARKS START BYTE OF ML :rem 41
10 DATA 165,45,133,34,165,46,133,35,165,55,133,36,
   165,56,133                              :rem 143
11 DATA 37,160,0,165,34,208,2,198,35,198,34,177,34
   ,208,60                                 :rem 253
12 DATA 165,34,208,2,198,35,198,34,177,34,240,33,1
   33,38,165                               :rem 103
13 DATA 34,208,2,198,35,198,34,177,34,24,101,36,17
   0,165,38                                :rem 50
14 DATA 101,37,72,165,55,208,2,198,56,198,55,104,1
   45,55,138                               :rem 108
15 DATA 72,165,55,208,2,198,56,198,55,104,145,55,5
   6,176,184                               :rem 123
16 DATA 201,255,208,237,165,55,133,51,165,56,133,5
   2,108,55,0                              :rem 137
19 DATA 55,56,176,184,201,255,208,237,165 :rem 245
20 DATA 55,133,51,165,56,133,52,108,55     :rem 75
```

Chapter 10

Vocabulary of the 6502 Chip

Chapter 10

Vocabulary of the 6502 Chip

This chapter lists each opcode with full details and helpful examples. The following conventions have been used:

:=
Read as "becomes." For example, A:=X means that the value in A becomes that currently in X.

x, 0, and 1
Show the effect of an opcode on the status flags. x means that the flag depends on the operation's result. 0 and 1 represent flags which an opcode always sets to 0 or 1 respectively. All other flags are left unchanged.

$ and %
Prefix hexadecimal and binary numbers; where these are omitted, a number is decimal.

A, X, and Y
The accumulator and the two index registers, X and Y.

M
Memory. This may be ROM in the case of LOAD instructions. Note that immediate addressing mode (#) loads from the byte immediately following the opcode in memory. All other addressing modes load from elsewhere in memory.

PSR (or SR)
The processor status register.

S
The location within the processor stack (locations $0100–$01FF) currently referenced by the stack pointer.

SP
The stack pointer.

PC
The program counter; this is composed of two eight-bit registers, PCL (program counter low byte) and PCH (program counter high byte).

ADC

Add memory plus carry to the accumulator. A:= A+M+C

Instruction	Addressing	Bytes	Cycles
$61 (97 %0110 0001)	ADC (zero page, X)	2	6
$65 (101 %0110 0101)	ADC zero page	2	3
$69 (105 %0110 1001)	ADC # immediate	2	2
$6D (109 %0110 1101)	ADC absolute	3	4
$71 (113 %0111 0001)	ADC (zero page),Y	2	5*
$75 (117 %0111 0101)	ADC zero page,X	2	4
$79 (121 %0111 1001)	ADC absolute,Y	3	4*
$7D (125 %0111 1101)	ADC absolute,X	3	4*

*Add 1 if page boundary crossed.

Flags:

N	V	—	B	D	I	Z	C
x	x					x	x

Operation: Adds together the current contents of the accumulator, the byte referenced by the opcode, and the carry bit. If the result is too large for a single byte, C is set to 1. If A holds zero (each bit equals zero), the Z flag is set to 1; otherwise, it is 0. If bit 7 in A is 1, the N flag is also set 1, to denote a negative value in A.

Uses:

1. Single-, double-, and multiple-byte additions. The carry bit automatically provides for overflow from one byte to the next. For example:

```
CLC          ; INSURES CARRY BIT IS 0
LDA  $4A     ; WE WISH TO ADD #$0A (10 DECIMAL) TO THE CONTENTS
ADC  #$0A    ; OF ($4A), I.E., THE DOUBLE-BYTE ADDRESS WHERE $4A
STA  $4A     ; IS THE LOW BYTE AND $4B THE HIGH BYTE
LDA  $4B
ADC  #$00    ; ADDS THE CARRY BIT WHERE APPLICABLE
STA  $4B     ; RESULT MUST BE STORED, ELSE IT WILL REMAIN ONLY IN A
```

2. Increasing or decreasing the accumulator. There is no INC A opcode.

```
CLC
ADC  #$01    ; INCREMENTS A; FF BECOMES 0.
```

3. In binary-coded decimal mode, obtained by setting D to 1, each nybble represents 0–9 and addition is corrected for this basis. This example adds 123 (decimal) to the contents of locations 0 and 1, which are assumed to contain, in ascending order, four binary-coded digits.

```
SED          ; SET THE DECIMAL FLAG
CLC          ; CLEAR CARRY FLAG
LDA  $01     ; WE'VE ASSUMED THE BCD DATA IS STORED IN NORMAL ORDER
ADC  #$23    ; WITH LOW BYTES FOLLOWING HIGHER ONES, NOT 6502 ORDER
STA  $01     ; ADD 23 DECIMAL
```

```
LDA  $00
ADC  #$01   ; ADD 01 DECIMAL PLUS POSSIBLY CARRY BIT EQUIVALENT TO 100
STA  $00
CLD         ; CLEAR THE DECIMAL BIT, UNLESS MORE DECIMAL MATH
              NEEDED
```

Notes: In decimal mode, the zero flag doesn't operate normally with ADC because of the automatic correction (adding 6) which the 6502 carries out. Testing for a zero result requires (for example) CMP #$00/ BEQ—which is an extra step not required in hexadecimal arithmetic.

The V flag is important if the twos complement convention is in use, and is set if the apparent sign of the result (bit 7) is not the true sign. In decimal mode, V is not used.

AND

Logical AND of memory with the accumulator. A:= A AND M

Instruction	Addressing	Bytes	Cycles
$21 (33 %0010 0001)	AND (zero page, X)	2	6
$25 (37 %0010 0101)	AND zero page	2	3
$29 (41 %0010 1001)	AND # immediate	2	2
$2D (45 %0010 1101)	AND absolute	3	4
$31 (49 %0011 0001)	AND (zero page),Y	2	5
$35 (53 %0011 0101)	AND zero page,X	2	4
$39 (57 %0011 1001)	AND absolute,Y	3	4*
$3D (61 %0011 1101)	AND absolute,X	3	4*

*Add 1 if page boundary crossed.

Flags:

N	V	—	B	D	I	Z	C
x						x	

Operation: Performs AND of the eight bits currently in the accumulator and the eight bits referenced by the opcode. When both bits are 1, the result is 1, but if either or both bits are zero, the result is 0. The resulting byte is stored in A. If A now holds 0—that is, all its bits are zero—the Z flag is set to 1; and if the high bit is set (bit 7 is 1), the negative flag N is set to 1. Otherwise, the flag is 0.

Uses:

1. Masking off unwanted bits, typically to test for the existence of a few high bits, or to test that some bits are zero:

```
LDA  $E081,X ; LOADS ACCUMULATOR FROM A TABLE OF CODED VALUES
AND  #$3F    ; TURNS OFF BITS 6 AND 7, LEAVING ALPHABETIC ASCII.
```

2. AND #$FF resets flags as though LDA had just occurred.
   AND #$00 has the same effect as LDA #$00.

ASL

Shift memory or accumulator left one bit.

$$\boxed{C} \leftarrow \boxed{7\ 6\ 5\ 4\ 3\ 2\ 1\ 0} \leftarrow \boxed{0}$$

Instruction	Addressing	Bytes	Cycles
$06 (6 %0000 0110)	ASL zero page	2	5
$0A (10 %0000 1010)	ASL accumulator	1	2
$0E (14 %0000 1110)	ASL absolute	3	6
$16 (22 %0001 0110)	ASL zero page,X	2	6
$1E (30 %0001 1110)	ASL absolute,X	3	7

Flags:

N	V	—	B	D	I	Z	C
x						x	x

Operation: Moves the content of memory or the accumulator left by one bit position, moving 0 into low bit, and the high bit into the carry flag. The carry bit therefore is set to 0 or 1 depending on bit 7 previously being 0 or 1. Z and N are set according to the result; thus Z can be true (that is, 1) only if the location or A held $00 or $80 before ASL. The N bit can be set true if bit 6 was previously 1.

Uses:

1. Doubles a byte (though not in decimal mode). If signed arithmetic is not being used, the result can safely reach values not exceeding 254, after which the carry must be taken into account, often with ROL. This example uses A from 1 to 127 to load two bytes from a table of address pointers and store them on the stack:

```
ASL  A
TAY
LDA  ADDHI,Y
PHA
LDA  ADDLO,Y
PHA
```

   The following example multiplies the contents of location $20 by 3, provided that the value it originally held was no greater than 85 decimal. In this case, the carry bit is automatically cleared by the shift:

```
LDA  $20
ASL  A
ADC  $20
```

2. Tests a bit by moving it into Z or N, to be followed by an appropriate branch. Note that four ASLs move the low nybble into the high nybble.

BCC

Branch if the carry bit is 0. PC:= PC + offset if C=0

Instruction	Addressing	Bytes	Cycles
$90 (144 %1001 0000)	BCC relative	2	2*

*Add 1 if branch occurs; add 1 more if the branch crosses a page.

Flags:

N V — B D I Z C

Operation: If C holds 0, the byte following the opcode is added to PC to calculate the address of the next opcode. If C holds 1, the program counter is unaffected. The effect is to cause a jump to the offset address when C is clear.

Uses:

1. As "branch always." If the carry bit is known to be clear, this command becomes effectively a "branch always" instruction. The flag may be set in a purely signaling sense, with no significance other than to show that one of two conditions applies. Many Kernal routines return with C clear if there were no errors, allowing JSR KERNAL/BCC OK followed by error-handling routines.
2. After previous operations. Usually the test is concerned with the result of a previous operation which may or may not set the carry flag. This compare routine is an example:

```
JSR  GETCHAR ; LOAD THE ACCUMULATOR WITH SOME VALUE, THEN
CMP  #$0A    ; COMPARE IT WITH DECIMAL 10.
BCC  LOW     ; BRANCH TO PROCESS VALUES 0-9,
             ; CONTINUE HERE WITH VALUES, 10-225
```

After any comparison, C is clear with a smaller value but is set with an equal or greater value. Bit 7 is irrelevant.

BCS

Branch if the carry bit is 1. PC:= PC + offset if C=1

Instruction	Addressing	Bytes	Cycles
$B0 (176 %1011 0000)	BCS relative	2	2*

*Add 1 if branch occurs; add 1 more if branch crosses a page.

Flags:

N V — B D I Z C

Operation: Identical to BCC, except that the branch is taken if C=1 and not C=0.

Uses: Identical to BCC. The choice between BCC and BCS at a branch point depends on convenience. For example, suppose a hardware port is to be read until bit 0 is set to 0. This routine:

```
LOOP LDA  PORT  ; READ LOCATION UNTIL XXXXXXX0
     LSR  A
     BCS  LOOP
```

is obviously tidier than:

```
LOOP LDA  PORT
     LSR  A
     BCC  NEXT
     BCS  LOOP
```

Similarly, JSR KERNAL/BCS ERROR followed by the normal processing path is probably preferable to the BCC version.

BEQ

Branch if zero flag is 1. PC:= PC + offset if Z=1

Instruction	Addressing	Bytes	Cycles
$F0 (240 %1111 0000)	BEQ relative	2	2*

*Add 1 if branch occurs; add 1 more if branch crosses a page.

Flags:

N	V	—	B	D	I	Z	C

Operation: If Z=1, the byte following the opcode is added, in twos complement arithmetic, to the program counter, which currently points to the next opcode. The effect is to cause a jump, forward or backward, up to a maximum of +127 or −128 locations if the zero flag is set. If Z=0 the branch is ignored.

Uses:

1. Common as an unconditional branch. It may be used to make routines relocatable, where the branch command isn't wide-ranging enough to span the program without an intermediate hop. The example inserts a couple of branches at a point where they will never be taken by the ML immediately before, and so are available as long branches.

   ```
   LDA  #$F5   ; NONZERO VALUE
   BEQ  BACK   ; THESE TWO BRANCHES
   BEQ  FWRD   ; RELY ON Z=1
   ```

2. To end a loop, either when a counter is decremented to zero, or because a zero byte is deliberately used as a terminator:

   ```
   LOOP LDA  TABLE,X   ; LOAD A WITH THE NEXT CHARACTER
        BEQ  EXIT      ; EXIT LOOP WHEN ZERO BYTE FOUND
        ... CONTINUE, E.G., STA OUTPUT,X/ INX/ BNE LOOP
   ```

3. After comparisons. BEQ is popular after comparisons because it's easy to use. For example, JSR GETCHR/ CMP #$2C/ BEQ COMMA looks for a comma in BASIC.

Notes: When a result is zero, the zero flag Z is made true (1). This point can be confusing. BEQ is usually read "branch if equal to zero," but when comparisons are being made it could read "branch if equal." The zero flag cannot be set directly (there is no SEZ instruction), but can be set only as the result of a location, register, or difference becoming zero.

BIT

Test memory bits. Z flag set on A AND M; N flag:= M7; V flag:= M6

Instruction	Addressing	Bytes	Cycles
$24 (36 %0010 0100)	BIT zero page	2	3
$2C (44 %0010 1100)	BIT absolute	3	4

Flags:

N	V	—	B	D	I	Z	C
M7	M6					x	

Operation: BIT affects only three flags, leaving registers and data unchanged. Z is set as if A AND M had been performed. If no bit position is 1 in both the memory location and A, then A AND M is zero and Z=1. Also, bits 6 and 7 are copied from memory to the V and N flags.

Uses:

1. Multiple entry points for subroutines. The three-byte absolute address BIT is the only instruction regularly used to provide alternate entry points for a routine. The example loads A with RETURN, space, or a cursor right depending on the entry point into the routine.

```
033C  LDA  #$0D    A9  0D       ; LDA #$0D
033E  BIT  #20A9   2C  A9  20   ; LDA #$20
0341  BIT  $1DA9   2C  A9  1D   ; LDA #$1D
```

   If the routine is entered with JSR $033C, the accumulator is loaded with $0D and the two BIT operations are then performed. These will change the settings of the status register flags, but will not affect the contents of the accumulator. If the routine is entered with JSR $033F, the routine begins with the A9 20 (LDA #$20) operation, and the contents of the accumulator will not be affected by the following BIT operation. A JSR $0342 will leave $1D in the accumulator.

   This is a compact way to load values into A (or X or Y). BIT $18, in the same way, alters three flags, but if entered at the $18 byte clears the carry flag. Both constructions are common in Commodore ROM, which explains why you may frequently see BIT instructions when you disassemble ROM.

2. Testing bits 7 and 6. BIT followed by BMI/BPL or BVC/BVS tests bits 7 and 6.

```
BIT   $0D
BMI   ERR
```

   This example tests location $0D, with a branch taken if it holds a negative twos complement value. Location $0D is in fact used to check for type mismatches. A value of $FF there denotes a string, $00 a numeric variable, so BMI occurs with strings.

3. Used as AND without affecting the accumulator. The following example shows the AND feature in use. CHRFLG holds 0 if no character is to be output, and $FF otherwise. Assuming the accumulator holds a nonzero value, BIT tests whether to branch past the output routine, while retaining A's value.

```
LDA  VALUE
BIT  CHRFLG
BEQ  NOTOUT
```

BMI

Branch if the N flag is 1. PC:= PC + offset if N=1

Instruction	Addressing	Bytes	Cycles
$30 (48 %0011 0000)	BMI relative	2	2*

*Add 1 if branch occurs; add 1 more if branch crosses a page.

Flags:

N	V	—	B	D	I	Z	C

Operation: If the N flag is set, the byte following the opcode is added to the program counter in twos complement form. The effect is to force a jump to the new address. The maximum range of a branch is −128 to +127 locations. When N is clear, the branch command is ignored.

Uses:

1. Testing bit 7 of a location. For example:

```
LOOP BIT  PORT ;TEST BITS OF A HARDWARE PORT (PRESERVING VALUE
                IN A)
     BMI  LOOP ;WAIT UNTIL BIT 7 OF THE PORT IS 0
```

2. Conventional use. Like the other flags, N may be used in a purely conventional sense. As an example, consider BASIC's keyword tokens. All have values, in decimal, of 128 or more, which keeps keywords logically separate from other BASIC and also permits instructions like this:

```
LDA NEXT   ; LOAD NEXT CHR INTO ACCUMULATOR
BMI  TOKEN ; BRANCH TO PROCESS A KEYWORD
           ; OTHERWISE, PROCESS DATA AND EXPRESSIONS
```

Notes:

1. It's important to realize that the minus in BMI refers only to the use of bit 7 to denote a negative number in twos complement arithmetic. Comparisons (for example, with CMP) followed by BMI implicitly use bit 7. Mostly it is easier to think of this operation as "branch if the high bit is set."
2. BPL is exactly the opposite of BMI. Where one branches, the other does not.

BNE

Branch if Z is 0. PC:= PC + offset if Z=0

Instruction	Addressing	Bytes	Cycles
$D0 (208 %1101 0000)	BNE relative	2	2*

*Add 1 if branch occurs; add 1 more if branch crosses a page.

Flags:

N	V	—	B	D	I	Z	C

Operation: BNE operates exactly like BEQ, except that the condition is opposite. If Z=0 the offset contained in the byte after BNE is added to the program counter, so the branch takes place. If Z=1 the branch is ignored.

Uses:

1. In unconditional branches. BNE may be used in unconditional branches in circumstances like those which apply to BEQ.
2. In a loop, where a counter is being decremented. BNE is very often used in a loop in which a counter is being decremented. This is probably the easiest type of loop to write. Watch the data's starting address, as offset 0 isn't executed by a loop like this. The example prints ten characters from a table, their offsets being 10, 9, 8, ... 2, 1.

```
     LDX  #$0A
LOOP LDA  TABLE,X
     JSR  OUTPUT
     DEX
     BNE  LOOP
```

3. After comparisons. BNE, like BEQ, is popular after comparisons:

```
B4C0  LDA  $C1     ;CHECK CONTENTS OF $C1
B4C2  CMP  #$42    ;IS IT B?
B4C4  BNE  $B4C9   ;BRANCH IF NOT
```

Notes: When a result is nonzero, the zero flag Z is made false (set to 0). This can be confusing. BNE is usually read "branch if not equal to zero." The result of a comparison is zero if both bytes are identical, because one is subtracted from the other, so "branch if not equal" is an optional alternative.

BPL

Branch if the N flag is 0. PC:= PC + offset if N=0

Instruction	Addressing	Bytes	Cycles
$10 (16 %0001 0000)	BPL relative	2	2*

*Add 1 if branch occurs; add 1 more if branch crosses a page.

Flags:

N	V	—	B	D	I	Z	C

Operation: BPL operates exactly like BMI, except that the condition is opposite. The branch is taken to the new address given by program counter plus offset if N=0. This means that if the result was positive or zero, the branch is taken.

Uses:

1. In testing bit 7 of a memory location. This code, for example waits until the accumulator holds a byte with bit 7 on. Such a location must be interrupt- or hardware-controlled, not just RAM.

```
LOOP LDA  TESTLOCN
     BPL  LOOP
```

2. Testing for the end of a loop. Where a counter is being decremented, and the counter's value 0 is needed, this command can be useful. This simple loop prints ten bytes to screen:

```
     LDX  #$09     ;X REGISTER WILL COUNT 9,8,7, ... ,1,0
LOOP LDA  BASE,X   ;"BASE" IS THE STARTING ADDRESS OF THE 10 BYTES
     STA  $1E00,X  ;START OF SCREEN (UNEXPANDED VIC)
     DEX           ; DECREMENT X
     BPL  LOOP     ; BRANCH WHEN POSITIVE OR ZERO
```

BRK

Force break. S:= PCH, SP:= SP−1, S:= PCL, SP:= SP−1, S:= PSR, SP:= SP−1, PCL:= $FFFE, PCH:= $FFFF

Instruction	Addressing	Bytes	Cycles
$00 (0 %0000 0000)	BRK implied	1	7

Flags:

N	V	—	B	D	I	Z	C
			1		1		

Operation: BRK is a forced interrupt, which saves the current program counter and status register values and jumps to a standard address. Note that the value saved for the program counter points to the BRK byte plus two (like a branch) and that the processor status register on the stack has flag B set to 1.

The IRQ service routine behaves like BRK. The break flag is a sort of designer's patch so that BRK can be recognized as different from IRQ interrupts.

Uses:

1. With ML monitors. BRK is mainly used with ML monitors. The ML stops when BRK is encountered, and the vector points back to the monitor, typically printing the current values of the program counter, flags' status register, A, X, Y, stack pointer, and possibly other ML variables.

In the VIC's ROM, locations $FFFE and $FFFF point to a routine beginning at $FF72. If the B flag is set, a jump is made through a vector at location $0316, so the BRK handling routine can be modified by changing the values in $0316 and $0317. Altering these locations to point to the monitor is a function of initialization of the monitor; it isn't inherent in the system that BRK behaves like that. BRK is valuable when developing ML programs.

2. Monitors can be entered from BASIC if $0316/0317 points to their start. POKE 790,0: POKE 791,96, for example, points this vector to $6000, and SYS 13 (or a SYS to any location containing a zero byte) enters a monitor there. Usually $0316/0317 points to a ROM routine used by STOP-RESTORE which resets I/O and Kernal pointers. BRK is not widely used in ML that must interact directly with BASIC.

BVC

Branch if the internal overflow flag (V) is 0. PC:=PC + offset if V=0

Instruction	Addressing	Bytes	Cycles
$50 (80 %0101 0000)	BVC relative	2	2*

*Add 1 if branch occurs; add 1 more if branch crosses a page.

Flags:

N	V	—	B	D	I	Z	C

Operation: If V is clear, the byte following the opcode is added, as a twos complement number, to the program counter, set to point at the following instruction. The effect is to jump to a new address. If V=1, the next instruction is processed and the branch ignored.

Uses:

1. As a "branch always" instruction. For instance:

```
CLV
BVC  LOAD
```

2. With signed arithmetic, to detect overflow from bit 6 into bit 7, giving a spurious negative bit. This is rarely used since the sign of a number can be held elsewhere, so that ordinary arithmetic can be used without the complication of the V bit.

   The following routine adds two numbers in twos complement form; the numbers must therefore be in the range −128 to 127. CLC is necessary; otherwise, it may add 1 to the result. Overflow will occur if the total exceeds 127 or is less than −128.

```
LDA  ADD1
CLC
ADC  ADD2
BVC  OK
JMP  OVERFL
```

3. Testing bit 6. BIT copies bit 6 of the specified location into the V flag of the processor status register, so BVC or BVS can be used to test bit 6. For example, the following routine waits until the hardware sets bit 6 of hardware location PORT to 1.

```
F103   BIT   PORT
F106   BVC   $F103
```

BVS

Branch if the internal overflow flag (V) is 1. PC:= PC + offset if V=1

Instruction	Addressing	Bytes	Cycles
$70 (112 %0111 0000)	BVS relative	2	2*

*Add 1 if branch occurs; add 1 more if branch crosses a page.

Flags:

N	V	—	B	D	I	Z	C

Operation: This branch is identical to BVC except that the test logic to decide whether the branch is taken is opposite.

CLC

Clear the carry flag. C:= 0

Instruction	Addressing	Bytes	Cycles
$18 (24 %0001 1000)	CLC implied	1	2

Flags:

N	V	—	B	D	I	Z	C
							0

Operation: The carry flag is set 0. All other flags are unchanged.

Uses: The carry bit is automatically included in add and subtract commands (ADC and SBC), so that accurate calculations require the flag to be in a known state. CLC is the usual preliminary to additions:

```
CLC
LDA  #$02
ADC  #$02
JSR  PRINT
```

After CLC, this routine adds 2 and 2 and prints the resulting byte 4. In multiple-byte additions, C is cleared at the start but is subsequently used to carry through the overflows, if they exist.

CLD

Clear the decimal flag. D:= 0

Instruction	Addressing	Bytes	Cycles
$D8 (216 %1101 1000)	CLD implied	1	2

Flags:

N	V	—	B	D	I	Z	C
				0			

Operation: The decimal flag is set 0; all other flags are unchanged.

Uses: Resets the mode for ADC and SBC so that hexadecimal arithmetic is performed, not binary-coded decimal. Typically, SED precedes some decimal calculation, with CLD following when this is finished.

Notes: BASIC uses no decimal mode calculations; on switching the machine on, CLD is executed and the flag is left off. ML monitors clear the flag on entry too.

CLI

Clear the interrupt disable flag. I:= 0

Instruction	Addressing	Bytes	Cycles
$58 (88 %0101 1000)	CLI implied	1	2

Flags:

N	V	—	B	D	I	Z	C
					0		

Operation: The interrupt disable flag is set to 0. From now on, IRQ interrupts will take place and be processed by the system.

Notes:

1. Interrupts through the NMI line (non-maskable interrupts) take place irrespective of the I flag.
2. Typically, CLI is used after SEI plus changes to interrupt vectors. Often, CLI isn't needed when used with BASIC, as a number of BASIC routines themselves use CLI.

CLV

Clear the internal overflow flag. V:= 0

Instruction	Addressing	Bytes	Cycles
$B8 (184 %1011 1000)	CLV implied	1	2

Flags:

N	V	—	B	D	I	Z	C
	0						

Operation: Sets V to 0.
Notes: CLV is used only in "branch always" instructions (for example, CLV/BVC). Unlike C, V isn't added to results, so clearing is not necessary before calculations.

CMP

Compare memory with the contents of the accumulator. PSR set by A−M

Instruction	Addressing	Bytes	Cycles
$C1 (193 %1100 0001)	CMP (zero page,X)	2	6
$C5 (197 %1100 0101)	CMP zero page	2	3
$C9 (201 %1100 1001)	CMP # immediate	2	2
$CD (205 %1100 1101)	CMP absolute	3	4
$D1 (209 %1101 0001)	CMP (zero page),Y	2	5*
$D5 (213 %1101 0101)	CMP zero page,X	2	4
$D9 (217 %1101 1001)	CMP absolute,Y	3	4*
$DD (221 %1101 1101)	CMP absolute,X	3	4*

*Add 1 if page boundary is crossed.

Flags:

N	V	—	B	D	I	Z	C
x						x	x

Operation: CMP affects three flags only, leaving registers and data intact. The accumulator is not changed. The byte at the address specified by the opcode is subtracted from A, and the three flags N, Z, and C are set depending on the result. Thus, if the accumulator holds the same value as the memory location, the result is zero and the zero flag is set.

Within the chip, what happens is that the value in the accumulator is added to the twos complement of the data. The result of this determines how the flags are set.

Uses:

1. With the zero flag, Z. This is the easiest flag to use with CMP. Z=0 after a CMP means the two values were equal.

```
FF22  JSR  $FFCF  ;INPUT A CHARACTER
FF25  CMP  #$20   ;IS IT A SPACE?
FF27  BEQ  $FF22  ;YES. INPUT AGAIN
FF29  CMP  #$OD   ;IS IT C.RETURN?
FF2B  BEQ  $FF47  ;YES. BRANCH ...
FF2D  CMP  #$22   ;..NO. IS IT QUOTES? ETC.
```

This is part of a ROM routine to search through BASIC lines from the keyboard buffer for particular characters such as spaces, RETURNs, and quotes, which require special handling.

2. With the carry flag, C. If the value of the byte is less than A, or equal to A, the carry flag is set; that is, C=0 (tested with BCC) after a CMP means that A<M, while C=1 (tested with BCS) indicates that A≥M. Here, "less than" is in the absolute sense, not the twos complement sense. Thus, 100 is less than 190, although in twos complement notation, 190 (being negative) would count as the smaller number of the two.

   The following example shows how a range of values may be tested for and processed. Starting with the lowest ranges, comparisons are carried out until the correct range is found. Each comparison is followed by a branch to B1, B2, etc., where processing is carried out for 0–$1F, $20–$3F, and so on.

```
LDY  #$00
LDA  (PTR),Y
CMP  #$20
BCC  B1
CMP  #$40
BCC  B2
```

3. With the negative flag, N. This is the trickiest flag to use with CMP. The reason is that twos complement numbers are assumed, and if you are working with these, CMP operates as expected, subtracting the memory from the accumulator. If both numbers are positive, or both negative, the N flag is set as though absolute subtraction were being used, and in these circumstances BMI/BPL can be used. But if the two data items have different signs, the comparison process is complicated by the fact that the V bit may register internal overflow. Generally, use the carry flag.

CPX

Compare memory with the contents of the X register. PSR set by X−M

Instruction	Addressing	Bytes	Cycles
$E0 (224 %1110 0000)	CPX # immediate	2	2
$E4 (228 %1110 0100)	CPX zero page	2	3
$EC (236 %1110 1100)	CPX absolute	3	4

Flags:

N	V	—	B	D	I	Z	C
x						x	x

Operation: CPX affects three flags only, leaving the registers and data intact. The byte referenced by the opcode is subtracted from the contents of the X register, and the flags N, Z, and C are set depending on the result. The value in X is not affected. Within the chip, X is added to the twos complement of the data, and the result determines how the flags are set.

Uses:

1. With the zero flag, Z. This flag tests equality.

```
     LDX  #$00
LOOP LDA  $0278,X
     STA  $0277,X
     INX
     CPX  $C6
     BNE  LOOP
```

   The loop in this example is part of the keyboard buffer processing, showing how the contents of the buffer are shifted one character at a time. $C6 is a zero page location, updated whenever a new character is keyed in, which holds the current number of characters in the buffer. The comparison provides a test to end the loop.

2. With the carry flag, C. This flag tests for X≥M and X<M.

```
LDX  $FE
CPX  #$15
BCS  EXIT; IF X>21
...
```

   The test routine is part of a graphics plot program; location $FE holds the horizontal coordinate, which is to be in the range 0–21 to fit the screen. The comparison causes exit, without plotting, when X holds 22–255.

3. With the negative flag, N. When X and the data have the same sign (both are 0–127 or 128–255), then BMI has the same effect as BCC, and vice versa. When the signs are opposite, the process is complicated by the possibility of overflow into bit 7. For example, 78 compared with 225 sets N=0, but 127 compared with 255 sets N=1. (Note that 225=−31 as a twos complement number; thus 78+31=109 with N=0, but 127+31=158 with N=1.)

CPY

Compare memory with the contents of the Y register. PSR set by Y−M.

Instruction	Addressing	Bytes	Cycles
$C0 (192 %1100 0000)	CPY # immediate	2	2
$C4 (196 %1100 0100)	CPY zero page	2	3
$CC (204 %1100 1100)	CPY absolute	3	4

Flags:

N	V	—	B	D	I	Z	C
x						x	x

Operation: CPY affects three flags only, leaving the registers and data intact. The byte referenced by the opcode is subtracted from Y, and the flags N, Z, and C are set depending on the result. Apart from the use of Y in place of X, with the resulting symmetry in the implementation of addressing, this opcode is identical in its effects to CPX.

Notes: The major difference in addressing between X and Y is the fact that post-indexing of indirect addresses is available only with Y. This type of construction, in which a set of consecutive bytes (perhaps a string in RAM or an error message) is processed up to some known length, tends to use the Y register.

```
     LDY  #$00
LOOP LDA  (PTR),Y
     JSR  OUTPUT
     INY
     CPY  LENGTH
     BNE  LOOP
```

DEC

Decrement contents of memory location. M:=M−1

Instruction	Addressing	Bytes	Cycles
$C6 (198 %1100 0110)	DEC zero page	2	5
$CE (206 %1100 1110)	DEC absolute	3	6
$D6 (214 %1101 0110)	DEC zero page,X	2	6
$DE (222 %1101 1110)	DEC absolute,X	3	7

Flags:

N	V	—	B	D	I	Z	C
x						x	

Operation: The byte referenced by the addressing mode is decremented by 1, conditioning the N flag and the Z flag. If the byte contains a value from $81 to $00 after DEC, the N flag will be set. The Z flag will be 0 except for the one case where the location held $01 before the decrement. DEC is performed within the chip itself by adding $FF to the contents of the specified location, setting N and Z on the result.

The carry bit is unchanged regardless of the outcome of DEC.

Uses:

1. To decrement a double-byte value.

```
LDA  $93
BNE  +2
DEC  $94
DEC  $93
```

   This short routine shows an efficient method to decrement a zero page pointer or any other double-byte value. It uses the fact that the high byte must be decremented only if the low byte is exactly zero.

2. Implementing other counters. Counters other than the X register and Y register can easily be implemented with this command (or INC). Such counters must be in RAM. DEC cannot be used to decrement the contents of the accumulator. This simple delay loop which decrements locations $FB and $FC shows an example:

```
     AND  #$00  ;FOR A CHANGE
     STA  $FB   ;SET THESE BOTH
     STA  $FC   ;TO ZERO
LOOP DEC  $FB
     BNE  LOOP  ;255 LOOPS...
     DEC  $FC
     BNE  LOOP  ;... BY 255
```

A zero page decrement takes five clock cycles to carry out; a successful branch takes three (assuming a page boundary isn't crossed). The inside loop therefore takes 8*255 cycles to complete, and the whole loop requires a little more than 8*255*255 cycles. Divide this by a million to get the actual time in seconds, which is about half a second.

DEX

Decrement the contents of the X register. X:= X−1

Instruction	Addressing	Bytes	Cycles
$CA (202 %1100 1010)	DEX implied	1	2

Flags:

N	V	—	B	D	I	Z	C
x						x	

Operation: The value in the X register is decremented by 1, setting the N flag if the result has bit 7 set, and setting the Z flag if the result is 0. As with DEC, the carry bit is unaltered.

Uses: To count X in a loop. DEX is almost exclusively used to count X in a loop. Its maximum range, 255 bytes, is often insufficient, so several loops may be necessary. This routine moves 28 bytes from ROM to RAM, including the CHRGET routine.

```
     LDX  #$1C
NEXT LDA  E378,X
     STA  $73,X
     DEX
     BNE  NEXT
```

DEY

Decrement the contents of the Y register. Y:= Y−1

Instruction	Addressing	Bytes	Cycles
$88 (136 %1000 1000)	DEY implied	1	2

Flags:

N	V	—	B	D	I	Z	C
x						x	

Operation: The value in the Y register is decremented by 1, setting the N flag if the result has bit 7 set (that is, is greater than 127), and setting the Z flag if the result is 0. As with DEC, the carry bit is unaltered.

Uses: Counting within loops. DEY, like DEX, is almost exclusively used to count within loops. There are more opcodes which have indexing by X than by Y, so X is more popular for this purpose. The example uses Y to count from 2 to 0.

```
LDY  #$02
LDA  (PTR),Y  ;LOAD 2ND BYTE
DEY
ORA  (PTR),Y  ;ORA 1ST BYTE
DEY
ORA  (PTR),Y  ;ORA 0TH BYTE
BNE  CONT     ;END IF ZERO
```

This inclusively ORs together three adjacent bytes; if the result is 0, each of the three must have been a zero.

EOR

The byte in the accumulator is exclusively ORed bitwise with the contents of memory. A:= A EOR M

Instruction	Addressing	Bytes	Cycles
$41 (65 %0100 0001)	EOR (zero page,X)	2	6
$45 (69 %0100 0101)	EOR zero page	2	3
$49 (73 %0100 1001	EOR # immediate	2	2
$4D (77 %0100 1101)	EOR absolute	3	4
$51 (81 %0101 0001)	EOR (zero page),Y	2	5*
$55 (85 %0101 0101)	EOR zero page,X	2	4
$59 (89 %0101 1001)	EOR absolute,Y	3	4*
$5D (93 %0101 1101)	EOR absolute,X	3	4*

*Add 1 if page boundary is crossed.

Flags:

N	V	—	B	D	I	Z	C
x						x	

Operation: An exclusive OR (compare ORA for a description of an inclusive OR) is a logical operation in which bits are compared, and EOR is considered to be true if A or B—but not both or neither—is true. For example, consider $AB EOR $5F. The byte $AB is %1010 1011, and $5F is %0101 1111. So the EOR of these two is

%1111 0100, or $F4. You get this result by a process of bit comparisons, where bit 7 is 0 EOR 1=1, and so on.

Uses:

1. Reversing a bit. EORing a bit with 0 leaves the bit unaffected; EORing a bit with 1 flips the bit.

```
LDA  LOCN
EOR  #$02    ;FLIPS BIT 1
STA  LOCN
```

   The example shows how a single bit can be reversed. To reverse an entire byte, use EOR #$FF; to reverse bit 7, use EOR #$80.

2. In hash totals and encryption algorithms. Hash totals and encryption algorithms often use EOR. For example, if you have a message you wish to conceal, you can EOR each byte with a section of ROM or with bytes generated by some repeatable process. The message is recoverable with the same EOR sequence.

INC

Increment contents of memory location. M:= M+1

Instruction	Addressing	Bytes	Cycles
$E6 (230 %1110 0110)	INC zero page	2	5
$EE (238 %1110 1110)	INC absolute	3	6
$F6 (246 %1111 0110)	INC zero page,X	2	6
$FE (254 %1111 1110)	INC absolute,X	3	7

Flags:

N	V	—	B	D	I	Z	C
x						x	

Operation: The byte referenced by the addressing mode is incremented by 1, setting the N flag and the Z flag. The N flag will be 1 if the high bit of the byte is 1 after the INC, and otherwise 0. The Z flag will be 1 only if the location held $FF before the INC. The carry bit is unchanged.

Uses:

1. Incrementing a double-byte value. This short routine shows an efficient method to increment a zero page pointer or any other double-byte value. The high byte is incremented only when the low byte changes from $FF to $00.

```
INC  $FB
BNE  CONT
INC  $FC     CONT  ...
```

2. Implementing counters in RAM. INC may be used to implement counters in RAM where the X and Y registers are insufficient. Suppose we use the IRQ interrupt servicing to change a tune regularly.

```
IRQ  INC  $FE
     BEQ  +3
     JMP  IRQCONT
     LDA  #20
     STA  $FE
```

Where IRQCONT is the interrupt's usual routine, this allows some periodic routine to be performed. Here, the zero page location $FE is used to count from $20 up to $FF and $00, so the processing occurs every 255−32=223 jiffies—about every 3.7 seconds.

Notes:

1. The accumulator can't be incremented with INC. CLC/ADC #$01 or SEC/ADC #$00 must be used. TAX/ INX/ TXA or some other variation may also be used.
2. Remember that INC doesn't load the contents of the location to be incremented into any of the registers. If the incremented value is wanted in A, X, or Y, then INC $C6 must be followed by LDA $C6, LDX $C6, or LDY $C6.

INX

Increment the contents of the X register. X:= X+1

Instruction	Addressing	Bytes	Cycles
$E8 (232 %1110 1000)	INX implied	1	2

Flags:

N	V	—	B	D	I	Z	C
x						x	

Operation: The byte in the X register is incremented by 1, setting the N flag if the result has bit 7 set, and the Z flag if the result is zero. These flags may both be 0, or one of them may be 1; it is impossible for both to be set 1 by this command. The carry bit is unchanged.

Uses: As a loop variable. INX is common as a loop variable. It is also often used to set miscellaneous values which happen to be near each other, for example:

```
LDX  #$00
STX  $033A
STX  $033C
INX
STX  $10
```

Stack-pointer processing tends to be connected with the use of the X register, because TXS and TSX are the only ways of accessing SP.

INY

Increment the contents of the Y register. Y:= Y+1

Instruction	Addressing	Bytes	Cycles
$C8 (200 %1100 1000)	INY implied	1	2

Flags:

N	V	—	B	D	I	Z	C
x						x	

Operation: The byte in the Y register is incremented by 1, setting N=1 if the result has bit 7=1 (and vice versa) and setting Z=1 if the result is zero (and vice versa). A zero result is obtained by incrementing $FF. Note that the carry bit is unchanged in all cases.

Uses: To control loops. Like DEX, DEY, and INX, this command is often used to control loops. It is often followed by a comparison, CPY, to check whether its exit value has been reached.

JMP

Jump to a new location anywhere in memory. PC:= M

Instruction	Addressing	Bytes	Cycles
$4C (76 %0100 1100)	JMP absolute	3	3
$6C (108 %0110 1100)	JMP (absolute)	3	5

Flags:

N	V	—	B	D	I	Z	C

Operation: JMP is the 6502 equivalent of a GOTO, transferring control to some other part of the program. An absolute JMP, opcode $4C, transfers the next byte to the low byte of PC, and the next to highest byte of PC, causing an unconditional jump.

The indirect absolute jump is more elaborate and takes longer. PCL and PCH are loaded from the address following JMP and from the next address respectively. This is the only absolute indirect command available on the 6502.

Uses: JMP, unlike JSR, keeps no record of its present position; control is just shifted to another part of a program. Branch instructions are preferable if ML is required to work even when moved around in memory, except for JMPs to fixed locations like ROM.

```
CMP  #$2C     ; IS IT COMMA?
BEQ  +3
JMP  ERROR
```

The example is part of a subroutine which checks for a comma in a BASIC line; if the comma has been omitted, an error message is printed.

Notes:

1. Indirect addressing. This is a three-byte command that takes the form JMP ($0072) or JMP ($7FF0). A concrete example is the IRQ vector. When a hardware interrupt occurs, an indirect jump to ($3014) takes place. A look at this region of RAM with a monitor reveals something like this:

   ```
   0314 3C 03 97 FF 47 FE
   ```

   So JMP ($0314) is equivalent to JMP $033C in this instance. Pairs of bytes can be collected together to form an indirect jump table. Note that this instruction has a bug; JMP ($02FF) takes its new address from $02FF and $0200, not $0300.
2. A subroutine call followed by a return is exactly identical to a jump, except that the stack use is less and the timing is shorter. Replacing JSR CHECK/ RTS by JMP CHECK is a common trick.

JSR

Jump to a new memory location, saving the return address. S:= PC+2 H, SP:= SP−1, S:= PC+2 L, SP:= SP−1, PC:= M

Instruction	Addressing	Bytes	Cycles
$20 (32 %0010 0000)	JSR absolute	3	6

Flags:

N	V	—	B	D	I	Z	C

Operation: JSR is the 6502 equivalent of a GOSUB, transferring control to another part of the program until an RTS is met, which has an effect like RETURN. Like BRK, this instruction saves PC+2 on the stack, which points to the last byte of the JSR command. RTS therefore has to increment the stored value in order to execute a correct return. Note that no flags are changed by JSR. RTS also leaves flags unaltered, making JSR $FFC0/ BCC, for example, feasible.

Uses:

1. Breaking programs into subroutines. JSR allows programs to be separated into subroutines, which is a very valuable feature. The Kernal commands, all of which are called as subroutines by JSR, illustrate the convenience which subroutines bring to programming. Neither JSR nor RTS sets flags, so LDA #$0D/ JSR $FFD2 (Kernal output routine) successfully transfers the accumulator contents—in this case, a RETURN character—since the carry flag status is transferred back after RTS.

   ```
   LOOP JSR  $FFE4   ;GET RETURNS A=0
        BEQ  LOOP    ;IF NO KEY IS PRESSED
        STA  BUFFER  ;WE HAVE A KEY: PROCESS IT
   ```

   The example uses a Kernal subroutine which gets a character, usually from the keyboard. The subroutine is a self-contained unit. Chapter 8 has examples in

which several JSR calls follow each other, performing a series of operations between them.
2. Other applications. See RTS for the PLA/ PLA construction which pops one subroutine return address from the stack. RTS also explains the special construction in which an address (minus 1) is pushed onto the stack, generating a jump when RTS occurs. Finally, see JMP for a note on the way in which JSR/RTS may be replaced by JMP.

LDA

Load the accumulator with a byte from memory. A:= M

Instruction	Addressing	Bytes	Cycles
$A1 (161 %1010 0001)	LDA (zero page,X)	2	6
$A5 (165 %1010 0101)	LDA zero page	2	3
$A9 (169 %1010 1001)	LDA # immediate	2	2
$AD (173 %1010 1101)	LDA absolute	3	4
$B1 (177 %1011 0001)	LDA (zero page),Y	2	5*
$B5 (181 %1011 0101)	LDA zero page,X	2	4
$B9 (185 %1011 1001)	LDA absolute,Y	3	4*
$BD (189 %1011 1101)	LDA absolute,X	3	4*

*Add 1 if page boundary crossed.

Flags:

N	V	—	B	D	I	Z	C
x						x	

Operation: Loads the accumulator with the contents of the specified memory location. The zero flag Z is set to 1 if the accumulator now holds zero (all bits loaded are 0s). Bit 7 is copied into the N (negative) flag. No other flags are altered.

Uses:
1. General transfer of data from one part of memory to another. Such transfer needs a temporary intermediate storage location, which A (or X or Y) can be. As an example, this program transfers 256 consecutive bytes of data beginning at $7000 to an area beginning at $8000. The accumulator is alternately loaded with data and written to memory.

```
LDX  #00
LDA  $7000,X
STA  $8000,X
DEX
BNE  -9
```

2. Binary operations. Some binary operations use the accumulator. ADC, SBC, and CMP all require A to be loaded before adding, subtracting, or comparing. The addition (or whatever) can't be made directly between two RAM locations, so LDA is essential.

```
LDA  $C5    ; WHICH KEY?
CMP  #$40   ; PERHAPS NONE?
BNE  KEY    ; BRANCH IF KEY
```

3. Setting chip registers. Sometimes a chip register is set by reading from it; this explains some LDA commands in initialization routines with no apparent purpose.

LDX

Load the X register with a byte from memory. X:= M

Instruction	Addressing	Bytes	Cycles
$A2 (162 %1010 0001)	LDX # immediate	2	2
$A6 (166 %1010 0101)	LDX zero page	2	3
$AE (174 %1010 1110)	LDX absolute	3	4
$B6 (182 %1011 0101)	LDX zero page,Y	2	4
$BE (190 %1011 1110)	LDX absolute,Y	3	4*

*Add 1 if page boundary crossed.

Flags:

N	V	—	B	D	I	Z	C
x						x	

Operation: Loads X from memory and sets Z=1 if X holds zero. Bit 7 from the memory is also copied into N. No other flags are altered.

Uses:

1. Transfer of data and holding temporary values. These applications closely resemble LDA.
2. Offset with indexed addressing. X has two characteristics which distinguish it from A: It is in direct communication with the stack pointer, and it can be used as an offset with indexed addressing. There are other differences too. Constructions like LDX #$FF/ TXS and LDX #$00/.../ DEX/ BNE are common.

LDY

Load the Y register with a byte from memory. Y:= M

Instruction	Addressing	Bytes	Cycles
$A0 (160 %1010 0000)	LDY # immediate	2	2
$A4 (164 %1010 0100)	LDY zero page	2	3
$AC (172 %1010 1100)	LDY absolute	3	4
$B4 (180 %1011 0100)	LDY zero page,X	2	4
$BC (188 %1011 1100)	LDY absolute,X	3	4*

*Add 1 if page boundary crossed.

Flags:

N	V	—	B	D	I	Z	C
x						x	

Operation: Loads Y from memory and sets Z=1 if Y now holds zero. Bit 7 from memory is copied into N. No other flags are altered.

Uses:

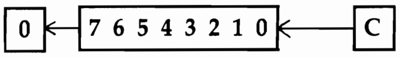
1. Transfer of data and storage of temporary values.
2. Loops. Since Y can be used as an index, and can be incremented or decremented easily, it is often used in loops. However, X generally has more combinations of addressing modes in which it is used as an index. Therefore, X is usually reserved for indexing, while A and Y between them process other parameters. When indirect addressing is used, this preference is reversed, since LDA (addr,X) is usually less useful than LDA (addr),Y.

```
       LDY  #$00     ;X HOLDS LENGTH
LOOP   DEX           ;DECREMENT IT
       BEQ  EXIT     ;EXIT WHEN 0
       LDA  (PTR),Y  ;LOAD ACCUMULATOR
       JSR  PRINT    ;PRINT SINGLE CHR
       CMP  #$0D     ;EXIT IF
       BEQ  EXIT     ; RETURN
       BNE  LOOP     ;CONTINUE LOOP
```

This admittedly unexciting example shows how A, X, and Y have distinct roles. The ROM routine to print the character is assumed to return the original X and Y values, as in fact it does.

LSR

Shift memory or accumulator right one bit.



Instruction	Addressing	Bytes	Cycles
$46 (70 %0100 0110)	LSR zero page	2	5
$4A (74 %0100 1010)	LSR accumulator	1	2
$4E (78 %0100 1110)	LSR absolute	3	6
$56 (86 %0101 0110)	LSR zero page,X	2	6
$5E (94 %0101 1110)	LSR absolute,X	3	7

Flags:

N	V	—	B	D	I	Z	C
0						x	x

Operation: Moves the contents of a memory location or the accumulator right by one bit position, putting 0 into bit 7 and the N (negative) flag and moving the rightmost bit, bit 0, into the carry flag. The Z flag is set to 1 if the result is 0, and cleared if not. Z can therefore become 1 only if the location held either $00 or $01 before LSR.

Uses:

1. Similar to ASL. This might well have been called arithmetic shift right. A byte is halved by this instruction (unless D is set), and its remainder is moved into the carry flag. With ASL, ROL, ROR, ADC, and SBC, this command is often used in ML calculations.
2. Other applications. LSR/ LSR/ LSR/ LSR moves a high nybble into a low nybble; LSR/ BCC tests bit 0, and branches if it was not set to 1. In addition, LSR turns off bit 7, giving an easy way to convert a negative number into its positive equivalent, when the sign byte is stored apart from the number's absolute value.

NOP

No operation.

Instruction	Addressing	Bytes	Cycles
$EA (234 %1110 1010)	NOP implied	1	2

Flags:

N	V	—	B	D	I	Z	C

Operation: Does nothing, except to increment the program counter and continue with the next opcode.

Uses:

1. Filling unused portions of program. This is useful with hand assembly and other methods where calculation of branch addresses cannot be done easily.
2. When writing machine code. A large block of NOPs (or an occasional sprinkling of them) can simplify the task of editing the code and inserting corrections. NOP can also be used as part of a timing loop.

ORA

Logical inclusive OR of memory with the accumulator A:= A OR M

Instruction	Addressing	Bytes	Cycles
$01 (1 %0000 0001)	ORA (zero page,X)	2	6
$05 (5 %0000 0101)	ORA zero page	2	3
$09 (9 %0000 1001)	ORA # immediate	2	2
$0D (13 %0000 1101)	ORA absolute	3	4
$11 (17 %0001 0001)	ORA (zero page),Y	2	5
$15 (21 %0001 0101)	ORA zero page,X	2	4
$19 (25 %0001 1001)	ORA absolute,Y	3	4*
$1D (29 %0001 1101)	ORA absolute,X	3	4*

*Add 1 if page boundary crossed.

Flags:

N	V	—	B	D	I	Z	C
x						x	

Operation: Performs the inclusive OR of the eight bits currently in the accumulator with the eight bits referenced by the opcode. The result is stored in A. If either bit is 1, the resulting bit is set to 1, so that, for example, %0011 0101 ORA %0000 1111 is %0011 1111. The negative flag N is set or cleared depending on bit 7 of the result. The Z (zero) flag is set if the result is zero, and clear otherwise.

Uses:

1. Setting a bit or bits. This is the opposite of masking out bits, as described under AND.

```
LDA  #ERROR
ORA  $90
STA  $90
```

   The example shows the method by which an error code of 1, 2, 4, or whatever, held in A, is flagged into the VIC's BASIC I/O status byte ST, stored in location $90, without losing the value currently in that location. For example, if ERROR is 4 and the current contents of ST is 64, then ORA $90 is equivalent to $04 OR $40, which gives $44. If ERROR is 0, then ORA $90 leaves the current value from location $90 unchanged. Note the necessity for STA $90; without it, only A holds the correct value of ST.

2. Other uses. These include the testing of several bytes for conditions which are intended to be true for each of them—for instance, that three consecutive bytes are all zero or that several bytes all have bit 7 equal to zero. LDY #00/ LDA (PTR),Y/ INY/ ORA (PTR),Y/ INY/ ORA (PTR),Y/ BNE ... branches if one or more bytes contains a nonzero value.

PHA

Push the accumulator's contents onto the stack. S:= A, SP:= SP−1

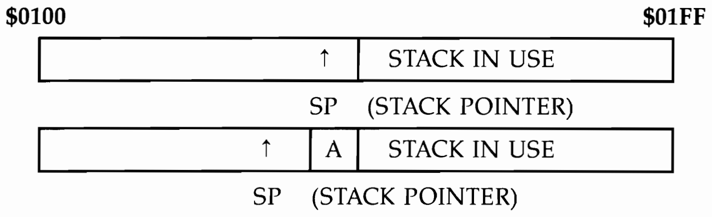
Instruction	Addressing	Bytes	Cycles
$48 (72 %0100 1000)	PHA implied	1	3

Flags:

N V — B D I Z C

Operation: The value in the accumulator is placed into the stack at the position currently pointed to by the stack pointer; the stack pointer is then decremented. Figure 10-1 illustrates the position before and after the push:

Figure 10-1. Effect of PHA



Uses: This instruction is used for temporary storage of bytes. It may be used to hold intermediate values of calculations produced during the parsing of numeric expressions, to temporarily store values for later recovery while A is used for other processing, for storage when swapping bytes, and for storage of A, X, and Y registers at the start of a subroutine.

The example shows a printout routine which is designed to end when the high bit of a letter in the table is 1. The output requires the high bit to be set to 0; but the original value is recoverable from the stack and may be used in a test for the terminator at the end of message.

```
LOOP JSR   GETC  ;GET NEXT CHR
     PHA         ;STORE ON STACK
     AND   #$7F  ;REMOVE BIT 7
     JSR   PRINT ;OUTPUT A CHARACTER
     PLA         ;RECOVER WITH BIT 7
     BPL   LOOP  ;CONTINUE IF BIT 7=0
```

PHP

Push the processor status register's contents onto the the stack. S:= PSR, SP:= SP−1

Instruction	Addressing	Bytes	Cycles
$08 (8 %0000 1000)	PHP implied	1	3

Flags:

N	V	—	B	D	I	Z	C

Operation: The operation is similar to PHA, except that the processor status register is put in the stack. The PSR is unchanged by the push.

Uses: Stores the entire set of flags, usually either to be recovered later and displayed by a monitor program or for recovery followed by a branch. PHP/ PLA leaves the stack in the condition it was found; it also loads A with the flag register, SR, so the flags' status can be stored for use later.

PLA

Pull the stack into the accumulator. SP:= SP+1, A:= S

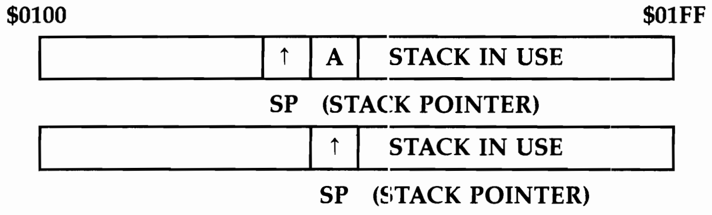
Instruction	Addressing	Bytes	Cycles
$68 (104 %0110 1000)	PLA implied	1	4

Flags:

N	V	—	B	D	I	Z	C
x						x	

Operation: The stack pointer is incremented, then the RAM address to which it points is read and loaded into A, setting the N and Z flags accordingly. The effect is similar to LDA. This diagram illustrates the position before and after the pull:

Figure 10-2. Effect of PLA



Uses:

1. PLA is the converse of PHA. It retrieves values put on the stack by PHA, in the reverse order. PLA/ PHA leaves the stack unchanged, but leaves A holding the contents of the current top of the stack. Flags N and Z are set as though by LDA.
2. To remove the top two bytes of the stack. This is a frequent use of PLA; it is equivalent to adding 2 to the stack pointer. This is done to "pop" a return address from the stack; in this way, the next RTS which is encountered will not return to the previous JSR, but to the one before it (assuming that the stack has not been added to since the JSR).

```
PLA  ;DISCARD ADDRESS STORED
PLA  ;BY JSR
RTS  ;RETURN TO EARLIER SUBROUTINE CALL
```

PLP

Pull the stack into the processor status register. SP:= SP+1, PSR:= S

Instruction	Addressing	Bytes	Cycles
$28 (40 %0010 1000)	PLP implied	1	4

Flags:

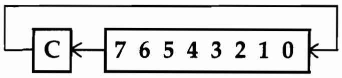
N	V	—	B	D	I	Z	C
x	x		x	x	x	x	x

Operation: The operation of PLP is similar to that of PLA, except that the processor status register, not the accumulator, is loaded from the stack.

Uses: Recovers previously stored flags with which to test or branch. See the notes on PHP. This can also be used to experiment with the flags, to set V, for example.

ROL

Rotate memory or accumulator and the carry flag left one bit.



Instruction	Addressing	Bytes	Cycles
$26 (38 %0010 0110)	ROL zero page	2	5
$2A (42 %0010 1010)	ROL accumulator	1	2
$2E (46 %0010 1110)	ROL absolute	3	6
$36 (54 %0011 0110)	ROL zero page,X	2	6
$3E (62 %0011 1110)	ROL absolute,X	3	7

Flags:

N	V	—	B	D	I	Z	C
x						x	x

Operation: Nine bits, consisting of the contents of the memory location referenced by the instruction or of the accumulator, and the carry bit, are rotated as the diagram shows. In the process, C is changed to what was bit 7; bit 0 takes on the previous value of C; and the negative flag becomes the previous bit 6. In addition, Z is set or cleared depending on the new memory contents.

Uses:

1. Doubles the contents of the byte that it references. In this way, ROL operates like ASL, but in addition the carry bit may be used to propagate the overflow from such a doubling. Multiplication and division routines take advantage of this property where a chain of consecutive bytes has to be moved one bit leftward. ROR is used where the direction of movement is rightward.

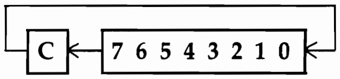
ASL $4000/ ROL $4001/ ROL $4002 moves the entire 24 bits of $4000–$4002 over by one bit, introducing 0 into the rightmost bit; if there is a carry, the carry flag will be 1.

2. Like ASL, ROL may be used before testing N, Z, or C, especially N.

```
ROL  A        ;ROTATE 1 BIT LEFTWARD
BMI  BRANCH   ;BRANCHES IF BIT 6 WAS ON
```

ROR

Rotate memory or accumulator and the carry flag right one bit.



Instruction	Addressing	Bytes	Cycles
$66 (102 %0110 0110)	ROR zero page	2	5
$6A (106 %0110 1010)	ROR accumulator	1	2
$6E (110 %0110 1110)	ROR absolute	3	6
$76 (118 %0111 0110)	ROR zero page,X	2	6
$7E (126 %0111 1110)	ROR absolute,X	3	7

Flags:

N	V	—	B	D	I	Z	C
x						x	x

Operation: Nine bits, consisting of the contents of memory referenced by the instruction and the carry bit, are rotated as the diagram shows. C becomes what was bit 0, bit 7 and the N flag take on the previous value of C, and Z is set or cleared depending on the byte's current contents. For applications, see ROL.

RTI

Return from interrupt. SP:= SP+1, PSR:= S, SP:= SP+1, PCL:= S, SP:= SP+1, PCH:= S

Instruction	Addressing	Bytes	Cycles
$40 (64 %0100 0000)	RTI implied	1	6

Flags:

N	V	—	B	D	I	Z	C
x	x		x	x	x	x	x

Operation: RTI takes three bytes from the stack, deposited there by the processor itself when the hardware triggered the interrupt. The processor status flags are recovered as they were when the interrupt occurred, and the program counter is restored so that the program resumes operation at the byte at which it was interrupted. Note that the contents of A, X, and Y are not saved or recovered automatically in this way, but must be saved by the interrupt processing and restored immediately before RTI. If you follow the vector stored in ROM at $FFFE/FFFF, you will see how this works.

Uses:

1. To resume after an interrupt. The techniques presented in Chapter 8 use the interrupt-processing routine in ROM, which is the simplest approach; it's not necessary even to understand RTI. The routines invariably end PLA/ TAY/ PLA/ TAX/ PLA/ RTI because the contents of A, X, and Y are pushed on the stack in A, X, Y order by CBM ROMs when interrupt processing begins.
2. To execute a jump. It is possible, as with RTS, to exploit the automatic nature of this command to execute a jump by pushing three bytes onto the stack, imitating an interrupt, then using RTI to pop the addresses and processor status. By simulating the stack contents left by an interrupt, the following routine jumps to 256*HI + LO with its processor flags equal to whatever was pushed on the stack as PSR.

```
LDA  HI
PHA
LDA  LO
PHA
LDA  PSR
PHA
RTI
```

RTS

Return from subroutine. SP:= SP+1, PCL:= S, SP:= SP+1, PCH:= S, PC:= PC+1

Instruction	Addressing	Bytes	Cycles
$60 (96 %0110 0000)	RTS implied	1	6

Flags:

N V — B D I Z C

Operation: RTS takes two bytes from the stack, increments the result, and jumps to the address found by putting the calculated value into the program counter. It is similar to RTI but does not change the processor flags, since an important feature of subroutines is that, on return, flags should be usable. Also, unlike RTI in which the address saved is the address to return to, RTS must increment the address it fetches from the stack, which points to the second byte after a JSR.

Uses:

1. Return after a subroutine. This is straightforward; a batch of ML to be callable by JSR is simply ended or exited from with RTS. This also applies to ML routines

callable from BASIC with SYS calls; in this case the return address to the loop which executes BASIC is put on the stack first by the system.

2. As a form of jump. RTS is used as a form of jump which takes up no RAM space and can be loaded from a table. For example, the following routine jumps to the address $HILO+1, so put the desired address−1 on the stack.

```
LDA  #$HI
PHA
LDA  #$LO
PHA
RTS
```

Notes: See PLA for the technique of discarding (popping) return addresses. JSR SUB/ RTS is identical in effect to JMP SUB, since SUB must end with an RTS. This point can puzzle programmers.

SBC

Subtract memory with borrow from accumulator. A:= A−M−(1−C)

Instruction	Addressing	Bytes	Cycles
$E1 (225 %1110 0001)	SBC (zero page,X)	2	6
$E5 (229 %1110 0101)	SBC zero page	2	3
$E9 (233 %1110 1001)	SBC # immediate	2	2
$ED (237 %1110 1101)	SBC absolute	3	4
$F1 (241 %1111 0001)	SBC (zero page),Y	2	5*
$F5 (245 %1111 0101)	SBC zero page,X	2	4
$F9 (249 %1111 1001)	SBC absolute,Y	3	4*
$FD (253 %1111 1101)	SBC absolute,X	3	4*

*Add 1 if page boundary crossed.

Flags:

N	V	—	B	D	I	Z	C
x	x					x	x

Operation: It is usual to set the carry bit before this operation, or to precede it by an operation which is known to leave the carry bit set. Then SBC appears to subtract from the accumulator the data referenced by the addressing mode. If the carry flag is still set, this indicates that the result did not borrow (that is, that the accumulator's value is greater than or equal to the data). When C is clear, the data exceeded the accumulator's contents; C shows that a borrow is needed. Within the chip, A is added to the twos complement of the data and to the complement of C; this conditions the N, V, Z, and C flags.

Uses:

1. Single-byte subtraction. The following example is a detail from PRINT. When processing the comma in a PRINT statement, the cursor is moved to position 0, 10, 20, etc. Suppose the cursor is at 17 horizontally; subtract 10's until the carry flag is clear, when A will hold −3. The twos complement is 3, so three spaces or

cursor-rights take you to the correct position on the screen. Note that ADC #$01 adds 1 only; the carry flag is known to be 0 by that stage.

```
        LDA  HORIZ ;LOAD CURRENT CURSOR POSN
        SEC        ;CARRY FLAG SET DURING LOOP
LOOP    SBC  #$0A  ;SUBTRACT 10 UNTIL CARRY...
        BCS  LOOP  ;...IS CLEAR (A IS NEG)
        EOR  #$FF  ;FLIP BITS AND ADD 1 TO
        ADC  #$01  ;CONVERT TO POSITIVE.
```

2. Double-byte subtraction. The point about subtracting one 16-bit number from another is that the borrow is performed automatically by SBC. The C flag is first set to 1; then the low byte is subtracted; then the high byte is subtracted, with borrow if the low bytes make this necessary.

   In the following example $026A is subtracted from the contents of addresses (or data) LO and HI. The result is replaced in LO and HI. Note that SEC is performed only once. In this way, borrowing is performed properly. For example, suppose the address from which $026A is to be subtracted holds $1234. When $6A is subtracted from $34, the carry flag is cleared, so that $02 and 1 is subtracted from the high byte $12.

```
SEC
LDA  LO
SBC  #$6A
STA  LO
LDA  HI
SBC  #$02
STA  HI
```

SEC

Set the carry flag to 1. C:= 1

Instruction	Addressing	Bytes	Cycles
$38 (56 %0011 1000)	SEC implied	1	2

Flags:

N	V	—	B	D	I	Z	C
							1

Operation: Sets the carry flag. This is the opposite of CLC, which clears it.

Uses: Used whenever the carry flag has to be put into a known state; usually SEC is performed before subtraction (SBC) and CLC before addition (ADC) since the numeric values used are the same as in ordinary arithmetic. Some Kernal routines require C to be cleared or set, giving different effects accordingly. SEC/BCS is sometimes used as a "branch always" command.

SED

Set the decimal mode flag to 1. D:= 1

Instruction	Addressing	Bytes	Cycles
$F8 (248 %1111 1000)	SED implied	1	2

Flags:

N	V	—	B	D	I	Z	C
				1			

Operation: Sets the decimal flag. This is the opposite of CLD, which clears it.

Uses: Sets the mode to BCD (binary-coded decimal) arithmetic, in which each nybble holds a decimal numeral. For example, ten is held as 10 and ninety as 90. Two thousand four hundred fifteen is 2415 in two bytes. ADC and SBC are designed to operate in this mode as well as in binary, but the flags no longer have the same meaning, except C. The result is not much different from arithmetic using individual bytes for each digit 0–9, but it takes up only half the space and is faster.

SEI

Set the interrupt disable flag to 1. I:= 1

Instruction	Addressing	Bytes	Cycles
$78 (120 %0111 1000)	SEI implied	1	2

Flags:

N	V	—	B	D	I	Z	C
					1		

Operation: Sets the interrupt disable flag. This is the opposite of CLI, which clears it.

Uses: When this flag has been set, no interrupts are processed by the chip, except non-maskable interrupts (which have higher priority), BRK, and RESET. IRQ interrupts are processed by a routine vectored through locations $FFFE/FFFF, like BRK. If the vector in the very top locations of ROM is followed, the interrupt servicing routines can be found. In the VIC, these are not all in ROM: The vectors use an address in RAM before jumping back to ROM.

The example here is a typical initialization routine to redirect the VIC's RAM IRQ vector into the user's own program at $0345 (where it may set a musical tone or whatever). See Chapter 8 for other examples.

```
033A   SEI
033B   LDA  #$45
033D   STA  $0314
033F   LDA  #$03
0341   STA  $0315
0343   CLI
0344   RTS
```

STA

Store the contents of the accumulator into memory. M:= A

Instruction	Addressing	Bytes	Cycles
$81 (129 %1000 0001)	STA (zero page,X)	2	6
$85 (133 %1000 0101)	STA zero page	2	3
$8D (141 %1000 1101)	STA absolute	3	4
$91 (145 %1001 0001)	STA (zero page),Y	2	6
$95 (149 %1001 0101)	STA zero page,X	2	4
$99 (153 %1001 1001)	STA absolute,Y	3	5
$9D (157 %1001 1101)	STA absolute,X	3	5

Flags:

N	V	—	B	D	I	Z	C

Operation: The value in A is sent to the address referenced by the opcode. All registers and flags are unchanged.

Uses:

1. Intermediate storage. Transfer of blocks of data from one part of memory to another needs a temporary intermediate store, usually in A, which is alternately loaded and stored. See LDA.
2. Saving results of binary operations. Binary operations using the accumulator, notably ADC and SBC, are performed within the accumulator; a common bug in machine language programs is forgetting to save the result.

```
LDA  $90   ; ST BYTE
AND  #$FD  ; BIT 1 OFF
STA  $90   ; REMEMBER THIS!
```

3. Setting the contents of certain locations to known values.

```
LDA  #$89
STA  $22   ; SETS VECTOR AT $22/23
LDA  #$C3
STA  $23   ; TO $C389
```

STX

Store the contents of the X register into memory. M:= X

Instruction	Addressing	Bytes	Cycles
$86 (134 %1000 0110)	STX zero page	2	3
$8E (142 %1000 1110)	STX absolute	3	4
$96 (150 %1001 0110)	STX zero page,Y	2	4

Flags:

N	V	—	B	D	I	Z	C

Operation: The byte in the X register is sent to the address referenced by the opcode. All registers and flags are unchanged.

Uses: The uses are identical to those of STA. There is a tendency for X to be used as an index, so STX is less used than STA.

STY

Store the contents of the Y register into memory. M:= Y

Instruction	Addressing	Bytes	Cycles
$84 (132 %1000 0100)	STY zero page	2	3
$8C (140 %1000 1100)	STY absolute	3	4
$94 (148 %1001 0100)	STY zero page,X	2	4

Flags:

N	V	—	B	D	I	Z	C

Operation: The byte in the Y register is sent to the address referenced by the opcode. All registers and flags are unchanged.

Uses: STY resembles STX; the comments under STX apply.

TAX

Transfer the contents of the accumulator into the X register. X:= A

Instruction	Addressing	Bytes	Cycles
$AA (170 %1010 1010)	TAX implied	1	2

Flags:

N	V	—	B	D	I	Z	C
x						x	

Operation: The byte in A is transferred to X. The N and Z flags are set as though LDX had taken place.

Uses: This transfer is generally used to set X for use as an index or a parameter, or to temporarily hold A. The example is from a high-resolution screen-plotting routine; it plots a black dot in a location with a coded value of 1, 2, 4, or 8 in $FB. X on entry holds the position of the current X in a table. On exit X holds the position of the new character. Intermediate calculations use the accumulator because there is no "EOR with X" instruction.

```
TXA
EOR  #$FF
ORA  $FB
EOR  #$FF
TAX
LDA  TABLE,X
```

Note that registers A, X, Y, and the stack pointer are interchangeable with one instruction in some cases but not in others. The connections are shown below:

Y = A = X = S.

TAY

Transfer the contents of the accumulator into the Y register. Y:= A

Instruction	Addressing	Bytes	Cycles
$A8 (168 %1010 1000)	TAY implied	1	2

Flags:

N	V	—	B	D	I	Z	C
x						x	

Operation: The byte in A is transferred to Y. The N and Z flags are set as though LDY had taken place.

Uses: See TAX.

TSX

Transfer the stack pointer into the X register. X:= SP

Instruction	Addressing	Bytes	Cycles
$BA (186 %1011 1010)	TSX implied	1	2

Flags:

N	V	—	B	D	I	Z	C
x						x	

Operation: The stack pointer is transferred to X. Note that the stack pointer is always offset onto $0100, so when the stack is accessed, the high byte of its memory location is $01. The pointer itself is a single byte.

Uses:

1. To look at current values on the stack. TSX/ LDA $0100,X loads A with the contents presently at the top of the stack; LDA $0101,X loads the last item pushed on the stack (one byte higher) into A, and so on. BASIC tests for BRK or interrupt with PHA/ TXA/ PHA/ TYA/ PHA/ TSX/ LDA $0104,X/ AND #$10 because the return-from-interrupt address and the SR are pushed by the interrupt before the system saves its own three bytes. LDA $0104,X loads the flags saved when the interrupt or BRK happened.
2. To determine space left on the stack. BASIC does this and signals ?OUT OF MEMORY ERROR if there are too many GOSUBs, FOR-NEXT loops, or complex calculations with intermediate results.
3. Processing. Sometimes the stack pointer is stored and a lower part of the stack temporarily used for processing.

TXA

Transfer the contents of the X register into the accumulator. A:= X

Instruction	Addressing	Bytes	Cycles
$8A (138 %1000 1010)	TXA implied	1	2

Flags:

N	V	—	B	D	I	Z	C
x						x	

Operation: The byte in X is transferred to A. The N flag and Z flag are set as though LDA had taken place.

Uses: See TAX.

TXS

Transfer the X register into the stack pointer. SP:= X

Instruction	Addressing	Bytes	Cycles
$9A (154 %1001 1010)	TXS implied	1	2

Flags:

N	V	—	B	D	I	Z	C

Operation: X is stored in the stack pointer. PHA or PHP will place a byte onto the stack at $0100 plus the new stack pointer, and PLA or PLP will pull from the next byte up from this. In addition, RTI and RTS will return to addresses determined by the stack contents at the new position of the stack.

Uses:

1. As part of the RESET sequence. TXS is always part of the RESET sequence; otherwise, the stack pointer could take any value. CBM computers use the top bytes of the stack for BASIC addresses. When the VIC is turned on, LDX #$FF/ TXS sets the pointer to the top of the stack, but if BASIC is to run (that is, if no autorun cartridge is in place), SP is moved to leave locations $01FA–$01FF ready for use by the RUN command.

   SP has high values to start with because it is decremented as data is pushed on the stack. If too much data is pushed, perhaps by an improperly controlled loop, SP decrements right through $00 to $FF again, crashing its program.
2. Switching to a new stack location. This is a rarely seen use of TXS. As a simple example, the following routine is an equivalent to PLA/ PLA which you have seen (under RTS) to be a "pop" command which deletes a subroutine's return address. Incrementing the stack pointer by 2 has the identical effect.

```
CLC
TSX
TXA
ADC  #$02
TAX
TXS
```

TYA

Transfer the contents of the Y register into the accumulator. A:= Y

Instruction	Addressing	Bytes	Cycles
$98 (152 %1001 1000)	TYA implied	1	2

Flags:

N	V	—	B	D	I	Z	C
x						x	

Operation: The byte in Y is transferred to A. The N flag and Z flag are set as though LDA had taken place.

Uses: See TAX. The transfers TAX, TAY, TXA, and TYA all perform similar functions.

Chapter 11

VIC-20 ROM Guide

Chapter 11

VIC-20 ROM Guide

VIC-20 Memory Map

This chapter maps in detail the first few hundred RAM locations, the BASIC ROM, and the Kernal ROM. It will be especially valuable to programmers who want to make full use of VIC-20 BASIC.

Locations are listed for both the VIC and the Commodore 64, since many locations are the same on the two computers.

BASIC is stored in ROM from $C000 to $DFFF. The computer's *operating system*, the ML that controls input/output and related operations, is stored in ROM from $E000 to $FFFF, called the Kernal ROM. It contains large numbers of routines, but generally Kernal routines are taken to be only those which are called through the Kernal jump table.

Commodore recommends that ML programmers use only Kernal routines. That, however, rules out most of BASIC. Moreover, transportability between machines is likely to be very difficult even with the Kernal. Generally, you should use any of these routines where they are likely to make better programs.

There is a potential problem between machines of the *same* type. For example, several 64 ROM versions exist, with Kernal ROM variations. In practice this is rarely a problem. But if you want to be certain, relocate your routines into RAM as far as possible.

A number of routines are vectored through RAM; Chapter 8 explains how to take advantage of this.

Free RAM, available for programming, is listed in Chapter 6.

Notation

Labels have been included as reference points.

BASIC number-handling is a bit complex. FAC1 and FAC2 refer to Floating Point Accumulators 1 and 2. They hold two numbers during addition, multiplication, etc., which is done in a six-byte format (EMMMMS, consisting of exponent/mantissa or data/sign), called FLPT for short. MFLPT refers to the way numbers are stored in memory after BASIC, in a five-byte format with one bit of data less than FLPT. MFLPT format is explained in Chapter 6. BASIC of course has routines to interconvert these. INT or FIX format is the simpler format with bytes in sequence.

A, X, and Y are the 6502's registers. A/X means the two-byte value with A holding the low byte and X the high byte. (A/Y) is a two-byte pointer held in A and Y, with A holding the low byte and Y the high byte. String descriptors are three bytes of data, the first holding the string's length, the second and third the low and high bytes of the pointer to the start of the string.

The following listings consist of three columns. The first column gives the label. The second column lists the VIC and 64 addresses; where one address is given it applies to both computers, but where two are given the VIC address comes first. Finally, a description of the use of the location, or of the routine that begins at the specified address, is given.

Page 0: RAM $0000–$00FF

Label	VIC/64	Descriptions
	$000/$310	USR function JMP instruction ($4C).
	$001–$002/ $311–$312	USR function address (low byte/high byte). Initialized to point to routine to print ILLEGAL QUANTITY ERROR ($D248).
FACINT	$003–$004	Vector to routine to convert FAC to integer in A/Y (usually $D1AA).
INTFAC	$005–$006	Vector to routine to convert integer in A/Y to floating point in FAC (usually $D391).
CHARAC	$007	Delimiting character used when scanning. Also temporary integer (0–255) used during INT.
INTEGR	$007–$008	Intermediate integer used during OR/AND.
ENDCHR	$008	Delimiter used when scanning strings.
TRMPOS	$009	Temporary location used for calculating TAB and SPC column.
VERCHK	$00A	Flag to indicate LOAD (0) or VERIFY (1).
COUNT	$00B	Temporary pointer used with BASIC input buffer.
DIMFLG	$00C	Flag: default array dimension.
VALTYP	$00D	Current variable data type flag; 0 means numeric, $FF means string.
INTFLG	$00E	Current variable data type flag; 0 means floating point; $80 means integer.
GARBLF	$00F	Flag: used in garbage collection, LIST, DATA, error messages.
SUBFLG	$010	Flag to indicate integers or array elements (flag=$80), used to forbid them in FOR loops and function definitions.
INPFLG	$011	Flag used by READ routine; $00 means INPUT, $40 means GET, $98 means READ.
TANSGN	$012	Sign byte used by TAN, SIN. Also set according to any comparison being performed: > sets $01, = sets $02, < sets $04.
CHANNL	$013	Current I/0 device number; prompts suppressed if not 0.
LINNUM	$014–$015	Line number integer (0–63999) or standard two-byte integer used by GOTO, GOSUB, POKE, PEEK, WAIT, and SYS.
TEMPPT	$016	Index to next entry on string descriptor stack (may be $19, $1C, $1F, or $22).
LASTPT	$017–$018	Pointer to current entry on string descriptor stack.
TEMPST	$019–$021	Stack for three temporary string descriptors.
INDEX1	$022–$023	General-purpose pointer, for example, for memory moves.
INDEX2	$024–$025	General-purpose pointer, for example, for number movements.
RESHO	$026–$02A	Floating point workspace used by multiply and divide.
TXTTAB	$02B–$02C	Pointer to first byte of BASIC program (usually 4097 for the unexpanded VIC, 1025 for VIC with 3K, or 4609 for VIC with 8K or more).
VARTAB	$02D–$02E	Pointer to start of program variables; first byte beyond end of
ARYTAB	$02F–$030	Pointer to start of arrays; first byte beyond end of variables.
STREND	$031–$032	Pointer to start of free RAM available for strings; first byte beyond end of arrays.
FRETOP	$033–$034	Pointer to current lower boundary of string area. (Set to MEMSIZ on CLR, RUN.)
FRESPC	$035–$036	Utility pointer used when new string is being added to string area.
MEMSIZ	$037–$038	Pointer to one byte beyond the top of RAM available to BASIC.

CURLIN	**$039–$03A**	BASIC line number being interpreted ($FF in $03A indicates immediate mode).
OLDLIN	**$03B–$03C**	If STOP/END/BREAK occurs, this holds the previous BASIC line number for CONT.
OLDTXT	**$03D–$03E**	Pointer to beginning of current BASIC line for CONT.
DATLIN	**$03F–$040**	Line number of current DATA statement. Initialized to $0000 on RUN.
DATPTR	**$041–$042**	Pointer to one byte beyond the DATA item read by the last READ statement. Initialized to BASIC start on RUN.
INPTR	**$043–$044**	Temporary storage of DATPTR during READ statement; also pointer within input buffer during INPUT (points to last character entered).
VARNAM	**$045–$046**	Current BASIC variable; two-character name with most significant bit (bit 7) of each byte used to indicate variable type: floating point = bit 7 clear in both bytes, integer = bit 7 set in both, string = bit 7 set in $46, function = bit 7 set in $45.
VARPNT	**$047–$048**	Pointer to current variable's address in RAM. Points one byte beyond variable name.
FORPNT	**$049–$04A**	Temporary pointer to variables in memory for INPUT, assignments, etc., and for loop variable in FOR loops. Also holds the two parameters for WAIT statements .
OPPTR	**$04B**	Pointer within operator table during expression evaluation in routine FRMEVL.
OPMASK	**$04D**	Comparison mask used in FRMEVL: > sets $02, = sets $04, < sets $08.
DEFPNT	**$04E–$04F**	Pointer to variable in function definition, within variable table in RAM. Also used by garbage collection routine GARBAG.
TEMPF3	**$04E–$052**	Temporary storage for a MFLPT item.
DSCPNT	**$050–$051**	Pointer to descriptor in variable list or to string in dynamic string area; used during string operations.
SIZE	**$052**	Length of the current BASIC string.
FOUR6	**$053**	Length of string variable during garbage collection.
JMPER	**$054–$056**	Jump vector for function evaluations, JMP ($4C) followed by function address from function vector table.
TEMPF1	**$057–$05B**	Temporary pointers (for example, in memory move); temporary floating point accumulator.
HIGHDS	**$058–$059**	Pointer used by block transfer routine BLTU.
ARYPNT	**$058–$059**	Pointer used when initializing arrays (when DIM encountered).
HIGHTR	**$05A–$05B**	Pointer used by block transfer routine BLTU.
TEMPF2	**$05C–$060**	Temporary floating point accumulator.
DECCNT	**$05D**	Number of digits after/before decimal point in ASCII-to-FLPT and FLPT-to-ASCII conversion for FIN/FOUT.
TENEXP	**$05E**	Exponent used in ASCII-to-FLPT and FLPT-to-ASCII conversion in FIN/FOUT.
DPTFLG	**$05F**	Flag used by FIN when inputting numbers; $80 if string contains decimal point.
LINPTR	**$05F–$060**	Pointers used when searching for line numbers, searching for variables in variable list, doing block transfers.
EXPSGN	**$060**	Sign of exponent of number being input by FIN; $80 signifies negative.

FAC1	**$061-$066**	Floating Point Accumulator #1. Consists of exponent byte, four mantissa bytes, and a sign byte. (The results of most arithmetic operations are placed here.) Integer results in two bytes FAC1+3, FAC1+4.
SGNFLG	**$067**	Flag used by FIN when inputting numbers; $FF if number is negative. Also stores count of terms in polynomial series when evaluating trig functions.
BITS	**$068**	Bit overflow area on normalizing FAC1.
FAC2	**$069-$06E**	Floating Point Accumulator #2; used with FAC1 in evaluation of products, sums, differences, etc.
ARISGN	**$06F**	Sign comparison between FAC1 and FAC2; $00 means same sign, $FF means opposite.
FACOV	**$070**	Rounding/overflow byte for FAC1.
TEMPTX	**$071-$072**	General pointer used in CRUNCH, VAL, series evaluation, with tape buffer, etc.
CHRGET	**$073-$08A**	Fetch next BASIC character into A (spaces are skipped) and set flags; C cleared if ASCII numeral 0–9; Z set if end-of-line or colon (:).
CHRGOT	**$079**	Entry point within CHRGET to re-get current BASIC character and set flags as CHRGET does. Does not increment TXTPTR first.
TXTPTR	**$07A-$07B**	Pointer into BASIC text used by CHRGET and CHRGOT routines.
RNDX	**$08B-$08F**	Floating point random number seed and subsequent pseudorandom values.
STATUS	**$090**	Status ST for serial devices and cassette.
STOPFL	**$091**	Flag: contains $FE (254) if STOP key pressed.
TSERVO	**$092**	Tape timing constant.
VERCK	**$093**	Flag to indicate LOAD (0) or VERIFY (1).
ICHRFL	**$094**	Serial flag: $FF indicates a character is awaiting output.
IDATO	**$095**	Serial character to be output; $FF indicates no character.
TEOB	**$096**	Flag: end of data block from tape.
TEMPXY	**$097**	Temporary X,Y storage during cassette read/RS-232 get.
NFILES	**$098**	Number of files open (maximum of 10); index to file table.
DFLTI	**$099**	Current input device number; default = 0 (keyboard).
DFLTO	**$09A**	Current output device number; default = 3 (screen).
TPARIT	**$09B**	Parity of byte written to tape.
TBYTFL	**$09C**	Flag: byte read from tape is complete.
MSGFLG	**$09D**	Flag: $00 means program mode, $80 means direct mode.
HDRTYP	**$09E**	Tape buffer header ID.
PTR1	**$09E**	Cassette pass 1 read errors.
PTR2	**$09F**	Cassette pass 2 read errors.
TIME	**$0A0-$0A2**	Three-byte jiffy clock for TI, updated 60 times per second. Bytes arranged in order of decreasing significance.
TSFCNT	**$0A3**	Tape read/write bit counter.
TBTCNT	**$0A4**	Tape read/write pulse counter.
CNTDN	**$0A5**	Tape synchronization write countdown.
BUFPNT	**$0A6**	Count of bytes in tape I/0 buffer.
INBIT	**$0A7**	RS-232 temporary storage for received bits.
PASNUM	**$0A7**	General temporary store for cassette read/write.
BITCI	**$0A8**	RS-232 received bit count. Also temporary store for cassette read/write.
RINONE	**$0A9**	RS-232 receive: check for start bit.
TBITER	**$0A9**	Write start bit/read bit sequence error.

RIDATA	**$0AA**	Tape read mode; 0 = scan, 1–15 = count, $40 = LOAD, $80 = end-of-tape marker.
RIDATA	**$0AA**	RS-232 received byte buffer.
TCKS	**$0AB**	Counter of seconds before tape write. Also checksum.
RPRTY	**$0AB**	RS-232 received byte parity.
SAL	**$0AC–$0AD**	Start address for LOAD/SAVE. Pointer also used by scroll and INSert.
EAL	**$0AE–$0AF**	End address for LOAD/SAVE. Also used as pointer to color RAM used by INSert.
CMPO	**$0B0,$0B1**	Timing constants for tape.
TAPE1	**$0B2–$0B3**	Pointer to start of cassette buffer, usually $033C.
BITTS	**$0B4**	RS-232 transmit bit count.
TTIX	**$0B4**	Tape read timer flag.
NXTBIT	**$0B5**	RS-232 transmit: next bit to send.
TEOT	**$0B5**	End of tape read.
RODATA	**$0B6**	RS-232 transmit: byte to be sent.
TERRR	**$0B6**	Tape read error flag.
FNLEN	**$0B7**	Number of characters in filename; 0 = no name.
LA	**$0B8**	Current logical file number.
SA	**$0B9**	Current secondary address as used.
FA	**$0BA**	Current device number; for example, 3 = screen, 4 = printer.
FINADR	**$0BB–$0BC**	Pointer to start of current filename.
ROPRTY	**$0BD**	RS-232 output parity.
TCHR	**$0BD**	Byte to be written to/read from tape.
FSBLK	**$0BE**	Number of blocks remaining to read/write.
MYCH	**$0BF**	Serial word buffer where byte is assembled.
CAS1	**$0C0**	Cassette motor control flag.
STAL	**$0C1–$0C2**	Start address for LOAD and cassette write.
MEMUSS	**$0C3–$0C4**	Pointer for general use, for example, calculating LOAD address.
LSTX	**$0C5**	Matrix value of key pressed during last keyboard scan; $40 = no key pressed.
NDX	**$0C6**	Number of characters in keyboard buffer.
RVS	**$0C7**	Flag: print reverse characters; 0 = normal, $12 = reverse.
INDX	**$0C8**	Count of characters in line input from screen.
LXSP	**$0C9**	Cursor Y value (row) at start of INPUT.
LYSP	**$0CA**	Cursor X value (column) at start of INPUT.
KEYVAL	**$0CB**	Copy of keypress LSTX checked by interrupt so that a held key registers only once.
BLNCT	**$0CD**	Countdown to next cursor toggle (from $14).
GDBLN	**$0CE**	Character (screen code) at cursor position (never reverse of character).
BLNON	**$0CF**	Flag: 1 = cursor in blink phase, 0 = not in blink phase.
CRSW	**$0D0**	Flag: 3 = input from screen, 0 = input from keyboard.
PNT	**$0D1–$0D2**	Address of start of current line on-screen.
PNTR	**$0D3**	Cursor position (X value) along current logical line (0–$57).
QTSW	**$0D4**	Quotes mode flag: flips each time quotes are encountered; 0 = move cursor, etc.; 1 = print reverse characters.
LNMX	**$0D5**	Length of current logical screen line (21, 43, 65, or 87).
TBLX	**$0D6**	Row of cursor.
TMPD7	**$0D7**	CHR$ value of last character input/output to screen; tape temporary I/0 storage and checksum.

INSRT	**$0D8**	Number of keyboard inserts outstanding.
LDTB1	**$0D9–$0F0**	Table of 23 high bytes of pointers to the start of screen lines in RAM. (The low bytes are held in ROM from $EDFD.) Lines with wraparound have bit 7 = 0; otherwise, bit 7 = 1.
USER	**$0F3–$0F4**	Pointer to byte in color RAM corresponding to beginning of current line onscreen.
KEYTAB	**$0F5–$0F6**	Address of current keyboard decoding table.
RIBUF	**$0F7–$0F8**	RS-232: pointer to start of receive buffer.
ROBUF	**$0F9–$0FA**	RS-232: pointer to start of transmit buffer.
	$0FB–$0FE	Unused; available for user programs.
BASZPT	**$0FF**	Temporary storage area for FLPT-to-ASCII conversion.

Page 1 (Stack Area): RAM $0100–$01FF

Label	VIC/64	Descriptions
ASCWRK	**$0FF–$10A**	Area for conversion of numerals into ASCII string format for printing.
BAD	**$100–$13E**	Table of tape read errors.
STACK	**$140–$1FF**	BASIC stack area.

Page 2: RAM $0200–02FF

Label	VIC/64	Descriptions
BUF	**$200–$258**	System input buffer; all keyboard input is read into here.
LAT	**$259–$262**	Table of up to 10 active logical file numbers.
FAT	**$263–$26C**	Table of up to 10 corresponding device numbers.
SAT	**$26D–$276**	Table of 10 corresponding secondary addresses as used by system.
KEYD	**$277–$280**	Keyboard buffer: maximum of 10 characters are read from keyboard and placed here by the interrupt routine.
LORAM	**$281–$282**	Pointer to lowest available BASIC RAM byte (initialized on power-up; value varies with RAM expansion).
HIRAM	**$283–$284**	Pointer to highest available BASIC RAM byte (initialized on power-up).
TIMOUT	**$285**	Serial timeout flag.
COLOR	**$286**	Current color code: POKEd into color RAM when printing characters to screen.
GDCOL	**$287**	Color of character under cursor.
HIBASE	**$288**	High byte of screen memory address.
XMAX	**$289**	Maximum number of characters storable in keyboard buffer (initialized to 10).
RPTFLG	**$28A**	Flag controlling key repeats; $00 = repeat cursor move and space keys; $80 = repeat all keys; $40 = repeat no keys. $00 is default.
KOUNT	**$28B**	Delay before repeat operates (system resets this).
DELAY	**$28C**	Delay between repeats.
SHFLAG	**$28D**	Detect SHIFT, Commodore, CTRL keypress: $01 = SHIFT, $02 = Commodore, $04 = CTRL. These are additive: $05 = SHIFT and CTRL keys together, etc.
LSTSHF	**$28F**	Last SHIFT key pattern; used for debouncing.
KEYLOG	**$28F–$290**	Vector to routine to check SHIFT pattern; used by SCNKEY Kernal routine.
MODE	**$291**	Flag: $00 = enable upper/lowercase toggle using SHIFT and Commodore; $80 = disable.

AUTODN	**$292**	Flag: autoscroll down during input; 00 = disable.
M51CTR	**$293**	RS-232: control register.
M51CDR	**$294**	RS-232: command register.
M51AJB	**$295-$296**	RS-232: nonstandard transmission rate value—not used.
RSSTAT	**$297**	RS-232: status register ST.
BITNUM	**$298**	RS-232: number of bits to send/receive.
BAUDOF	**$299-$29A**	RS-232: baud rate timing constant.
RIDBE	**$29B**	RS-232: input buffer pointer; points to latest character input (end of buffer).
RIDBS	**$29C**	RS-232: input buffer pointer; points to first available character (start of buffer).
RODBS	**$29D**	RS-232: output buffer pointer: start of buffer.
RODBE	**$29E**	RS-232: output buffer pointer: end of buffer.
IRQTMP	**$29F-$2A0**	Temporary store for IRQ vector during tape operations.
	$2A1-$2FF	Free RAM available to user. (The Commodore 64 uses $02A1–$02A6.) The *Super Expander* cartridge and other utilities use part of this area.

Page 3: RAM $0300–$03FF

Label	VIC/64	Descriptions
IERROR	**$300-$301**	Vector to BASIC print error message (normally $C43A); X register holds error message number.
IMAIN	**$302-$303**	Vector to routine to input or execute line of BASIC (normally $C483).
ICRNCH	**$304-$305**	Vector to BASIC tokenizing routine (normally $C57C).
IQPLOP	**$306-$307**	Vector to BASIC LIST routine (normally $C71A).
IGONE	**$308-$309**	Vector to BASIC RUN routine (normally $C7E4).
IEVAL	**$30A-$30B**	Vector to BASIC single-expression evaluation routine (normally $CE86).
SAREG	**$30C**	6502 Accumulator storage for SYS; A is loaded from this location on SYS call and stored back into it when SYS call ends.
SXREG	**$30D**	6502 X register storage for SYS; handling as above.
SYREG	**$30E**	6502 Y register storage for SYS; handling as above.
SPREG	**$30F**	6502 Status register storage for SYS; handling as above.

(Note: The vectors from $314 to $333 are reinitialized each time STOP-RESTORE is pressed, assuming NMINV holds its normal value.)

CINV	**$314-$315**	Vector for IRQ interrupt (normally $EABF). Called from $FF82.
CBINV	**$316-$317**	Vector for BRK (normally $FED2). Called from $FF7F.
NMINV	**$318-$319**	Vector for NMI (normally $FEAD).
IOPEN	**$31A-$31B**	Vector to Kernal OPEN routine (normally $F40A). Called from $FFC0.
ICLOSE	**$31C-$31D**	Vector to Kernal CLOSE routine (normally $F34A). Called from $FFC3.
ICHKIN	**$31E-$31F**	Vector to Kernal CHKIN routine (normally $F2C7). Called from $FFC6.
ICKOUT	**$320-$321**	Vector to Kernal CHKOUT routine (normally $F309). Called from $FFC9.
ICLRCH	**$322-$323**	Vector to Kernal CLRCHN routine (normally $F3F3). Called from $FFCC.

IBASIN	**$324-$325**	Vector to Kernal CHRIN routine (normally $F20E). Called from $FFCF.
IBSOUT	**$326-$327**	Vector to Kernal CHROUT routine (normally $F27A). Called from $FFD2.
ISTOP	**$328-$329**	Vector to Kernal STOP routine (normally $F770). Called from $FFE1.
IGETIN	**$32A-$32B**	Vector to Kernal GETIN routine (normally $F1F5). Called from $FFE4.
ICLALL	**$32C-$32D**	Vector to Kernal CLALL routine (normally $F3EF). Called from $FFE7.
USRCMD	**$32E-$32F**	Unused vector: may be defined by user; initialized to BRK vector ($FED2).
ILOAD	**$330-$331**	Vector to Kernal LOAD routine (normally $F549).
ISAVE	**$332-$333**	Vector to Kernal SAVE routine (normally $F685).
	$334-$33B	Eight unused bytes.
TBUFFR	**$33C-$3FB**	Tape I/0 buffer (192 bytes long). Can be used for ML programs but tape use will overwrite.
	$3FC-$3FF	Four unused bytes.

Addresses from $8000 to $BFFF

The block of memory from $8000 to $97FF is used by the VIC-20 as follows (all else up to $BFFF is available for plug-in RAM or ROM):

$8000-$8FFF	Character generator ROM (see Chapter 12)
$9000-$900F	VIC chip (see Chapter 5 for registers; Chapter 12 for programs)
$9110-$911F	VIA#1 (see Chapter 5 for description and programs)
$9120-$912F	VIA#2 (see Chapter 5 for description and programs)
$9400-$97FF	Color RAM (see Chapter 12)

BASIC and Kernal ROM

VIC-20 and Commodore 64 BASIC and Kernal ROMs are similar. VIC's BASIC ROM starts at $C000 and is exactly $2000 bytes up from the 64 BASIC ROM, which starts at $A000. Both Kernal ROMs start at $E000, but the 64 has an extra JMP instruction to bridge the gap between BASIC and the Kernal, so the addresses of routines in the Kernal initially differ by three bytes between these machines.

Label	VIC/64	Descriptions
BCOLD	**$C000/$A000**	BASIC cold start vector ($E378). NEWs BASIC, prints BYTES FREE and READY. Part of the reset sequence; see routines at $E378 and $FD22.
BWARM	**$C002/$A002**	BASIC warm start vector ($E467). CLRs BASIC, prints READY. Part of the NMI sequence; see routines at $E467 and $FEA9.
	$C004/$A004	CBM BASIC message.
	$C00C/$A00C	Table of addresses−1 of routines for handling BASIC statements (FOR, RUN, PRINT, REM, CONT, etc.). (Address−1 because of the way they are utilized.)
	$C052/$A052	Table of true addresses of routines for handling numeric and string functions (FRE, POS, SQR, etc.).

	$C080/$A080	Table of addresses − 1 of routines for handling BASIC operators (add, subtract, divide, etc.); each address is followed by a byte indicating the operator priority.
	$C09E/$A09E	BASIC keywords as CBM ASCII strings with bit 7 of final character set high.
	$C129/$A129	Table of miscellaneous keywords (TAB, STEP, etc., with no action address) with bit 7 of final character set high.
	$C140/$A140	Table of operator tokens; also AND, OR as strings with bit 7 of final character set high.
	$C14D/$A14D	Table of function keywords (SGN, INT, ABS, etc.) with bit 7 of final character set high.
	$C19E/$A19E	Table of 28 error messages (TOO MANY FILES, FILE OPEN, etc.) with bit 7 of final character set high.
	$C328/$A328	Table of pointers to error messages.
	$C364/$A364	Table of other messages OK, ERROR IN, READY, BREAK.
FNDFOR	**$C38A/$A38A**	Check stack for FOR entry. Called by NEXT; if FOR not found, ?NEXT WITHOUT FOR results. Also clears stack of a FOR data block if called by RETURN.
BLTU	**$C3B8/$A3B8**	Open up a gap in BASIC text to allow insertion of new BASIC line. Check whether there is enough room.
BLTUC	**$C3BF/$A3BF**	Move block starting at address pointed to by $5F/60 and ending at address pointed to by $5A/5B − 1 up to a new block ending at the address pointed to by $58/59 − 1.
GETSTK	**$C3FB/$A3FB**	Test to see whether stack will accommodate A*2 bytes: ?OUT OF MEMORY if not.
REASON	**$C408/$A408**	Check whether address pointed to by A (low byte) and Y (high byte) is below FRETOP (current bottom of string area). If yes, exit; otherwise, do garbage collection and check again. If still not, then print ?OUT OF MEMORY.
ERROR	**$C437/$A437**	Print error message; X holds error number (half of offset within error message address table). Vectored via ($0300) to $C43A. Then set keyboard input and screen output, reset stack and print IN with line number if in program mode.
READY	**$C474/$A474**	Restart BASIC; print READY, set direct mode.
MAIN	**$C480/$A480**	Receive a line into input buffer and add a terminating 0 byte. Check for program line or immediate mode command; if immediate mode command, execute it. MAIN is vectored via ($0302) to $C483.
MAIN1	**$C49C/$A49C**	If program line, tokenize it.
INSLIN	**$C4A4/$A4A4**	If the line number already exists, replace it. If it's new, insert it. Line number is in $14,$15 on entry, length+4 is in Y. If the first byte in buffer is 0, the line is null; delete it.
FINI	**$C52A/$A52A**	Having inserted a new line, do RUNC (thus variables are lost on editing, and you cannot CONT after editing) and LNKPRG; then jump to MAIN.
LNKPRG	**$C533/$A533**	Chain link pointers in BASIC program using end-of-line 0 markers.
INLIN	**$C560/$A560**	Input a screen line into the BASIC text buffer at $200, and add 0 terminating byte.
CRUNCH	**$C579/$A579**	Tokenize keywords in input buffer. Vectored via ($0304) to $C57C.

FNDLIN	**$C613/$A613**	Search BASIC text from beginning for line number in $14/15. Carry set if line found. Locations $5F/60 point to link address.
FNDLNC	**$C617/$A617**	Search BASIC text from address in A (low byte) and X (high byte) for line number in $14/15.
NEW	**$C642/$A642**	NEW routine enters here; check syntax, and continue with SCRTCH.
SCRTCH	**$C644/$A644**	Reset first two bytes of text (first link pointer) to 0; load start-of-variables pointer $2D/2E with start-of-BASIC + 2, and continue with RUNC.
RUNC	**$C659/$A659**	Set pointer within CHRGET to start of BASIC text, using STXPT, then continue with CLEAR.
CLEAR	**$C65E/$A65E**	BASIC CLR routine; erase variables by resetting end-of-variables pointers to coincide with end-of-program pointer; appropriate string variable pointers are also reset. Abort I/0 activity and reset stack.
STXPT	**$C68E/$A68E**	Reset pointer within CHRGET routine to beginning of BASIC text ($2B/2C−1 is loaded into $7A/7B).
LIST	**$C69C/$A69C**	Entry point of routine to process LIST command.
LIST1	**$C6C9/$A6C9**	List one line of BASIC; line number, then text.
QPLOP	**$C717/$A717**	Handle character to be listed; if ordinary character or control character in quotes, print it; expand and print tokens. Vectored via ($0306) to $C71A.
FOR	**$C742/$A742**	Entry point for routine to handle FOR statement. Push 18 bytes onto stack: pointer to following statement, current line number, upper-loop value, step value (default 1), loop variable name, and FOR token.
NEWSTT	**$C7AE/$A7AE**	Execute BASIC; test for STOP key and check for end-of-line zero byte or colon.
CKEDL	**$C7C4/$A7C4**	If at end of text, stop; otherwise, set pointer within CHRGET to beginning of next line.
GONE	**$C7E1/$A7E1**	Handle the BASIC statement in the current line. Vectored via ($0308) to $C7E4, loop back to NEWSTT.
EXCC	**$C7ED/$A7ED**	Execute a BASIC keyword. Uses address for start of routine from table at $C00C. Assumes LET if a token is not the first byte in the statement. Address pushed on stack so RTS of GETCHR jumps to it.
RESTOR	**$C81D/$A81D**	Entry point for routine to handle RESTORE; set the data pointer at $41/42 to start of BASIC text.
STOP	**$C82C/$A82C**	Entry point for routine to handle STOP; also END and break in program. Information for CONT (pointer in BASIC text, line number) is stored. STOP prints BREAK IN *nnn* while END skips this to READY. The STOP key invokes STOP. Reaching the end of BASIC program-text calls END.
CONT	**$C857/$A857**	Entry point for routine to handle CONT; performs this by setting current line number (stored in $39/3A) and the pointer within CHRGET to values stored by STOP. ?CANNOT CONTINUE ERROR occurs if the high byte of the pointer has been set to 0 on syntax error.
RUN	**$C871/$A871**	Entry point for routine to handle RUN; if RUN is encountered alone, then CLR variables and reset stack, set CHRGET to start of BASIC, and begin execution. If RUN *nnn*, CLR variables and reset stack, then do GOTO *nnn*.

GOSUB	**$C883/$A883**	Entry point for routine to handle GOSUB; push five bytes onto stack: pointer within CHRGET (two bytes), current line number (two bytes), and the GOSUB token. The GOTO routine is then called.
GOTO	**$C8A0/$A8A0**	Entry point for routine to handle GOTO; fetch the line number following the GOTO command and search BASIC text for this line. If high byte of destination is higher than high byte of current line number, search from position of current line onwards to shorten search time; otherwise, search from beginning. Put pointer to found line into CHRGET pointer.
RETURN	**$C8D2/$A8D2**	Entry point for routine to handle RETURN; stack is cleared up to GOSUB token (?RETURN WITHOUT GOSUB if not found); then the calling line number and pointer are reinstated, and execution continues.
DATA	**$C8F8/$A8F8**	Entry point for routine to handle DATA statements; routine to let CHRGET skip DATA statement up to terminating byte or colon.
DATAN	**$C906/$A906**	Search for statement terminator; exits with Y containing displacement to end of line from CHRGET's pointer.
REMN	**$C909/$A909**	Search for end of BASIC line.
IF	**$C928/$A928**	Entry point for routine to handle IF statement. Evaluate the expression; if result is false (0), skip the THEN or GOTO clause by doing REM.
REM	**$C93B/$A93B**	Entry point for routine to handle REM; scan for end of line and update pointer in CHRGET, to ignore contents of REM statement.
DOCOND	**$C940/$A940**	Continue IF; if expression true, then execute next command, or do GOTO if digit follows.
ONGOTO	**$C94B/$A94B**	Entry point for routine to handle ON-GOTO and ON-GOSUB statements; evaluate expression, test for GOTO or GOSUB token, scan line number list, skipping commas for specified line number, and GOTO or GOSUB it.
LINGET	**$C96B/$A96B**	Read an integer (usually a line number) from the BASIC text into locations $14 and $15; must be in range 0–63999.
LET	**$C9A5/$A9A5**	Entry point for routine to handle LET statement; find target variable in variable list (or create it if it doesn't exist), test for = token, evaluate expression, and move result or string descriptor into the variable list.
PUTINT	**$C9C4/$A9C4**	Round FAC1 and put, as integer, into variable list at current variable position, pointed to by $49/4A.
PTFLPT	**$C9D6/$A9D6**	Put FAC1 into variable list at location pointed to by $49/4A.
PUTTIM	**$C9E3/$A9E3**	Assign the system variable TI$.
ASCADD	**$CA27/$AA27**	Add ASCII digit to FAC1.
GETSPT	**$CA2C/$AA2C**	LET for strings; put string descriptor pointed to by FAC1+3/FAC1+4 into variable list at location pointed to by $49/4A.
PRINTN	**$CA80/$AA80**	Entry point for routine to handle PRINT# statement; call CMD, then clear I/0 channels and restore default I/0 device numbers.
CMD	**$CA86/$AA86**	Entry point for routine to handle CMD; set output device from file table using Kernal CHKOUT routine, then call PRINT.

STRDON	**$CA9A/$AA9A**	Part of PRINT routine; print string and continue with punctuation of PRINT.
PRINT	**$CAA0/$AAA0**	Entry point for routine to handle PRINT statement; identify PRINT parameters (TAB, SPC, comma, semicolon, etc.), and evaluate expression.
VAROP	**$CAB8/$AAB8**	Print variable; if numeral, convert to string before printing.
CRDO	**$CAD7/$AAD7**	Print carriage return (ASCII 13) followed (if channel <128) by linefeed (ASCII 10).
STROUT	**$CB1E/$AB1E**	Print string beginning at address specified in A (low byte) and Y (high byte), and terminated by a zero byte or quotes.
STRPRT	**$CB21/$AB21**	Print string; FAC1+3/FAC1+4 points to string descriptor.
OUTSTR	**$CB24/$AB24**	Output string; locations $22/23 point to string, length in A.
OUTSPC	**$CB3B/$AB3B**	Output cursor right (or space if the screen is not the current output device).
PRTSPC	**$CB3F/$AB3F**	Output space.
OUTSKP	**$CB42/$AB42**	Output cursor right.
OUTQST	**$CB45/$AB45**	Output question mark for error messages.
OUTDO	**$CB47/$AB47**	Output the character in A.
TRMNOK	**$CB4D/$AB4D**	Output appropriate error messages for GET, READ, and INPUT.
GET	**$CB7B/$AB7B**	Entry point for routine to handle GET and GET# statements; test for direct mode (illegal) and fetch one character from keyboard or file.
INPUTN	**$CBA5/$ABA5**	Entry point for routine to handle INPUT# statement; fetch file number, turn the device on, call INPUT, and then turn the device off.
INPUT	**$CBBF/$ABBF**	Entry point for routine to handle INPUT statement; output user's prompt string if present, then continue with QINLIN routine.
QINLIN	**$CBF9/$ABF9**	Print ? prompt and receive line of text (terminated by RETURN) into input buffer.
READ	**$CC06/$AC06**	Entry point for routine to handle the READ statement. GET and INPUT also share this routine, but are distinguished by a flag in location $11.
INPCON	**$CC0D/$AC0D**	Entry point into READ routine for INPUT; set flag and call READ, with buffer at the address specified in X (low byte) and Y (high byte).
INPCO1	**$CC0F/$AC0F**	Entry point into READ routine for GET; set flag and call READ, with buffer at the address specified in X (low byte) and Y (high byte).
DATLOP	**$CCB8/$ACB8**	Scan text and read DATA statements.
VAREND	**$CCDF/$ACDF**	Tests for 0 at end of input buffer; if not found, print ?EXTRA IGNORED.
EXINT	**$CCFC/$ACFC**	Messages ?EXTRA IGNORED and ?REDO FROM START.
NEXT	**$CD1E/$AD1E**	Entry point for routine to handle NEXT; check for FOR token and matching variable on stack, and print ?NEXT WITHOUT FOR if not found; calculate next value. If the loop increment is still valid, reset current line number and the pointer in CHRGET and continue.
FRMNUM	**$CD8A/$AD8A**	Evaluate a numeric expression for BASIC by calling FRMEVL, then CHKNUM.

CHKNUM	**$CD8D/$AD8D**	Check that FRMEVL has returned a number by testing flag at location $0D. If a number was not returned, issue a ?TYPE MISMATCH ERROR message.
CHKSTR	**$CD8F/$AD8F**	Check that FRMEVL has returned a string by testing flag at location $0D. If a string was not returned, issue a ?TYPE MISMATCH ERROR message.
FRMEVL	**$CD9E/$AD9E**	Evaluate any BASIC expression in text and report any syntax errors; set $0D (VALTYP) to $00 if the expression is numeric and $FF if it is a string. For numeric expressions, location $0E (INTFLG) is set to $00 if the expression is floating point, and the value is placed in FAC1. If the variable type is integer, set INTFLG to $80, but leave the result in floating point format in FAC1. Complicated expressions may need simplifying to retain stack space and prevent ?OUT OF MEMORY.
EVAL	**$CE83/$AE83**	Evaluate a single term in an expression; look for ASCII numeral strings, variables, pi, NOT, arithmetic functions, etc.
PIVAL	**$CEA8/$AEA8**	Value of pi in five-byte floating point format.
PARCHK	**$CEF1/$AEF1**	Evaluate expression within parentheses.
CHKCLS	**$CEF7/$AEF7**	Check whether CHRGET points to a) character; issue a ?SYNTAX ERROR message if not.
CHKOPN	**$CEFA/$AEFA**	Check whether CHRGET points to a (character; issue a ?SYNTAX ERROR message if not.
CHKCOM	**$CEFD/$AEFD**	Check whether CHRGET points to a comma; issue a ?SYNTAX ERROR message if not.
SYNCHR	**$CEFF/$AEFF**	Check whether CHRGET points to byte identical to that in A; if it does, routine exits with next byte in A; otherwise, a ?SYNTAX ERROR message is issued.
SYNERR	**$CF08/$AF08**	Output a ?SYNTAX ERROR message and return to READY.
DOMIN	**$CF0D/$AF0D**	Evaluate NOT.
TSTROM	**$CF14/$AF14**	Set carry flag to 1 if FAC1+3/FAC1+4 point to the ROM area indicating reserved variables TI$, TI, ST.
ISVAR	**$CF28/$AF28**	Search variable list for variable named in locations $45/46; on exit FAC1 will hold numeric value in FLPT format (whether integer or floating point variable); FAC1+3/FAC1+4 will point to the descriptor if it's a string variable.
TISASC	**$CF48/$AF48**	Read clock and set up string containing TI$.
ISFUN	**$CFA7/$AFA7**	Identify function type and evaluate it.
OROP	**$CFE6/$AFE6**	Entry point for routine to handle the OR function; set flag and do OR between two two-byte integers in FAC1 and FAC2.
ANDOP	**$CFE9/$AFE9**	Entry point for routine to handle the AND function. Both AND and OR are performed by one routine; a flag (in Y) holds $FF for OR, $00 for AND. Convert FLPT to integer (and give an error message if the result is out of range). The result in FLPT format is left in FAC1.
DOREL	**$D016/$B016**	Entry point for routine to handle string and numeric comparisons (< = >). Check variable types, then continue with NUMREL or STRREL, as appropriate.
NUMREL	**$D01B/$B01B**	Perform numeric comparison, using FCOMP at $DC5B.
STRREL	**$D02E/$B02E**	Perform string comparison; exit with X holding $00 if strings equal, $01 if the first string is greater than the second, and $FF if the second is greater than the first.

DIM	**$D081/$B081**	Entry point for routine to handle the DIM statement; set up each array element using the PTRGET routine.
PTRGET	**$D08B/$B08B**	Validate a variable name in BASIC text; the first character must be alphabetic, the second may be either alphabetic or numeric; subsequent alphanumerics are discarded. Set VALTYP (location $0D) to $FF to indicate a string variable if $ is found; otherwise, set VALTYP to $00 to indicate a numeric variable. Set INTFLG (location $0E) to $80 to indicate an integer variable if % is found. The name is stored in VARNAM (locations $45/46) with high bits set to indicate the variable type, as described in Chapter 5.
ORDVAR	**$D0E7/$B0E7**	Search variable list for variable whose name is in VARNAM (locations $45/46) and set VARPNT (locations $47/48) to point to it. Create new variable if the name is not currently in the list.
ISLETC	**$D113/$B113**	Set the carry flag if the accumulator holds A–Z.
NOTFNS	**$D11D/$B11D**	Create a new simple (not array) variable in variable list immediately before arrays; name is in VARNAM ($45/46). Any arrays have to be moved up by seven bytes to accommodate the new variable. Exit with locations $5F/60 pointing to newly created variable.
FMAPTR	**$D194/$B194**	Calculate pointer value in $5F/60, to be used when setting up space for arrays.
N32768	**$D1A5/$B1A5**	Holds −32768 as a five-byte floating point number.
FACINX	**$D1AA/$B1AA**	Convert contents of FAC1 to two-byte integer (−32767 to +32768) in A/Y.
INTIDX	**$D1B2/$B1B2**	Fetch and evaluate a positive integer expression from the next part of BASIC text; if result is 0 to 32767, store in FAC1+3 and FAC1+4.
AYINT	**$D1BF/$B1BF**	Convert the contents of FAC1 to integer in range 0 to 32767; leave the result in FAC1+3/FAC1+4.
ISARY	**$D1D1/$B1D1**	Get array parameters from BASIC text (number of dimensions and number of elements) and push the values onto the stack.
FNDARY	**$D218/$B218**	Find array named in VARNAM ($45/46), with other details of the array stored on the stack.
BSERR	**$D245/$B245**	BAD SUBSCRIPT error. BSERR+3 will print ILLEGAL QUANTITY error message.
NOTFDD	**$D261/$B261**	If the specified array is not found, create it using details on stack with DIMension 10.
INPLN2	**$D30E/$B30E**	Locate specified element within array and point VARPNT ($47/48) to it.
UMULT	**$D34C/$B34C**	Compute offset of specified array element relative to array pointed at by VARPNT ($47/48); put in X/Y.
FRE	**$D37D/$B37D**	Entry point for routine to handle FRE function; perform garbage collection and set Y/A to point to lowest string minus pointer to end of arrays; then place in FAC1 by calling.
GIVAYF	**$D391/$B391**	Convert two-byte integer in Y/A (range −32768 to +32767) to FLPT in FAC1.
POS	**$D39E/$B39E**	Entry point for routine to handle POS function; calls Kernal routine PLOT to fetch cursor position, then loads it into FAC1 using SNGET.

SNGET	**$D3A2/$B3A2**	Convert byte in Y to FLPT in FAC1 (0 to 255).
ERRDIP	**$D3A6/$B3A6**	Test that command was not entered in direct mode; CURLIN+1 ($3A) containing $FF indicates direct mode. ?ILLEGAL DIRECT ERROR if it was. Called by routines that may not be used in direct mode (for example, GET).
DEF	**$D3B3/$B3B3**	Entry point for routine to handle DEF statement; create function definition and find or set up dependent variable. When a FN is invoked, the pointer within CHRGET is set to the beginning of the FN definition in the BASIC text and the expression found there is evaluated; it is then switched back. Information to enable it to do this is stored within the function variable set up in GETFNM.
GETFNM	**$D3E1/$B3E1**	Check syntax of FN; find or set up variable with function name and set DEFPNT ($4E/4F) to point to it (must be numeric, not string, variable).
FNDOER	**$D3F4/$B3F4**	Evaluate function; evaluate expression within parentheses in statement invoking function, leaving it in FAC1, then evaluate the FN expression (see DEF).
STRD	**$D465/$B465**	Entry point for routine to handle STR$ function; evaluate expression and convert to ASCII string.
STRINI	**$D475/$B475**	Make room in string space for a string to be inserted: A contains length and (FAC1+3) points to the string. On exit $61–$63 contains descriptor for new string. CHR$, LEFT$, and so on all use this routine.
STRLIT	**$D487/$B487**	Copy a string into string space at top of memory; A/Y points to the start of the string. Scans for ", : or 0 byte as terminator to determine length. Exit with descriptor in $61–$63.
GETSPA	**$D4F4/$B4F4**	Allocate space for string, length in A, in dynamic string space at top of memory; do garbage collection if space exhausted. Called by STRINI.
GARBA2	**$D526/$B526**	Do garbage collection; eliminate unwanted strings in string area and collect together valid strings. The garbage collection routine is slow for large numbers of strings.
DVARS	**$D606/$B606**	Search variables and arrays for next string to be saved by garbage collection.
CAT	**$D63D/$B63D**	Concatenate two strings.
MOVINS	**$D67A/$B67A**	Move string to string area high in RAM; entered with $6F/70 pointing at the descriptor of the string to be stored.
FRESTR	**$D6A3/$B6A3**	Discard string; entered with pointer to string descriptor in FAC1+3/FAC1+4, exits with new string length and pointer in INDEX1.
FRETMS	**$D6DB/$B6DB**	Clean the descriptor stack.
CHRD	**$D6EC/$B6EC**	Entry point for routine to handle CHR$ function; sets up a one-byte string.
LEFTD	**$D700/$B700**	Entry point for routine to handle LEFT$.
RIGHTD	**$D72C/$B72C**	Entry point for routine to handle RIGHT$.
MIDD	**$D737/$B7D7**	Entry point for routine to handle MID$.
PREAM	**$D761/$B761**	Pull string descriptor pointer to $50/51, length to A (also in X).
LEN	**$D77C/$B77C**	Entry point for routine to handle LEN function; floating point value of string length parameter placed in FAC1.

LEN1	**$D782/$B782**	Extract length of string, put in Y, leave string mode, and enter numeric mode. Called by LEN, VAL.
ASC	**$D78B/$B78B**	ASC function; get first character of string and convert to floating point in FAC1. String of length zero gives ?SYNTAX ERROR.
GTBYTC	**$D79B/$B79B**	Read and evaluate an expression from BASIC text; must evaluate to a one-byte value; value left in X and FAC1+4.
VAL	**$D7AD/$B7AD**	Entry point for routine to handle VAL function; convert value to floating point value in FAC1.
GETNUM	**$D7EB/$B7EB**	Read parameters for WAIT and POKE from BASIC text; put first (two-byte integer) in $14/15, second in X.
GETADR	**$D7F7/$B7F7**	Convert FAC1 to two-byte integer (range 0–65535) in $14/15 and Y/A.
PEEK	**$$D80/$B80D**	Entry point for routine to handle PEEK function; on entry FAC1 contains address to be PEEKed in FLPT form; exit with PEEKed value in Y.
POKE	**$D824/$B824**	Entry point for routine to handle POKE statement; fetch two parameters from BASIC text; do POKE.
WAIT	**$D82D/$B82D**	Entry point for routine to handle WAIT statement; fetch two parameters from text, plus optional third, which is 0 if none found; do WAIT loop.
FADDH	**$D849/$B849**	Add 0.5 to contents of FAC1; used when rounding.
FSUB	**$D850/$B850**	Floating point subtraction; FAC1 is replaced by MFLPT value pointed to by A/Y, minus FAC1.
FSUBT	**$D853/$B853**	Entry point for routine to handle floating point subtraction; FAC1 is replaced by FAC2 minus FAC1.
FADD	**$D867/$B867**	Floating point addition; FAC1 is replaced by MFLPT value pointed to by A/Y, plus FAC1.
FADDT	**$D86F/$B86F**	Entry point for routine to handle floating point addition; FAC1 is replaced by FAC2 plus FAC1. On entry, A holds FAC1's exponent (contents of $61) to speed the addition in the event that FAC1 contains zero.
COMPLT	**$D947/$B947**	Replace FAC1 with twos complement of the value currently there.
OVERR	**$D97E/$B97E**	Output ?OVERFLOW ERROR message, then READY.
MULSHF	**$D983/$B983**	Multiply by a byte.
FONE	**$D9BC/$B9BC**	Table of constants in MFLPT format: first 1, then constants for LOG evaluation; SQR(0.5), SQR(2), −0.5, and LOG(2).
LOG	**$D9EA/$B9EA**	Entry point for routine to handle the LOG function; compute logarithm to the base e of FAC1.
FMULT	**$DA28/$BA28**	Floating point multiply; FAC1 is replaced by MFLPT value pointed to by A/Y times FAC1.
FMULTT	**$DA30/$BA30**	Entry point for routine to handle floating point multiplication; FAC1 is replaced by FAC1 times FAC2.
MLTPLY	**$DA59/$BA59**	Multiply FAC1 by a byte and store in $26–$2A.
CONUPK	**$DA8C/$BA8C**	Load FAC2 from MFPLT value pointed to by A/Y, unpacking sign bit and storing it separately, forming FLPT format. On exit A holds FAC1's first byte.
MULDIV	**$DAB7/$BAB7**	Test floating point accumulators for multiply and divide; if FAC2 is 0, set FAC1 to 0; if exponents together are too large then ?OVERFLOW ERROR. If they are too small, force the result to 0 without an underflow message.

MUL10	**$DAE2/$BAE2**	Multiply FAC1 by 10 and put result in FAC1.
TENC	**$DAF9/$BAF9**	The value 10 in MFLPT format.
DIV10	**$DAFE/$BAFE**	Divide FAC1 by 10 and put result in FAC1.
FDIVF	**$DB07/$BB07**	Floating point division; FAC1 is replaced by FAC2 divided by MFLPT value pointed at by A/Y; on entry X contains sign of result.
FDIV	**$DB0F/$BB0F**	Floating point division; FAC1 is replaced by MFLPT divided by FAC1.
FDIVT	**$DB14/$BB14**	Entry point for routine to handle floating point division; FAC1 is replaced by FAC2 divided by FAC1. On entry, A holds FAC1's first byte.
MOVFM	**$DAB2/$BBA2**	Load FAC1 from MFLPT value pointed to by A/Y, unpacking sign bit and storing it separately, forming FLPT format.
MOV2F	**$DBC7/$BBC7**	Convert FAC1 to MFLPT format and store at $5C–$60, TEMPFP2.
MOV1F	**$DBCA/$BBCA**	Convert FAC1 to MFLPT format and store at $57–$5B, TEMPFP1.
MOVVF	**$DBD0/$BBD0**	Convert FAC1 to MFLPT format and store at address pointed to by $49/4A.
MOVMF	**$DBD4/$BBD4**	Convert FAC1 to MFLPT format and store at address pointed to by A/Y .
MOVFA	**$DBFC/$BBFC**	Copy FAC2 into FAC1.
MOVAF	**$DC0C/$BC0C**	Round FAC1 by calling ROUND, then copy into FAC2.
ROUND	**$DC1B/$BC1B**	Round FAC1.
SIGN	**$DC2B/$BC2B**	Get sign of FAC1; on exit A=0 if value is 0, A=1 if value is positive, A=$FF if value is negative.
SGN	**$DC39/$BC39**	Entry point for routine to handle SGN function; calls SIGN, then converts A into floating point form in FAC1.
ABS	**$DC58/$BC58**	Entry point for routine to handle ABS function; replace FAC1 with the absolute value of the current contents of FAC1.
FCOMP	**$DC5B/$BC5B**	Compare FAC1 with MFLPT value pointed to by A/Y; on exit A=0 if values were equal, A=1 if FAC1>MFLPT, A=$FF if FAC1<MFLPT.
QINT	**$DC9B/$BC9B**	Convert FAC1 to four-byte integer in FAC1+1 to FAC1+4, highest byte first.
INT	**$DCCC/$BCCC**	Entry point for routine to handle INT function; round down FAC1 but leave it in FAC1 in FLPT form.
FIN	**$DCF3/$BCF3**	Convert an ASCII string (for example, "−99.375") to a floating point value in FAC1. On entry, TXTPTR points to the start of the string, then JSR GETCHR/JSR FIN accomplishes the conversion.
AADD	**$DD7E/$BD7E**	Add contents of A to FAC1.
STCONS	**$DDB3/$BDB3**	Three constants used in string conversions, in MFLPT form: 99999999.9, 999999999, 1000000000.
INPRT	**$DDC2/$BDC2**	Print IN followed by current line number in CURLIN ($39/3A).
LINPRT	**$DDCD/$BDCD**	Output integer in A/Y, range 0–65535.
FOUT	**$DDDD/$BDDD**	Convert contents of FAC1 to ASCII string starting at location $100 and ending with 0 byte. On exit, A/Y holds start address, so STROUT can print string.
FOUTIM	**$DE68/$BE68**	Convert TI to ASCII string starting at $100 and ending with 0 byte.

TICONS	**$DF11/$BF11**	String and TI conversion constants: .5 in MFLPT form, then 15 four-byte integer constants.
SQR	**$DF71/$BF71**	Entry point for routine to handle SQR function; FAC1 is replaced by square root of FAC1.
FPWRT	**$DF7B/$BF7B**	Entry point for routine to perform power calculation; FAC1 is replaced by FAC2 raised to the power of FAC1. On entry, A must hold contents of FAC2 so powers of 0 are correct.
NEGOP	**$DFB4/$BFB4**	Negate FAC1.
EXCONS	**$DFBF/$BFBF**	Table of eight constants for evaluating EXP series.
EXP	**$DFED/$BFED**	Entry point for routine to handle EXP function; FAC1 is replaced by e raised to FAC1.
POLYX	**$E056/$E059**	Series evaluation routine. Entered with A/Y pointing to the counter at the beginning of the table of constants used in the power series evaluation.
RMULC	**$E08A/$E08D**	The value 11879546.4 in MFLPT format; multiplicative constant for RND evaluation.
RADDC	**$E08F/$E092**	The value 3.92767778E−8 in MFLPT format; additive constant for RND evaluation.
RND	**$E094/$E097**	Entry point for routine to handle RND function; set FAC1 to a number according to sign of FAC1 by branching to either RND0, QSETNR, or RND1.
RND0	**$E09B/$E09E**	If FAC1=0, load FAC1 from VIA timer registers; a simple way of reseeding it with a random number.
QSETNR	**$E0BB/$E0BE**	If FAC1>0, load FAC1 with the result of multiplying the stored random number (in $88–$8C) generated by previous calls, by RMULC, and adding RADDC.
RND1	**$E0D0/$E0D3**	If FAC1<0, load FAC1 with mixed digits from FAC1 itself, so RND with a negative argument is constant and therefore repeatable. After any of these three conditions, FAC1 is stored in $88–$8C.
RNDRNG	**$E0E0/$E0E5**	Force the value in FAC1 into the range 0–1 excluding 0 and 1.
BIOERR	**$E0F6/$E0F9**	I/0 error message routine if any of the following calls return error flags:.
BCHOUT	**$E109/$E10C**	Output character; uses CHROUT.
BCHIN	**$E10F/$E112**	Input character; uses CHRIN.
BCKOUT	**$E115/$E118**	Set up for output; uses CHKOUT.
BCKIN	**$E11B/$E11E**	Set up for input; uses CHKIN.
BGETIN	**$E121/$E124**	Get one character; uses GETIN.
SYS	**$E127/$E12A**	Entry point for routine to handle SYS statement; load A, X, Y, and SR from locations $30C to $30F, call machine language routine at address specified by the argument, then reload the register contents into $30C–$30F on return from the routine.
SAVET	**$E153/$E156**	Entry point for routine to handle SAVE; save a BASIC program. Set A to point to address in zero page pointing to start address, set X/Y to the value in $2D/2E (end-of-program pointer). Then Kernal routine SAVE is called via vector at $FFD8.
VERFYT	**$E162/$E165**	Entry point for routine to handle VERIFY; set flag in A to indicate VERIFY operation, enter LOADT and check for errors.
LOADT	**$E165/$E168**	Entry point for routine to handle LOAD; fetch parameters from BASIC text and set them up, call Kernal routine LOAD via vector at $FFD5 .

LOADR	**$E177/$E16F**	Load from device already set, into RAM starting at start of BASIC address pointed to by $2B/2C.
LDFIN	**$E195/$E195**	Finish LOAD; if LOAD was called in direct mode, set top-of-BASIC pointer ($2D/2E) to address of last byte loaded. This step is omitted if the routine is called from within a program, so variable list is preserved. Finally, reset pointer in CHRGET and warm start BASIC to run the new program.
OPENT	**$E1BB/$E1BE**	Entry point for routine to handle OPEN; read parameters from text and set them up via appropriate Kernal calls; call Kernal OPEN routine via vector at $FFC0.
CLOSET	**$E1C4/$E1C7**	Entry point for routine to handle CLOSE; read parameters from text and set them up; call Kernal CLOSE routine via vector at $FFC3.
SLPARA	**$E1D1/$E1D4**	Fetch parameters for LOAD, SAVE, and VERIFY from BASIC text; set defaults if not supplied. Set up file by a call to SETLFS via vector at $FFBA.
COMBYT	**$E1FD/$E200**	Check for comma and evaluate the following one-byte parameter, which is put in X.
CMMERR	**$E20B/$E20E**	Check for comma followed by anything other than end of statement; otherwise, issue a ?SYNTAX ERROR message.
OCPARA	**$E216/$E219**	Get parameters from BASIC text for OPEN/CLOSE calls; set defaults if not supplied.
COS	**$E261/$E264**	Entry point for routine to handle the COS function; the value in FAC1 is replaced by the cosine of that value.
SIN	**$E268/$E26B**	Entry point for routine to handle the SIN function; the value in FAC1 is replaced by the sine of that value.
TAN	**$E2B1/$E2B4**	Entry point for routine to handle the TAN function; the value in FAC1 is replaced by the tangent of that value.
	$E2DD/$E2E0	Table of constants in MFLPT format: $\pi/2$, $\pi*2$, and 0.25. Then comes a counter value (5) and six MFLPT constants used in evaluating SIN, COS, and TAN.
ATN	**$E30B/$E30E**	Entry point for routine to handle ATN; the value in FAC1 is replaced by the arc tangent of that value.
	$E33B/$E33E	A counter value (11) and table of 12 constants in MFLPT format for ATN evaluation.
INIT	**$E378/$E394**	BASIC cold start routine, entered on JMP ($C000); part of the reset sequence. Performs INITV, INITCZ, INITMS; sets stack and jumps to READY.
CHRCPY	**$E387/$E3A2**	CHRGET routine and RND seed in ROM for relocation into RAM.
INITCZ	**$E3A4/$E3BF**	Initialize USR jump instruction and default vector, vectors from $003 to $006; transfer CHRGET and RND seed to RAM; call Kernal routines MEMBOT and MEMTOP to set start-of-BASIC and top-of-memory pointers ($2B/2C and$37/38) from the pointers at $282–$285 initialized on power-up. Set end-of-program 0 byte at 4096.
INITMS	**$E404/$E422**	Output start-up message:**** CBM BASIC V2 ****, then number of free bytes, then BYTES FREE
INITV	**$E45B/$E453**	Initialize vectors for ERROR, MAIN, etc., at $0300–$030B.

BASSFT	**$E467/$E37B**	BASIC warm start routine, entered on JMP ($C002); part of the break sequence performed if BRK instruction encountered or STOP-RESTORE keys are pressed. Close all I/0 channels, initialize stack, output ?BREAK ERROR, and jump to READY.
CINT	**$E518/$E518**	General screen and VIC chip initialization; set up screen editing tables at $D9–$F0, initialize VIC chip, set character color to blue, do CLR and HOME, reset default I/0 device numbers at $99 and $9A.
HOME	**$E581/$E566**	Home the cursor.
INITVC	**$E5C3/$E5A8**	Initialize the VIC chip from table of values at $EDE4–$EDF3 (international variations).
GETKBC	**$E5CF/$E5B4**	Get character from keyboard queue and move remaining characters along; queue must contain at least one character on entry (queue size is stored in $C6). On exit the character is in A.
INPPRO	**$E5E5/$E5CA**	Input and process SHIFT-STOP, RETURN, etc.
QTSWC	**$E6B8/$E684**	Flip quotes flag ($D4) if A contains quotes on entry.
PRT	**$E742/$E716**	Print character in A to screen, like PRINT CHR$; handles such characters as home cursor, clear screen, delete, etc.
CHKCOL	**$E912/$E8CB**	Test A for character color code; change color in $286 if one is found.
COLTAB	**$E921/$E8DA**	Table of color-change codes, arranged Black, White, Red, Cyan, etc.
SCROL	**$E975/$E8EA**	Scroll screen up. If the top line is more than 22 characters long, the routine scrolls up appropriate number of lines to completely remove it. The CTRL key is tested for by directly interrogating the VIA chip, and a slight delay is performed if it is held down.
DSPP	**$EAA1/$EA13**	Put the character in A onto the screen at the current cursor position; no checking for control characters, etc., is performed. The color for the character is held in X.
KEY	**$EABF/$EA31**	Interrupt servicing routine: All IRQ interrupts are processed by this routine unless the vector in $0314/0315 has been altered. The functions of KEY are to update the clock and location $91 using Kernal routine UDTIM, maintain flashing cursor if cursor is enabled (see $CC–$CF), set the cassette motor on or off according to the flags at $C0, and test the keyboard for new character using Kernal routine SCNKEY. Finally, the A, X, and Y registers are pulled from the stack and restored (they are pushed there by the routine at $FF72) and a return from interrupt instruction (RTI) continues processing the main program.
KBDTBL	**$EC46/$EB81**	Tables to convert keyboard matrix values to CBM ASCII values.
VICINT	**$EDE4/$ECB9**	Table of values from which VIC chip is initialized (the exact values vary internationally, depending on the local television standards).
LDRUN	**$EDF4/$ECC9**	The characters LOAD <RETURN> RUN <RETURN>, transferred to the keyboard queue when SHIFT-RUN.
RSTRAB	**$EFA3/$EEBB**	Part of the routine used by NMI when servicing RS-232 output.

	$F09F/$EF2E	Flag RS-232 errors into ST byte.
	$F0ED/$F017	Output RS-232 character.
	$F14F/$F086	Get RS-232 character.
	$F17F/$F0CA	Tape messages.
SPMSG	**$F1E2/$F12B**	Output Kernal message from table starting at $F174 if flag at $9D is set.
	$F230/$F179	Get character from tape.
	$F290/$F1DD	Output character to tape.
	$F44B/$F38B	Open tape file.
	$F495/$F3D5	Open serial device (printer, disk) file.
	$F4C7/$F409	Open RS-232 file.
	$F563/$F4BF	Load from disk.
	$F5D1/$F539	Load from tape.
	$F692/$F5FA	Save to disk.
	$F6F8/$F65F	Save to tape.
	$F77E/$F6FB	Table of I/0 error numbers (1–9) and messages.
FAH	**$F7AF/$F72C**	Load next tape header.
	$F7EF/$F76A	Write tape header.
	$F867/$F7EA	Load named tape header.
	$F8C9/$F84A	Load tape.
	$F8E6/$F867	Write tape.
READ	**$F98E/$F92C**	Routines for tape reading.
WRITE	**$FBEA/$FBA6**	Routines for tape writing.
START	**$FD22/$FCE2**	Reset routine; entered from the 6502 RESET vector at $FFFC. If a ROM cartridge is present, JMP ($A000) runs it. Otherwise, the routine calls RAMTAS, RESTOR, IOINIT, CINT, and NEW. Note that all other RAM is unaltered, so BASIC programs can be recovered after reset.
RAMTAS	**$FD8D/$FD50**	Fill low RAM (except for the stack area) with zeros, find the start and end of contiguous RAM for BASIC, and set the appropriate screen position according to the amount of memory present.
IOINIT	**$FDF9/$FDA3**	Initialize VIA chips on power-up.
NMI	**$FEA9/$FE43**	NMI routine; entered from the 6502 NMI vector at $FFFA. JMP ($318) at $FEAA routes control back to $FEAD; altering this vector is one way to modify STOP-RESTORE. If a ROM cartridge is present, the routine will warm start it by JMP ($A002). If the STOP key is down, Kernal routines RESTOR, IOINIT, and CINT are called, and a warm start of BASIC is performed by doing a JMP ($C002). This sequence is also performed on BRK. Otherwise, the interrupt is the result of RS-232 activity.
	$FF5C/$FEC2	RS-232 baud rate table (22 bytes; varies internationally).
PULS	**$FF72/$FF48**	IRQ or BRK routine; entered from the 6502 IRQ vector at $FFFE. Save the contents of A, X, and Y on the stack, and examine the status register already pushed onto the stack to determine whether a hardware IRQ interrupt or the execution of a BRK instruction occurred. If it was a standard IRQ interrupt, perform JMP ($314), usually to KEY at $EABF; if it was a BRK operation, JMP ($316), usually to part of the NMI sequence at $FED2, which resets chips and restarts BASIC.

Kernal Jump Table Routines

In Commodore computers, the uppermost half-page of ROM contains a very important collection of vectors known as the Kernal jump table. Each three-byte table entry consists of a JMP instruction and a two-byte address. The JMP may be either a direct JMP to an absolute address in ROM, or an indirect JMP through a RAM vector, such as those in locations $314–$333. The significance of the table is that the location of table entries should remain fixed regardless of future revisions of the ROM routines. Also, many of the locations for table entries are common to all the different Commodore computers.

For example, if you wish to use the Kernal routine which prints a character (held in the accumulator) to the screen, you could go directly to that routine with JMP $F27A, but then your program would work only on the VIC, and might not work with future VICs if the ROMs are revised. However, if you use JMP $FFD2, the jump table entry for the CHROUT routine, you could have some assurance that your program would still work on future VICs; moreover, that jump table entry would also work on the Commodore 64 and PET/CBM computers as well, even though the printing routines for those models are at different locations. On the VIC, JMP $FFD2 arrives at $F27A via an indirect jump through the RAM vector in locations $326/327.

The VIC's Kernal Jump Table begins at location $FF8A. Note that the table entries have their own labels, which may be different from the labels of the routines they point to. Both the table entry label and the Kernal routine label are shown in the list that follows.

Label	VIC/64	Jump Table Entry	Descriptions
RESTOR	$FD52/$FD15	$FF8A RESTOR	Restore standard input/output vectors.
VECTOR	$FD57/$FD1A	$FF8D VECTOR	Store/set input/output vectors.
SETMSG	$FE66/$FE18	$FF90 SETMSG	Enable/disable message output to screen.
SECNDK	$EEC0/$EDB9	$FF93 SECOND	Send secondary address for LISTEN command on serial bus; LISTEN must be called before using this routine.
TKSAK	$EECE/$EDC7	$FF96 TKSA	Send secondary address for TALK command on serial bus; TALK must be called before using this routine.
MEMTOP	$FE73/$FE25	$FF99 MEMTOP	Read/set BASIC top-of-memory limit.
MEMBOT	$FE82/$FE34	$FF9C MEMBOT	Read/set BASIC bottom-of-memory limit.
SCNKK	$EB1E/$EA87	$FF9F SCNKEY	Scan keyboard.
SETTMO	$FE6F/$FE21	$FFA2 SETTMO	Set serial bus timeout.
ACPTRK	$EF19/$EE13	$FFA5 ACPTR	Get a byte from a serial device (usually disk).
CIOUTK	$EEE4/$EDDD	$FFA8 CIOUT	Output a byte to a serial device (usually a printer or disk).
UNTLKK	$EEF6/$EDEF	$FFAB UNTALK	Send an UNTALK command to devices on the serial bus.
UNLSNK	$EF04/$EDFE	$FFAE UNLSN	Send an unlisten command to devices on the serial bus.
LISTNK	$EE17/$ED0C	$FFB1 LISTN	Cause a device on the serial bus (usually a printer or disk) to listen.

TALKK	**$EE14/$ED09**	**$FFB4 TALK**	Cause a device on the serial bus (usually a disk drive) to talk.
READSS	**$FE57/$FE07**	**$FFB7 READST**	Read status byte into A.
SETLFS	**$FE50/$FE00**	**$FFBA SETLFS**	Set file number, device number, and secondary address.
SETNAM	**$FE49/$FDF9**	**$FFBD SETNAM**	Set filename.
NOPEN	**$F40A/$F34A**	**$FFC0 OPEN**	Open a file for reading or writing.
NCLOSE	**$F34A/$F291**	**$FFC3 CLOSE**	Close a file.
NCHKIN	**$F2C7/$F20E**	**$FFC6 CHKIN**	Prepare a file for input.
NCKOUT	**$F309/$F250**	**$FFC9 CHKOUT**	Prepare a file for output.
NCLRCH	**$F3F3/$F333**	**$FFCC CLRCHN**	Restore default I/0 devices.
NBASIN	**$F20E/$F157**	**$FFCF CHRIN**	Get a character from the designated input device.
NBSOUT	**$F27A/$F1CA**	**$FFD2 CHROUT**	Send a character to the designated output device.
LOADSP	**$F542/$F49E**	**$FFD5 LOAD**	Load data into memory from disk or tape.
SAVESP	**$F675/$F5DD**	**$FFD8 SAVE**	Save memory block to disk or tape.
SETTMK	**$F767/$F6E4**	**$FFDB SETTIM**	Set TI clock.
RDTIMK	**$F760/$F6DD**	**$FFDE RDTIM**	Read TI clock.
NSTOP	**$F770/$F6ED**	**$FFE1 STOP**	Test whether STOP key is pressed.
NGETIN	**$F1F5/$F13E**	**$FFE4 GETIN**	Get a character, usually from the keyboard.
NCLALL	**$F3EF/$F32F**	**$FFE7 CLALL**	Abort all I/0 and close all files.
UDTIMK	**$F734/$F69B**	**$FFEA UDTIM**	Add 1 to TI clock; reset to 0 if the count reaches 240000.
SCRENK	**$E505/$E505**	**$FFED SCREEN**	Return the maximum number of screen columns and rows (for the VIC, 22 and 23).
PLOTK	**$E50A/$E50A**	**$FFF0 PLOT**	Move the cursor to a specified row and column, or read the current row and column position of the cursor.
IOBASK	**$E500/$E500**	**$FFF3 IOBASE**	Find the starting address of the keyboard VIA chip registers.

6502 Hardware Vectors

The 6502 microprocessor chip reserves the highest six bytes of the address space (locations $FFFA–$FFFF) for use as vectors. These three vectors point to routines that handle processing under three special sets of circumstances. The chip automatically causes a JMP through one of these vectors when external hardware sends a signal on the 6502's NMI, RESET, or IRQ lines.

The list below shows the label of the vector, the address of the first byte of the vector, and the address to which the vector points in the VIC and 64.

Label	Vector	VIC/64	Descriptions
NMI	**$FFFA**	**$FEA9/$FCE2**	When the 6502 receives a NMI (Non-Maskable Interrupt) signal, it causes a jump to the address held here.
RESET	**$FFFC**	**$FD22/$FE43**	When the 6502 receives a RESET signal, it causes a jump to the address held here.
IRQ	**$FFFE**	**$FF72/$FF48**	When the 6502 receives an IRQ (Interrupt ReQuest) signal or processes a machine language BRK instruction, it causes a jump to the address held here.

Chapter 12

Graphics

Chapter 12

Graphics

This chapter begins with the simplest types of graphics using only ordinary BASIC and progresses to user-defined graphics with motion. It is divided into eight sections.

Graphics using VIC's built-in characters. This section looks at the way the character sets are designed and stored and at BASIC methods for using them.

Machine language subroutines. Discusses subroutines that perform graphics tasks—for instance, reversing characters or plotting 44 × 46 double-density graphics.

Graphics and the VIC chip. Beginning with the easier aspects (color, color RAM, and multicolor mode), this section moves on to topics like position and dimensions of the TV display. Applications include defining extra-large screens, and smooth screen scrolling.

The VIC chip and user-defined characters. Topics covered include the position of screen memory and the position of character definitions, as well as methods for saving and reloading your own character sets.

Utility programs. Several graphics utility programs are presented, including 3 × 3 editors in high-resolution and multicolor modes and full-screen editors.

Other techniques. Subjects covered include user-defined lettering, use of the interrupt in split-screen graphics, the interrupt and flags, and achieving motion with graphics.

Notes on game graphics. A discussion of graphics techniques in games.

The *Super Expander*. A critical look, with detailed notes, at the graphics capabilities of the *Super Expander*.

Graphics with Ordinary BASIC

Effective graphics can be obtained on the VIC with ordinary BASIC. The range of effects is similar to that found on PET/CBM machines, with the addition of color. Before going into programming details, however, it is helpful to look at the internal structure of the VIC-20's graphics system.

How Standard Characters Are Stored in the VIC-20

On the screen, each character is made up from a matrix of 8 × 8 dots. The actual pattern of dots for each character is stored in ROM from $8000 (32768) to $8FFF (36863), occupying a total of 4K of memory. In the non-Japanese VIC-20, there are four subdivisions of this ROM:

$8000 (32768)–$83FF	Uppercase plus extended graphics
$8400 (33792)–$87FF	Reversed uppercase plus extended graphics
$8800 (34816)–$8BFF	Lowercase with uppercase and some graphics
$8C00 (35840)–$8FFF	Reversed lowercase with uppercase and some graphics

Each byte is made up of eight bits, which correspond to a single row of dots. Thus, every character is defined by exactly eight bytes. To show how this works, type FOR J=32768 TO 32775: PRINT PEEK(J): NEXT and press RETURN. These eight values define the first character, which is @, as follows:

PEEK Value		Bit Equivalent Which Defines the Character
28	($1C)	0 0 0 1 1 1 0 0
34	($22)	0 0 1 0 0 0 1 0
74	($4A)	0 1 0 0 1 0 1 0
86	($56)	0 1 0 1 0 1 1 0
76	($4C)	0 1 0 0 1 1 0 0
32	($20)	0 0 1 0 0 0 0 0
30	($1E)	0 0 0 1 1 1 1 0
0	($0)	0 0 0 0 0 0 0 0

Other areas in the character ROM give similar results. The eight bytes from 33792, for example, correspond to reverse-@ and have exactly the opposite bit pattern. In other words, dots which were light with @ are dark with reverse-@, and vice versa.

Program 12-1 allows the bit patterns of any consecutive eight bytes to be displayed at the top of the screen. After loading and running it, SYS (828) 8*4096 displays the eight bytes starting at $8000, representing the @ symbol, in either spaces or solid squares. The symbols + or − step forward or back by eight bytes, and 1 moves up one byte. This is useful in examining ROM software to find where the graphics definitions are stored. Two additional commands, F and B, step forward or back by 256 bytes to allow faster movement through memory.

Program 12-1. Eight-Byte Graphics Display, with Addresses

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
Ø DATA 32,138,2Ø5,32,247,215,169,147,44,169,19,32,
  21Ø,255,166,2Ø,165,21,32,2Ø5,221         :rem 189
1 DATA 169,13,32,21Ø,255,16Ø,Ø,132,253,162,Ø,177,2
  Ø,133,254,169,18,6,254,176,2,169          :rem 188
2 DATA 146,32,21Ø,255,169,32,32,21Ø,255,232,224,8,
  2Ø8,235,169,13,32,21Ø,255,23Ø,253         :rem 23Ø
3 DATA 164,253,192,8,2Ø8,216,32,225,255,2Ø8,1,96,3
  2,228,255,24Ø,251,2Ø1,7Ø,2Ø8,2,23Ø         :rem 31
4 DATA 21,2Ø1,66,2Ø8,2,198,21,2Ø1,49,2Ø8,6,23Ø,2Ø,
  2Ø8,2,23Ø,21,2Ø1,43,2Ø8,11,24,165         :rem 2Ø4
5 DATA 2Ø,1Ø5,8,133,2Ø,144,2,23Ø,21,2Ø1,45,2Ø8,11,
  56,165,2Ø,233,8,133,2Ø,176,2,198          :rem 159
6 DATA 21,16,13Ø,48,128                     :rem 1Ø1
1Ø FOR J=828 TO 964: READ X: POKE J,X: NEXT:rem 21
2Ø PRINT "{CLR}EXAMPLE:                     :rem 234
3Ø PRINT "SYS (828) 32768                    :rem 14
4Ø PRINT "THEN +,- MOVE 8 BYTES,            :rem 232
5Ø PRINT "F,B MOVE 256 BYTES,                :rem 79
6Ø PRINT "1 MOVES 1 BYTE.                    :rem 99
```

Since the total amount of memory dedicated to character definitions is 4K, there is room for 4096/8 = 512 characters. However, because the screen memory area can only store ordinary bytes, it is only possible to display 256 characters at one time. Thus there are two distinct sets of characters. SHIFT-Commodore switches between them by directing the VIC chip to point either at $8000 (which is the value set when the machine is turned on and is the uppercase mode) or at $8800 (lowercase mode). Lowercase mode can be selected by PRINT CHR$(14), and uppercase can be selected by PRINT CHR$(142).

The two sets are identical to those in Commodore's other machines. The idea is that one could be used for word-processing applications (where the distinction between capital and lowercase letters is important), while the other could be used for pictorial applications (for example, with the playing-card symbols). Because of this they are often called "text" and "graphics" modes.

Program 12-2, for the unexpanded VIC, displays all 256 characters of either mode in black at the top half of the screen. Press SHIFT-Commodore to change modes; note how many of the characters are duplicated.

Program 12-2. VIC Character Sets

```
10 FOR J=7680 TO 7680+255:REM 256 VALUES NEED 256
   {SPACE}SCREEN LOCATIONS
20 POKE J,Q:REM POKE 0,1,2,3,...,255 AS Q INCREASE
   S
30 POKE 38400+Q,0:REM COLOR RAM FROM $9600 SET TO
   {SPACE}BLACK
40 Q=Q+1
50 NEXT
```

A table of these characters is given in Appendix J. Except for space and SHIFT-space, which PEEK as 32 and 96, there is no ambiguity about these screen characters.

However, there is a rather confusing distinction between characters as they are POKEd into the screen by this program and as they are printed. PRINT translates many characters in special ways, changing color, clearing the screen, moving the cursor up or down or to the start of the next line, and so on. Some of them, like RETURN, are fairly standard, while others are peculiar to the VIC-20 and 64.

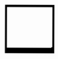
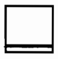
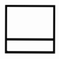
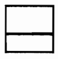
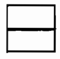
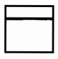
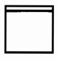
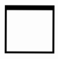
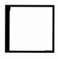
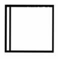
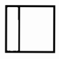
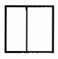
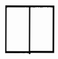
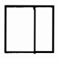
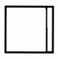
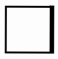
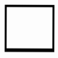
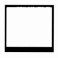
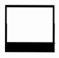
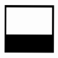
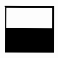
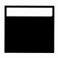
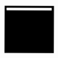
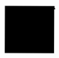
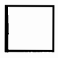
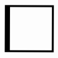
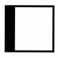
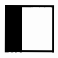
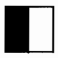
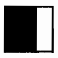
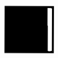
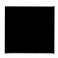
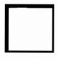
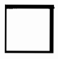
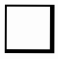
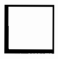
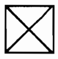
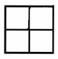
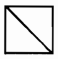
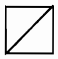
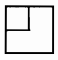
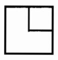
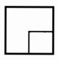
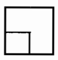
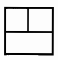
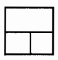
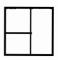
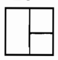
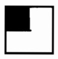
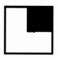
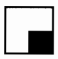
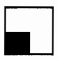
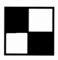
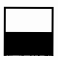
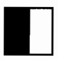
Another appendix, Appendix I, lists PRINTed characters as they are implemented in the VIC-20, in effect listing CHR$ values and their corresponding ASC values. True ASCII reserves the first 32 characters for control information, and Commodore has borrowed this idea. The VIC-20's characters as printed are closer to true ASCII than is the case in earlier CBM machines; the output routine has been modified to insure this, so conversion to true ASCII is easier. However, the upper- and lowercase alphabets are interchanged in relation to true ASCII, as the table in Appendix I shows.

The RVS (reverse) feature allows PRINT to match any of the 256 screen characters, and it effectively doubles the character set available. In fact, the built-in reverse characters are arranged in step with the unreversed equivalents, and the VIC chip points to them when bit 7 is set. Because of this, an easy way to reverse characters is to set bit 7 on (or, in BASIC terms, add 128). Try POKE 7680,128, for example.

The fact that this flag (or RVS) is necessary to print a complete graphics set can be irritating. Suppose you have laboriously designed a large graphic display on the screen. It is impossible to save reverse characters as strings by homing the cursor and typing RETURNs. The strings have to be punctuated with (RVS ON) and (RVS OFF). Block saving of screen memory is usually best; see Chapter 6.

There is no simple translation between unSHIFTed and SHIFTed keys because of rearrangements of the keyboard, but setting bit 6 "on" usually displays the SHIFTed version on the screen. In BASIC terms, add 64. Try POKE 7680,1 and POKE 7680,65 in lowercase mode.

Table 12-1. Quick Cross Reference to VIC Graphics

KEY:	C-@	SH-R	SH-F	SH-*	SH-C	SH-D	SH-E	C-T
CHR$:	228	210	198	192	195	196	197	227
POKE:	100	82	70	64	67	68	69	99
								
KEY:	C-G	SH-T	SH-G	SH-B	SH--	SH-H	SH-Y	C-M
CHR$:	229	212	199	194	221	200	217	231
POKE:	101	84	71	66	93	72	89	103
								
KEY:					REVERSE, CHR$(18), THEN-			
KEY:	C-@	C-P	C-O	C-I	C-U	C-Y	C-T	SH-space
CHR$:	228	239	249	226	184	183	163	160
POKE:	100	111	121	98	248	247	227	224
								
KEY:					REVERSE, CHR$(18), THEN-			
KEY:	C-G	C-H	C-J	C-K	C-L	C-N	C-M	SH-space
CHR$:	229	244	245	225	182	170	167	160
POKE:	101	116	117	97	246	234	231	224
								
KEY:	SH-O	SH-P	SH-@	SH-L	SH-V	SH-+	SH-M	SH-N
CHR$:	207	208	186	204	214	219	205	206
POKE:	79	80	122	76	86	91	77	78
								
KEY:	C-X	C-Z	C-A	C-S	C-E	C-R	C-W	C-Q
CHR$:	189	173	176	174	177	178	179	171
POKE:	125	109	112	110	113	114	115	107
								
KEY:	C-V	C-C	C-D	C-F	C-B	C-I	C-K	
CHR$:	190	188	172	187	191	226	225	
POKE:	126	124	108	123	127	98	97	
								

KEY:	SH-K	SH-J	SH-U	SH-I	SH-W	SH-Q
CHR$:	203	202	213	201	215	209
POKE:	75	74	85	73	87	81
	╯	╰	╭	╮	○	●
KEY:	SH-£	C-*	C-+	C-£	C-—	
CHR$:	169	223	166	168	220	
POKE:	105	95	102	104	92	
	◤	◥	▩	▄	▌	
KEY:	SH-A	SH-S	SH-Z	SH-X	SH-↑	
CHR$:	193	211	218	216	222	
POKE:	65	83	90	88	94	
	♠	♥	♦	♣	π	

Notes:

1. C- means press the Commodore key and the indicated character; SH- means press the SHIFT key and the indicated character.
2. There are ambiguities in many of the CHR$ figures—CHR$(227) or CHR$(163) for example might equally well be chosen. I've preferred the values with a constant difference of 64 or 128 from the screen POKE/PEEK value.
3. As the characters are made of 8 x 8 dots, a line cannot appear exactly in the center of any character; some characters, when positioned as neighbors, will not exactly line up together.
4. In lowercase mode, some characters aren't available; those with POKE values 65-90 appear as A-Z. The full 128 graphics characters are obtained by reversing all those in the table, whether by PRINTing the reverse character first or by POKEing the values listed here + 128.
5. Four extra graphics, obtainable only in lowercase mode, are:

KEY:	C-↑	C-*	SH-£	SH-@
CHR$:	126	223	169	186
POKE:	94	95	105	122
	▦	▧	▨	✓

Table 12-1 gives you a quick cross-reference to VIC graphics characters. Note that the pairs of symbols on the near side of most keys apply only in uppercase/ graphics mode, the default mode when the machine is turned on. After SHIFT-Commodore puts the machine into lowercase, only the left-hand graphics symbols can appear on the screen, and SHIFTing a key gives the capital version (except for a few keys with no SHIFTed version, like @, *, and £). Thus, the right-hand set of graphics is lost in lowercase mode.

Fortunately, however, some some very useful graphics are retained. For example, neat boxes can be ruled on the screen, using Commodore-A, Commodore-S, Commodore-Z, Commodore-X, SHIFT-*, and SHIFT-—. The toggle effect between these modes can be turned off by PRINT CHR$(8) and reenabled with PRINT CHR$(9), or with POKE 657,128 and POKE 657,0, which sets the relevant flag. Sometimes, programs with user-defined graphics do not disable this switch, in which case SHIFT-Commodore can produce odd results as graphics characters are looked for in a region $800 bytes away from that intended by the programmer.

Astute readers will note that some symbols are missing from the keys. Thirty-one keys have a pair of symbols, making 62 symbols in all. Adding π and SHIFT-space gives 64 graphics characters. But four new characters, accessible only in lowercase mode, also exist. These symbols are Commodore-up-arrow (a 4 × 4 checkerboard), Commodore-* (four downward-sloping diagonal lines), SHIFT-£ (four upward-sloping diagonal lines), and SHIFT-@ (square root or check mark).

The control key selects color and reverse characters and has no function apart from slowing screen scroll so LIST is more readable. It is programmable; try running 1 PRINT PEEK(653): GOTO 1. Press SHIFT, Commodore, and CTRL singly or together. You will see that each is individually detected in location 653.

PRINTing with BASIC Graphics

PRINTing is certainly the easiest way to produce graphics effects. First, though, you should see what PRINT actually does.

PRINT has to interpret the information it's given and convert it into POKEs to the correct parts of the screen and color RAM, perhaps also selecting lower- or uppercase mode. This is a complicated and relatively slow process. It uses temporary locations to store the current color (646 = $0286), the status of the reverse flag (199 = $C7), and the position on the screen to which the character is to be printed (row is 214 ($D6), column 211 ($D3)), among other things. Try POKE 646,7. Everything now prints in yellow, as though you'd typed CTRL-YELLOW. POKE 199,1:PRINT "HELLO" prints HELLO in reverse. The reverse flag is turned off with RETURN.

PRINT uses the Kernal output routine $FFD2 to put characters on the screen. Try POKE 780,65: SYS 65490. This uses $FFD2 and has the same effect as PRINT CHR$(A). ML programmers can trace the output to $E742. From there, ASCII characters above 128 go to $E800, while those from 0 to 127 are processed from $E756. Comparison instructions look for RETURN, space, SHIFT-space, and so on, and characters in quotes are output from $E6CB. The actual routine which POKEs the character is at $EAA1. The accumulator holds the character, and the X register holds its color code. Unlike some CBM machines, the VIC offers no way to increase the speed of PRINT with a simple POKE.

Program Examples Using PRINT

Programmers unused to the graphics set, or who are looking for new graphics ideas, can experiment with Program 12-3. It fills the screen with whatever string is entered.

Program 12-3. Graphics with Strings

```
10 INPUT "GRAPHICS";G$
20 FOR J=1 TO 500/LEN(G$):PRINT G$;:NEXT
30 PRINT:GOTO10
```

Try, for example, Commodore-* and SHIFT-£, or Commodore-A, Commodore-S, Commodore-Z, and Commodore-X, or other combinations of similar characters.

Program 12-4 fills the screen with a repetitive design based on three shapes which are designed to match, like tiles, when put next to each other (at least as far as the character set allows).

Program 12-4. Graphics with Shapes

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 A$="N{DOWN}N{UP}":B$="{DOWN}M{UP}M":C$="@{DOWN}
   P{UP}"                                          :rem 48
20 L=RND(1)*8+1                                    :rem 15
30 FOR J=1 TO L                                   :rem 241
35 R=RND(1)                                        :rem 93
40 IF R<.33 THEN X$=X$+A$                           :rem 9
50 IF .33<R AND R<.67 THEN X$=X$+B$                 :rem 7
60 IF .67<R THEN X$=X$+C$                          :rem 20
70 NEXT                                           :rem 166
100 FOR J=1 TO 300/L:PRINT X$;:NEXT               :rem 216
110 GET X$:IFX$="" GOTO 110                       :rem 127
120 RUN                                           :rem 136
```

Program 12-5 shows how strings can be overprinted to produce the effect of movement. It gives left-to-right scrolling. Note the slight delay when a color character is printed.

Program 12-5. Overprinting Strings

```
10 A$="{RVS}AND {PUR}NOW {RED}FOR SOMETHING"
20 FOR J=1 TO LEN(A$)
30 PRINT RIGHT$(A$,J)
40 PRINT"{UP}";
50 FOR K= 1 TO 200:NEXT
60 NEXT:PRINT
```

Frog-Style Graphics

Program 12-6 combines four separate sets of graphics characters, which alternately move in opposite directions across the screen. Note in line 150 that the HOME character gives a fixed reference point at the top left of the screen. From this reference, the position at which to print is calculated by moving the cursor down and across, leaving the remaining graphics untouched.

Program 12-6. Frog-Style Graphics

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
5 PRINT"{CLR}{WHT}":POKE 36879,110              :rem 214
20 A$ = "A{5 SPACES}S{5 SPACES}Z{5 SPACES}X
   {5 SPACES}"                                  :rem 142
30 A$ = A$+A$+A$+A$                              :rem 26
100 FOR I= 1 TO 22                               :rem 54
110 D1$ = MID$(A$,I,22)                          :rem 33
120 D2$ = MID$(A$,44-I,22)                      :rem 184
130 D3$ = MID$(A$,21+I,22)                      :rem 179
140 D4$ = MID$(A$,55-I,22)                      :rem 190
```

```
150 PRINT "{HOME}" +D1$ + "{HOME}{2 DOWN}" + D2$ +
     "{HOME}{4 DOWN}" + D3$ + "{HOME}{6 DOWN}" + D
    4$                                        :rem 226
160 NEXT                                      :rem 214
170 GOTO 100                                   :rem 98
```

Printing multiple cursor movements, as you can see, is slow. A cursor-positioning subroutine is faster. HTAB & VTAB (see Chapter 6) groups together several useful subroutines to position the cursor on the screen. The machine code version is extremely fast and can greatly improve the appearance of programs which print to the screen.

Kaleidoscope

Program 12-7 displays random colors. The colors are reflected, using string arrays, to give an attractive symmetry.

Program 12-7. Kaleidoscope

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
20 POKE 36879,8:REM BLACK                     :rem 129
30 PRINT"{CLR}"                               :rem 199
40 DIM M$(20,10),A$(10),RN$(20)               :rem 213
50 CO$="{RED}{CYN}{PUR}{GRN}{BLU}{YEL}"       :rem 206
60 FORI=0 TO 20: RN$(I) = MID$(CO$,RND(1)*6+1,1) +
   "{RVS} ": NEXT                             :rem 135
70 FOR J=0 TO 10: FOR I=0 TO J: C$=RN$(J-I)
                                              :rem 128
80 M$(I,J)=C$                                 :rem 141
90 M$(J,I)=C$                                 :rem 142
100 M$(20-J,I)=C$                              :rem 69
110 M$(20-I,J)=C$                              :rem 70
120 NEXT:NEXT                                  :rem 75
130 PRINT"{HOME}"                             :rem 120
140 FOR J=0 TO 10: A$(J)="":FOR I= 0 TO 20:rem 206
150 A$(J)=A$(J) + M$(I,J)                     :rem 127
160 NEXT: NEXT                                 :rem 79
170 FOR J=10 TO 0 STEP-1: PRINT A$(J): NEXT:rem 20
180 FOR J=0 TO 10: PRINT A$(J):NEXT           :rem 123
190 CLR:GOTO 40                                :rem 82
```

Because of the large number of individual strings, and despite the fact that most are very short, there is a problem with garbage collection (see Chapter 6). An unexpanded VIC takes about 30 seconds to generate a picture; with 8K expansion this drops to 5 seconds.

Experiments with POKEs

PRINTing to the screen puts the VIC through some tortuous calculations. Is POKEing faster? Usually not. In BASIC, POKE is not a fast command. BASIC POKEs spend a lot of time in calculations, so there is no great speed advantage. Color RAM also must be POKEd, unless the screen background color has been selected to make this unnecessary, or the color is already acceptable.

POKE has some advantages over PRINT. For instance, when you're using POKE, any part of the screen can be changed without the need to keep track of the cursor position. This is why the screen editor programs later in this chapter use POKE. POKEing can also handle the situation where the screen is redefined to have more (or less) than the usual 22 columns across; PRINT can't allow for this automatically.

PRINT is also liable to produce unwanted effects, like scrolling the screen when a character is printed in the lower right corner. POKEing has none of these side effects. In addition, the entire character set is available with POKE, without the need to use RVS.

To POKE to the screen, you must of course know where the screen is. This will be discussed in detail later on; for now, a summary will do. Color RAM matches the screen positions, and Tables 12-2 and 12-3 are handy guides to these addresses. Try POKE 8164,1: POKE 38884,0 with the unexpanded VIC to plot a black A at the start of line 22. Experiment with different values to see how color RAM works. It is equally essential to know what value to POKE, so Appendix J has a table of the 256 characters of either lowercase or uppercase mode available by POKEing.

Table 12-2. Screen Memory Locations (Decimal)

Screen Line Number	Unexpanded VIC VIC with 3K		VIC with 8K VIC with 16K	
	Screen	Color	Screen	Color
0	7680	38400	4096	37888
1	7702	38422	4118	37910
2	7724	38444	4140	37932
3	7746	38466	4162	37954
4	7768	38488	4184	37976
5	7790	38510	4206	37998
6	7812	38532	4228	38020
7	7834	38554	4250	38042
8	7856	38576	4272	38064
9	7878	38598	4294	38086
10	7900	38620	4316	38108
11	7922	38642	4338	38130
12	7944	38664	4360	38152
13	7966	38686	4382	38174
14	7988	38708	4404	38196
15	8010	38730	4426	38218
16	8032	38752	4448	38240
17	8054	38774	4470	38262
18	8076	38796	4492	38284
19	8098	38818	4514	38306
20	8120	38840	4536	38328
21	8142	38862	4558	38350
22	8164	38884	4580	38372

Table 12-3. Screen Memory Locations (Hexadecimal)

Screen Line Number	Unexpanded VIC and VIC with 3K Expansion		VIC with 8K Expansion and VIC with 16K Expansion	
	Screen	Color	Screen	Color
0	$1E00	$9600	$1000	$9400
1	$1E16	$9616	$1016	$9416
2	$1E2C	$962C	$102C	$942C
3	$1E42	$9642	$1042	$9442
4	$1E58	$9658	$1058	$9458
5	$1E6E	$966E	$106E	$946E
6	$1E84	$9684	$1084	$9484
7	$1E9A	$969A	$109A	$949A
8	$1EB0	$96B0	$10B0	$94B0
9	$1EC6	$96C6	$10C6	$94C6
10	$1EDC	$96DC	$10DC	$94DC
11	$1EF2	$96F2	$10F2	$94F2
12	$1F08	$9708	$1108	$9508
13	$1F1E	$971E	$111E	$951E
14	$1F34	$9734	$1134	$9534
15	$1F4A	$974A	$114A	$954A
16	$1F60	$9760	$1160	$9560
17	$1F76	$9776	$1176	$9576
18	$1F8C	$978C	$118C	$958C
19	$1FA2	$97A2	$11A2	$95A2
20	$1FB8	$97B8	$11B8	$95B8
21	$1FCE	$97CE	$11CE	$95CE
22	$1FE4	$97E4	$11E4	$95E4

The unexpanded VIC and the VIC with 3K expansion start their screen at $1E00 (7680) and their color RAM at $9600 (38400). A VIC with 8K or 16K expansion has its screen at $1000 (4096) and its color at $9400 (37888).

To see how this works, look at Program 12-8. It puts solid squares of random color onto the normal VIC screen; it is set up (in line 10) for the unexpanded VIC.

Program 12-8. Multicolored Squares

```
10 SC=7680:CO=38400
15 PRINT "{CLR}":POKE CO-1521,25
20 R=RND(1)*506
30 POKE SC+R,160
40 POKE CO+R,RND(1)*8
50 GOTO 20
```

Finding the offset from the start of the screen is simple if you take some care in numbering. It is easiest to start at zero, so the horizontal position is 0–21 and the vertical position is 0–22. The offset is then 22 times vertical position plus horizontal position.

Program 12-9 is a subroutine to POKE character X in color C at position H across, V down, for any VIC.

Program 12-9. POKEing Character X in Color C at Position H

```
30000 POKE SC+22*V+H,X
30010 POKE CO+22*V+H,C
30020 RETURN
```

Try Program 12-10, along with the subroutine given in Program 12-9.

Program 12-10. Using the Color POKE Subroutine

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 SC=7680:CO=38400:C=2:X=127:PRINT"{CLR}"  :rem 4
20 FOR H =1 TO 20                             :rem 4
30 V=1: GOSUB 30000                          :rem 212
40 V=21: GOSUB 30000                           :rem 7
50 NEXT                                      :rem 164
60 FOR V =2 TO 21                             :rem 24
70 H=1: GOSUB 30000                          :rem 202
80 H=20: GOSUB 30000                         :rem 252
90 NEXT                                      :rem 168
100 GET R$:IF R$="" THEN 100                 :rem 103
110 END                                      :rem 105
```

Another example, Program 12-11, draws a maze. This maze is "simply connected"—that is, it is basically a contorted tube with no isolated islands within it.

Program 12-11. Maze

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 A(0)=-2: A(1)=-44: A(2)=2:A(3)=44         :rem 159
20 SC=256*PEEK(648):A=SC+45+44*INT(10*RND(1))+2*IN
   T(10*RND(1))                              :rem 148
30 PRINT "{CLR}": FOR J=1 TO 21:PRINT"{RVS}
   {21 SPACES}":NEXT                          :rem 60
100 POKE A,4                                  :rem 97
110 J=INT(RND(1)*4): X=J                      :rem 50
112 IF S>SMAX THEN SMAX=S: FIN=B             :rem 123
120 B=A+A(J): IF PEEK(B)=160 THEN POKE B,J:POKE A+
    A(J)/2,32:A=B:S=S+1:GOTO 110             :rem 224
130 J=(J+1) AND 3: IF J<>X THEN GOTO120      :rem 248
140 J=PEEK(A): POKE A,32: S=S-1: IF J<4 THEN A=A-A
    (J):GOTO 110                               :rem 8
150 POKE A,1: POKE FIN,2                       :rem 7
160 GOTO160                                  :rem 103
```

The algorithm uses space characters to mark boundaries, so there's an unused border of space characters. This version selects a random starting point, and on finishing, POKEs A and B into the two points furthest removed from each other in the maze.

Any VIC will run this program successfully. Conversion to ML is needed to make it run faster; white-on-white plotting followed by color RAM POKEs is necessary if you want the plotting process to be invisible.

Machine Language Subroutines

BASIC is liable to be slow when dealing with graphics. In this section you will look at typical machine language methods, which are much faster. All the examples can be run by inexperienced programmers; knowledge of machine language is not necessary.

Printing Characters to the Screen

$FFD2, the Kernal's output routine, behaves like PRINT (except that it's your responsibility to store the characters you want printed in RAM). The speed advantage is considerable; the price to be paid is the need to organize memory.

Type in Program 12-12, then type SYS 828. The word HELLO appears, in reversed red characters. Line 40 holds the ASC characters for reverse, red, and the letters of HELLO; it also has a zero byte at the end, which is used to signal the end of the string. If line 40 is modified and the program run, SYS 828 will give other results.

The machine language isn't relocatable, because the position of the table varies if it is moved into other positions, but the only adjustment needed is to alter 74 and 3 in line 30, which point to $034A, to reflect the new table. Any free RAM area can hold the ML.

The disassembly of the ML is given below.

```
033C  LDX  #$00
033E  LDA  $034A,X  ;LOAD Xth CHARACTER FROM TABLE
0341  BEQ  $0349    ;EXIT IF ZERO
0343  JSR  $FFD2    ;OUTPUT CHARACTER
0346  INX           ;INCREASE COUNTER BY 1
0347  BNE  $033E    ;CONTINUE LOOP
0349  RTS
```

Although this example uses only straightforward lettering, the technique can obviously be extended to include cursor moves so a 3 × 3 block of characters can be printed by a SYS call.

Program 12-12. ML Printer

```
10 FOR J=828 TO 849
20 READ X:POKE J,X:NEXT
30 DATA 162,0,189,74,3,240,6,32,210,255,232,208,24
   5,96
40 DATA 18,28,72,69,76,76,79,0
```

POKEing the Entire Character Set to the Screen

The following program POKEs an ML routine into the cassette buffer:

Program 12-13. POKE to Screen

```
10 FOR I=828 TO 843:READ A:POKE I,A:NEXT
20 DATA 160,0,162,0,138,157,0,30
30 DATA 152,157,0,150,232,208,245,96
```

SYS 828 puts 256 black characters on the top half of the screen. The second value (0) controls the color; POKE 829,2, for example, gives red characters on typing in SYS 828.

The addresses of the screen and of color RAM apply to the unexpanded VIC-20. Note that the routine is relocatable; it can be put in any free area of RAM and still operate correctly with a SYS call to its first address.

The disassembled ML is given below:

```
033C  LDY  #$00
033E  LDX  #$00
0340  TXA
0341  STA  $1E00,X  ;X TAKES VALUES 0,1,2,...FF
0344  TYA
0345  STA  $9600,X  ;COLOR IS DETERMINED BY Y [BLACK HERE]
0348  INX
0349  BNE  $0340
034B  RTS
```

Reversing Part of the Screen

This effect, useful in highlighting parts of the screen or making them flash (by repeating the reversal several times), is easy to produce. The high bit of screen RAM characters determines whether or not the character is reversed; therefore, all you need to do is replace each character by its equivalent with the high bit reversed.

In machine language you can do this with LDA (from an address), EOR #$80 (to switch the high bit), and STA (back into address).

This version is relocatable into any secure area of RAM, for example, 828. Then SYS 828,7680,22 will reverse 22 characters starting at 7680, the topmost line of an unexpanded VIC screen.

Program 12-14. Reverse Characters on the Screen

```
10 FOR I=828 TO 854:READ A:POKE I,A:NEXT
20 DATA 32,178,209,165,100,133,253,165
30 DATA 101,133,252,32,155,215,138,168
40 DATA 136,177,252,73,128,145,252,136
50 DATA 16,247,96
```

Other analogous effects include reversing all characters, with ORA #$80, and unreversing all characters with AND #$7F. Flashing the whole screen is more easily done by altering the color rather than the characters; a couple of POKEs are all that's required.

The disassembled ML is given below:

```
033C  JSR   $D1B2  ;INPUT START ADDRESS
```

```
033F  LDA  $64
0341  STA  $FD
0343  LDA  $65
0345  STA  $FC      ;(FC) IS THE STARTING ADDRESS WITHIN THE SCREEN
0347  JSR  $D79B    ;INPUT NUMBER OF CHARACTERS PARAMETER
034A  TXA
034B  TAY           ;Y IS THE NUMBER OF CHARACTERS TO BE REVERSED
034C  DEY
034D  LDA  ($FC),Y
034F  EOR  #$80
0351  STA  ($FC),Y
0353  DEY
0354  BPL  $034D
0356  RTS
```

Plotting Rows or Columns to High Accuracy

Table 12-1, "Cross-Reference to VIC Graphics," groups similar graphics characters together; from the layout it is clear that the completeness of the graphics sets enables some progress to be made towards accurate graphics. To show the approach, we'll write a routine which plots vertical columns on the screen, to the nearest one-eighth of a square, that is, including 0–7 rows of dots on top of each column of solid characters.

Program 12-15. Histogram

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
Ø FOR J=828 TO 924                                :rem 13Ø
1 READ X: POKE J,X:NEXT                            :rem 85
1Ø DATA 32,178,2Ø9,165,1ØØ,133,252,165,1Ø1,133,251
   ,32,155,215,134                                :rem 1Ø9
2Ø DATA 143,32,155,215,138,16Ø,Ø,2Ø1,8,144,39,233,
   8,72,169,16Ø                                   :rem 227
3Ø DATA 145,251,165,251,133,253,165,252,41,3,9,148
   ,133,254,165                                   :rem 237
4Ø DATA 143,145,253,165,251,233,22,133,251,165,252
   ,233,Ø,133,252                                  :rem 62
5Ø DATA 1Ø4,76,82,3,17Ø,24Ø,21,189,149,3,145,251,1
   65,251,133,253                                  :rem 77
6Ø DATA 165,252,41,3,9,148,133,254,165,143,145,253
   ,96,1ØØ                                        :rem 249
7Ø DATA 111,121,98,248,247,227,16Ø                :rem 142
```

To make the routine easy to use, three parameters are input as part of the SYS call. The syntax is:

SYS 828,bottom of column,color,height

when the routine starts at 828. For example, try this,

Program 12-16. Histogram Example

```
10 FOR J=1 TO 20
20 SYS828,8164+J,2,J
30 NEXT
```

Graphics of this type are convenient because they coexist with ordinary text and graphics—there's no problem mixing text on the screen with them, and there are none of the complexities inevitable when defining special characters in RAM.

Double-Density Graphics

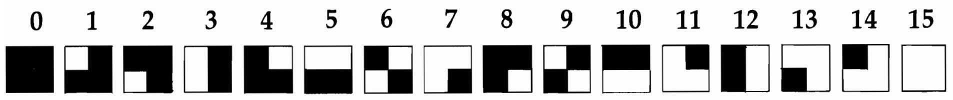
"Double-Density Plotting" is designed to fit into the top of the unexpanded VIC-20's memory; it can be moved to other areas of RAM if the pointer to the 16-byte table at the end, 226 and 29 in line 7, is modified. It exploits the fact that all 16 combinations of squares with internal quadrants exist in the VIC graphics set.

Program 12-17. Double-Density Plotting

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
0 DATA 32,155,215,134,254,32,155,215,134,255,32,15
  5,215,134,252,32,155                      :rem 106
1 DATA 215,134,251,165,254,48,89,201,44,176,85,165
  ,255,48,81,201,46,176                     :rem 193
2 DATA 77,169,0,133,253,169,46,229,255,70,254,38,2
  53,106,38,253,133,255                     :rem 192
3 DATA 166,253,169,0,133,253,133,210,56,38,253,202
  ,16,251,165,255,10,10                     :rem 152
4 DATA 10,10,38,210,133,209,165,255,10,101,255,10,
  144,2,230,210,24,101,209                    :rem 6
5 DATA 133,209,173,136,2,101,210,133,210,164,254,1
  77,209,162,15,221,226                     :rem 149
6 DATA 29,240,4,202,16,248,96,165,252,240,6,138,5,
  253,170,208,8,138,73,255                   :rem 73
7 DATA 5,253,73,255,170,189,226,29,145,209,32,178,
  234,165,251,145,243,96                    :rem 250
8 DATA 32,126,123,97,124,226,255,236,108,127,98,25
  2,225,251,254,160                         :rem 242
10 POKE56,29: POKE 55,80: CLR                :rem 20
20 FOR J=7505 TO 7665: READ X: POKE J,X: NEXT
                                            :rem 122
30 PRINT "{CLR}SYS 7505,H,V,ON/OFF,COLOR.   :rem 23
40 PRINT "{DOWN}EXAMPLE: SYS 7505,3,10,1,2:rem 225
50 PRINT "SETS A POINT AT 3 ACROSS, 10 UP, IN RED.
                                            :rem 175
60 SYS 7505,3,10,1,2                        :rem 221
```

Figure 12-1. 16 Quadrant Pictures



Since each quadrant may be either filled or not filled, there are 16 possible combinations ($2^4=16$), from completely blank to completely filled. The machine language disassembly is too long for inclusion here. Its method involves:

- Calculating the position in screen RAM corresponding to the horizontal and vertical coordinates.
- Checking this screen character—if it is not in the table, do nothing else.
- Adding or subtracting the appropriate quadrant and replacing this new character into the screen.

VIC's screen in effect now has a resolution of 44 × 46 small squares. Numbering is 0–43 horizontally and 0–45 vertically, starting at the bottom left. Like the previous program, the fact that the characters are ordinary graphics POKEd into the screen means that mixing text with the small square graphics is not a problem.

The syntax is SYS 7505,H,V,FLAG,COLOR. H is the horizontal position, V the vertical. If FLAG is 1, a small square is plotted; if 0, erased. The COLOR parameter allows the color (0–7) to be selected, subject to the limitation that only one color can be selected within a single character. The program can be used with a light pen.

The first demonstration program is called "Lissajous Figures." It creates figures similar to shapes obtained when two perpendicular pendulums swing together. The second draws centrally symmetrical patterns. Remember, both of these demonstrations require that the machine language program from Double-Density Plotting be in memory.

Program 12-18. Lissajous Figures Demonstration

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
Ø REM TRY INPUTTING 3,5                         :rem 1Ø5
1Ø POKE 36879,4Ø                                 :rem 49
2Ø INPUT A,B                                    :rem 161
3Ø PRINT "{CLR}"                                :rem 199
4Ø FOR J=Ø TO 9E9 STEP .Ø1                       :rem 39
5Ø X=(1+SIN(A*J))*22: Y=(1+COS(B*J))*23          :rem 29
6Ø SYS 75Ø5,X,Y,1,Ø                             :rem 248
7Ø NEXT                                         :rem 166
```

Program 12-19. Flower Design

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
1Ø INPUT M                                       :rem 62
2Ø PRINT"{CLR}"                                 :rem 198
3Ø FOR TH=Ø TO 1ØØ STEP .1                       :rem 34
4Ø S=SIN(TH*M)+.2                               :rem 2Ø5
5Ø X=S*COS(TH):Y=S*SIN(TH)                      :rem 1Ø9
6Ø X=22+17*X:Y=23+17*Y                          :rem 191
7Ø SYS 75Ø5,X,Y,1,2                             :rem 251
8Ø NEXT                                         :rem 167
```

Altering Color RAM with Machine Language

The routine to clear the screen is located in BASIC ROM at $E55F (try SYS58719). It resets the screen line link tables, and clears each line using a subroutine at $EABD where the X register holds the line's number. What actually happens is that the screen area is filled with 32's (space characters), and the color RAM fills with 1's (white). Since the screen has been filled with spaces, it actually isn't necessary to change color RAM because only the background color shows up.

The consequence is that the cleared screen cannot normally be POKEd with characters, since they are printed white on white and are invisible. PRINTing always sets color as it prints, so this is no problem, but POKEs will be visible only if the background color is set to a contrasting color, if the color RAM is POKEd along with the character, or if color RAM is set to a contrasting color. This last option is best done in machine language—500 or so POKEs can be quite slow. Program 12-20 performs this function. There's no really easy way to clear the screen to a specified color. White is hardcoded in ROM and can't be changed.

Direct modification of color RAM can be used for other purposes like giving a watery, quivery effect by changing between multicolor modes or between multicolor and high-resolution modes; giving an effect of motion by plotting bars of color in sequence, red, cyan, purple, green, blue, and yellow, then cycling through color RAM; or making portions of the screen disappear and reappear, as their colors are set to match the background.

Program 12-20. Change Color RAM

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
Ø REM POKE 14Ø, Ø-15, THEN{2 SPACES}SYS 828
                                        :rem 195
1 REM THIS CODE CAN BE LOCATED IN ANY VIC-2Ø
                                        :rem 173
1Ø FOR J =828 TO 855: READ X: POKE J,X: NEXT
                                        :rem 2Ø
2Ø POKE 14Ø,Ø: SYS 828: REM DEMO WITH RED :rem 183
3Ø DATA 173,136,2,41,2,9,148,133,143,16Ø,Ø:rem 255
4Ø DATA 132,142,162,2,165,14Ø,145,142,2ØØ :rem 2Ø2
5Ø DATA 2Ø8,251,23Ø,143,2Ø2,2Ø8,246,96     :rem 75
```

Scrolling the Whole Screen

In this section we'll see how to scroll the entire screen left, right, up, or down, moving one character's width or height. Color RAM usually has to be moved to match, so that the characters keep their colors. Landscapes can be scrolled right, stars can be scrolled down the screen, patterns can be scrolled up, or whatever. (Because the movement is in whole characters, scrolling is somewhat jerky. Later we'll see how to get smooth scrolling.)

These routines have to move 480 or so characters to new positions in screen memory, and repeat the process for color. About 480 characters, not 506, because in each direction 22 or 23 characters are lost when the scrolling effect overwrites them. A new row or column of characters has to be printed or POKEd to complete the scroll; alternatively, as in an example below, the displaced characters can be stored and reused, giving an indefinitely repeating scroll effect. POKEing is likely to be easier than PRINTing, since the methods used here manipulate the screen data by ignoring line link tables.

One thousand BASIC PEEKs and POKEs, with calculations, are slow, but repetitive loads and moves are easily written in machine language, so this is a very good practical application for machine language. The examples can be called by a simple SYS command from BASIC and are therefore easy to use. They assume an unexpanded VIC-20.

These BASIC loaders POKE machine language into RAM. Once loaded, each remains (unless overwritten). All of the following four routines are relocatable, so if the start address is altered from 828, they will run correctly, provided all the bytes are POKEd in and SYS calls the correct starting address. (Remember that 828 is the start of the cassette buffer, so machine code in here can't be saved easily to tape.)

Program 12-21. Scroll Down One Line

```
Ø DATA 169,31,133,253,169,227,133,252,169,151,133,
  255,169,227,133,254,16Ø,Ø,177,252
1 DATA 16Ø,22,145,252,16Ø,Ø,177,254,16Ø,22,145,254
  ,165,252,2Ø8,2,198,253,198,252,165
2 DATA 254,2Ø8,2,198,255,198,254,165,253,2Ø1,29,2Ø
  8,218,96
1Ø REM ****{2 SPACES}SYS 828 SCROLLS DOWN *****
2Ø FOR J=828 TO 882: READ X: POKE J,X: NEXT
```

Program 12-22. Scroll Up One Line

```
Ø DATA169,3Ø,133,253,169,Ø,133,252,169,15Ø,133,255
  ,169,Ø,133,254,16Ø,22,177,252,16Ø
1 DATAØ,145,252,16Ø,22,177,254,16Ø,Ø,145,254,23Ø,2
  52,2Ø8,2,23Ø,253,23Ø,254,2Ø8,2
2 DATA23Ø,255,165,252,2Ø1,228,2Ø8,222,165,253,2Ø1,
  31,2Ø8,216,96
1Ø REM **** SYS 828 SCROLLS UP *****
2Ø FOR J=828 TO 884: READ X: POKE J,X: NEXT
```

Program 12-23. Scroll One Column Right

```
Ø DATA 169,3Ø,133,253,169,Ø,133,252,169,15Ø,133,25
  5,169,Ø,133,254,162
1 DATA 22,16Ø,2Ø,177,252,2ØØ,145,252,136,177,254,2
  ØØ,145,254,136,136
2 DATA 16,241,24,165,252,1Ø5,22,133,252,165,253,1Ø
  5,Ø,133,253,165,254
3 DATA 1Ø5,22,133,254,165,255,1Ø5,Ø,133,255,2Ø2,16
  ,211,96
1Ø REM **** SYS 828 SCROLLS RIGHT ****
2Ø FOR J=828 TO 891: READ X: POKE J,X: NEXT
```

Program 12-24. Scroll One Column Left

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
Ø DATA 162,6,189,251,29,157,25Ø,29,189,251,149,157
  ,25Ø,149,232,2Ø8,241                         :rem 143
1 DATA 189,25Ø,3Ø,157,249,3Ø,189,25Ø,15Ø,157,249,1
  5Ø,232,2Ø8,241,96                            :rem 242
1Ø REM **** SYS 828 SCROLLS LEFT ****          :rem 131
2Ø FOR J=828 TO 86Ø: READ X: POKE J,X: NEXT:rem 17
1ØØ REM ** DEMONSTRATION **                      :rem 4
1Ø5 X=1Ø                                       :rem 14Ø
11Ø FOR J=21 TO 22*23 STEP 22: POKE J+768Ø,16Ø :NE
    XT                                         :rem 136
115 R=RND(1)                                   :rem 14Ø
12Ø IF R<.4 THEN X=X+1                         :rem 138
122 IF R>.6 THEN X=X-1                         :rem 146
124 IF X>15 THEN X=15                           :rem 76
126 IF X<Ø THEN X=Ø                            :rem 224
128 H=X*22                                       :rem 6
129 SYS 828                                     :rem 61
13Ø FOR J=21 TO 21+H STEP 22: POKE 384ØØ+J,5: NEXT
                                                :rem 53
14Ø FOR J=43+H TO 5Ø6 STEP 22: POKE 384ØØ+J,6: NEX
    T                                          :rem 115
16Ø GOTO 11Ø                                    :rem 98
```

How Is the Screen Scrolled?

Thanks to the systematic memory mapping of the VIC's screen, scrolling in any direction is quite easy. The diagrams show scrolling in two directions (down and right) on a simplified 4 × 4 screen, labeled to show the effects of each scroll. The cells marked with ? have no definite contents.

Figure 12-2. Scrolling Down

Screen

A	B	C	D
1	2	3	4
E	F	G	H
5	6	7	8

Scroll → Down

?	?	?	?
A	B	C	D
1	2	3	4
E	F	G	H

Screen RAM before and after scrolling down

A	B	C	D	1	2	3	4	E	F	G	H	5	6	7	8

?	?	?	?	A	B	C	D	1	2	3	4	E	F	G	H

Diagram of VIC-like screen scrolling down

Figure 12-3. Scrolling Left

Screen

A	B	C	D
1	2	3	4
E	F	G	H
5	6	7	8

Scroll
←
Left

B	C	D	?
2	3	4	?
F	G	H	?
6	7	8	?

Screen RAM before and after scrolling left

A	B	C	D	1	2	3	4	E	F	G	H	5	6	7	8

B	C	D	?	2	3	4	?	F	G	H	?	6	7	8	?

Diagram of VIC-like screen scrolling left

It is easy to deduce that for the VIC-20, scrolling up or down requires that the entire screenful of characters, excluding a set of 22 at the end, be moved 22 places along. Scrolling right or left, in the simplest case, requires only that every screen character (but one) be shifted one place along. Program 12-24, "Scroll One Column Left," uses this method and is therefore very short. FOR J=1 TO 10000: SYS 828: NEXT moves the screen left, and up, as the screen wraps around to the right. Program 12-23, "Scroll One Column Right," operates by moving 21 characters in each row to the right, leaving the leftmost column unchanged.

The order in which the program moves the characters is important—to scroll down, the characters at the bottom right must be moved first; otherwise, because there is no temporary storage, characters will be overwritten.

Modifications for VICs with 8K or 16K Expanders

It is not difficult to modify these routines to work with the screen at $1000 and color RAM at $9400. The following modifications are all that are necessary:

Scroll Down:	Line 0. Convert 31 to 17 and 151 to 149.
	Line 2. Convert 29 to 15.
Scroll Up:	Line 0. Convert 30 to 16 and 150 to 148.
	Line 2. Convert 31 to 17.
Scroll Right:	Line 0. Convert 30 to 16 and 150 to 148.
Scroll Left:	Substitute these lines:

```
0 DATA 162,6,189,251,15,157,250,15,189,251,147,157,250,147,
     232,208,241
1 DATA 189,250,16,157,249,16,189,250,148,157,249,148,232,
     208,241,96
```

It is easy enough to make other modifications, to leave color RAM unaltered, for example, or to scroll the colors while leaving the graphics.

The VIC Chip: Color, Position, and Dimensions of the TV Display

Color, Color RAM, and Multicolor Mode

Color TVs and monitors display color by electronically causing small patches of color *(phosphors)* to glow on the screen. This is *additive* color; each extra color adds to the brightness, unlike painting and printing, which use subtractive color. The primary colors for additive color are red, green, and blue. All three colors combined equally give white.

The phosphor dots can be seen with a low-power magnifier; typically there are columns of red, green, and blue repeating the full width of the screen. TVs are likely to show some defects—a "red" screen will have some green and blue also. As the color is turned down, TVs have a device to average colors locally, giving shades of gray.

Secondary colors are yellow, blue-green, and magenta; these are known by a variety of names—Commodore's choice is yellow, cyan, and purple. Each is made of about equal amounts of two colors: Yellow is red and green, cyan is green and blue, and purple is blue and red. On the VIC-20, adjacent pairs of keys are "complementary": They add to white or gray. For example, red and cyan contain between them red, green, and blue, and in the right proportion mix to give white. The eight basic colors of the VIC are simply the three primaries, each of which is either on or off, giving eight combinations.

The screen's background also has eight colors, of which six are generated by adding white to the primaries and secondaries. Presumably within the VIC chip, setting bit 7 of $900F causes the TV signal to add extra red, green, and blue output to the signal. Orange and light orange have also been put in. Table 12-4 illustrates the combinations.

There are at least two perceptual effects worth mentioning. One is that the weaker, less saturated colors are influenced by stronger adjacent colors, so that the light background colors are often difficult to identify properly. The other is the advancing effect of red and receding effect of blue, which can be striking with solid blocks of these colors. There are related problems in getting red (or any bright color) to look bright, when compared with white, which has three times as many phosphors lit.

Color RAM

Changing the border and background colors is straightforward. Color RAM is a more difficult concept, but seems natural enough after a time: VIC-20, like other CBM computers, has 256 fully defined characters, which use up all the available bytes, so why not have a complete parallel set of RAM to store colors? As Chapter 5 shows, only the low bits are relevant, so values from 0 to 15 apply; higher bit values are simply ignored. Colors 0–7 are the same as with the border and background. If the high bit is on, with color RAM containing 8–15, the character is displayed in multicolor mode.

Color RAM occupies $9400 to $97FF, a total of 1024 bytes. It is wired so that a screen at $1000 (8K or 16K expansion) has its color at $9400, and a screen at $1E00 (unexpanded or 3K expansion) has its color at $9600.

Table 12-4. Background and Border Colors

	Border Color															
	Reversed Characters								Normal Characters							
	Black	White	Red	Cyan	Purple	Green	Blue	Yellow	Black	White	Red	Cyan	Purple	Green	Blue	Yellow
BACKGROUND COLOR: Black	0 $00	1 $01	2 $02	3 $03	4 $04	5 $05	6 $06	7 $07	8 $08	9 $09	10 $0A	11 $0B	12 $0C	13 $0D	14 $0E	15 $0F
White	16 $10	17 $11	18 $12	19 $13	20 $14	21 $15	22 $16	23 $17	24 $18	25 $19	26 $1A	27 $1B	28 $1C	29 $1D	30 $1E	31 $1F
Red	32 $20	33 $21	34 $22	35 $23	36 $24	37 $25	38 $26	39 $27	40 $28	41 $29	42 $2A	43 $2B	44 $2C	45 $2D	46 $2E	47 $2F
Cyan	48 $30	49 $31	50 $32	51 $33	52 $34	53 $35	54 $36	55 $37	56 $38	57 $39	58 $3A	59 $3B	60 $3C	61 $3D	62 $3E	63 $3F
Purple	64 $40	65 $41	66 $42	67 $43	68 $44	69 $45	70 $46	71 $47	72 $48	73 $49	74 $4A	75 $4B	76 $4C	77 $4D	78 $4E	79 $4F
Green	80 $50	81 $51	82 $52	83 $53	84 $54	85 $55	86 $56	87 $57	88 $58	89 $59	90 $5A	91 $5B	92 $5C	93 $5D	94 $5E	95 $5F
Blue	96 $60	97 $61	98 $62	99 $63	100 $64	101 $65	102 $66	103 $67	104 $68	105 $69	106 $6A	107 $6B	108 $6C	109 $6D	110 $6E	111 $6F
Yellow	112 $70	113 $71	114 $72	115 $73	116 $74	117 $75	118 $76	119 $77	120 $78	121 $79	122 $7A	123 $7B	124 $7C	125 $7D	126 $7E	127 $7F
Orange	128 $80	129 $81	130 $82	131 $83	132 $84	133 $85	134 $86	135 $87	136 $88	137 $89	138 $8A	139 $8B	140 $8C	141 $8D	142 $8E	143 $8F
Lt. Orange	144 $90	145 $91	146 $92	147 $93	148 $94	149 $95	150 $96	151 $97	152 $98	153 $99	154 $9A	155 $9B	156 $9C	157 $9D	158 $9E	159 $9F
Lt. Red	160 $A0	161 $A1	162 $A2	163 $A3	164 $A4	165 $A5	166 $A6	167 $A7	168 $A8	169 $A9	170 $AA	171 $AB	172 $AC	173 $AD	174 $AE	175 $AF
Lt. Cyan	176 $B0	177 $B1	178 $B2	179 $B3	180 $B4	181 $B5	182 $B6	183 $B7	184 $B8	185 $B9	186 $BA	187 $BB	188 $BC	189 $BD	190 $BE	191 $BF
Lt. Purple	192 $C0	193 $C1	194 $C2	195 $C3	196 $C4	197 $C5	198 $C6	199 $C7	200 $C8	201 $C9	202 $CA	203 $CB	204 $CC	205 $CD	206 $CE	207 $CF
Lt. Green	208 $D0	209 $D1	210 $D2	211 $D3	212 $D4	213 $D5	214 $D6	215 $D7	216 $D8	217 $D9	218 $DA	219 $DB	220 $DC	221 $DD	222 $DE	223 $DF
Lt. Blue	224 $E0	225 $E1	226 $E2	227 $E3	228 $E4	229 $E5	230 $E6	231 $E7	232 $E8	233 $E9	234 $EA	235 $EB	236 $EC	237 $ED	238 $EE	239 $EF
Lt. Yellow	240 $F0	241 $F1	242 $F2	243 $F3	244 $F4	245 $F5	246 $F6	247 $F7	248 $F8	249 $F9	250 $FA	251 $FB	252 $FC	253 $FD	254 $FE	255 $FF

To use this table: (1) Given the contents of 36879 ($900E), the corresponding colors can be read from the chart. Thus, a program with this line: 5 POKE 36879,169 sets background to pink, border to white. (2) To find what value must be POKEd into 36879 to achieve same color combination, refer to the intersection of the two colors in the table. Thus, cyan border with cyan background, and normal characters, needs POKE 36879,59.

There is a conversion algorithm which derives the position of a character's color RAM from the screen position. In assembly language it is LDA high byte of screen position/ AND #$03/ ORA #$94. In BASIC, PEEK the high byte of the screen position, AND 3, OR 148. The low bytes of the screen position and color RAM are identical.

This arrangement has the fortunate side effect that larger than normal screens, say of 25 × 25 or 625 characters, will not run out of color RAM, as they would if only 512 bytes had been provided.

Multicolor Mode

We cannot fully explore this mode until we deal with user-defined characters. However, the general idea is fairly easy to grasp. This is another Commodore compromise: in order to get more color onto the screen, resolution is halved. Color RAM is the sole determinant of this mode, so it is possible to mix normal, high-resolution characters (made of individual dots) with multicolor characters (made up of pairs of dots).

To get the feel of this, POKE 646,10 and print a few lines of text. The current color being plotted is 2+8—red, but in multicolor mode. This is easier than POKEing color RAM, and has the same effect. The characters are vaguely recognizable, but in addition to red and white, some cyan and black are mixed in too.

Instead of 8 × 8 characters, we have 4 × 8 characters in which each dot can have one of four colors. These are determined by this table:

00 =	**Background color**	(one of sixteen)
01 =	**Border color**	(one of eight)
10 =	**Character color**	(one of eight)
11 =	**Auxiliary color**	(one of sixteen)

The auxiliary color is stored in the four highest bits of 36878 ($900E), and is set to zero, black, when the VIC is switched on. POKE 36878, C*16 + V puts in any auxiliary color from 0 to 15 and adds the volume of the sound too. So POKE 36878,5*16 sets the auxiliary color to green.

Figure 12-4 shows the effect multicolor mode has on the Commodore-+ character. After POKE 646,10 a few Commodore-+ characters should show alternate lines of red and cyan. POKE 646,11 makes the character color cyan, so Commodore-+ will show as a solid cyan square.

Figure 12-4. Multicolor Example One

Normal:	Multicolor:	Displays As:			
1 0 1 0 1 0 1 0	10 10 10 10	Chartr	Chartr	Chartr	Chartr
0 1 0 1 0 1 0 1	01 01 01 01	Border	Border	Border	Border
1 0 1 0 1 0 1 0	10 10 10 10	Chartr	Chartr	Chartr	Chartr
0 1 0 1 0 1 0 1	01 01 01 01	Border	Border	Border	Border
1 0 1 0 1 0 1 0	10 10 10 10	Chartr	Chartr	Chartr	Chartr
0 1 0 1 0 1 0 1	01 01 01 01	Border	Border	Border	Border
1 0 1 0 1 0 1 0	10 10 10 10	Chartr	Chartr	Chartr	Chartr
0 1 0 1 0 1 0 1	01 01 01 01	Border	Border	Border	Border

Figure 12-5 is more complex. SHIFT-Q in multicolor mode includes all four colors. It also loses some of the symmetry of the original, becoming a solid rectangle in the auxiliary color, surrounded by samples of the three other colors. Try POKE 36879,30 to change the border to blue; the cyan parts of the character change to blue.

Figure 12-5. Multicolor Example Two

Normal:	Multicolor:	Displays As:			
0 0 0 0 0 0 0 0	00 00 00 00	Backgd	Backgd	Backgd	Backgd
0 0 1 1 1 1 0 0	00 11 11 00	Backgd	Auxily	Auxily	Backgd
0 1 1 1 1 1 1 0	01 11 11 10	Border	Auxily	Auxily	Chartr
0 1 1 1 1 1 1 0	01 11 11 10	Border	Auxily	Auxily	Chartr
0 1 1 1 1 1 1 0	01 11 11 10	Border	Auxily	Auxily	Chartr
0 1 1 1 1 1 1 0	01 11 11 10	Border	Auxily	Auxily	Chartr
0 0 1 1 1 1 0 0	00 11 11 00	Backgd	Auxily	Auxily	Backgd
0 0 0 0 0 0 0 0	00 00 00 00	Backgd	Backgd	Backgd	Backgd

Program 12-25 displays characters in multicolor mode and then rapidly changes the auxiliary color for an interesting effect.

Program 12-25. Multicolor Demo

```
1Ø S=PEEK(646):POKE646,1Ø:FOR B=1 TO 15:FOR A=65TO
    9Ø:PRINT CHR$(A);:NEXT:NEXT
2Ø FOR A=1TO1Ø:FOR J=ØTO15:POKE 36878,J*16:NEXT:NE
   XT:POKE 646,S
```

With some work, characters in multicolor mode can produce impressive effects. Typically, we can enlarge the screen format so the border color isn't visible, which gives us another color to work with. Alternatively, we might set the border red and the background black, since red is likely to be a useful color, and leave them. If the auxiliary color is white, we have red, white, and the local color on a black background.

Multicolor mode is characterized by thin horizontal lines—the screen has a 3:2 ratio, so taking pairs of dots produces graphics made of little lines about three times as long as they are high. With a little practice it's possible to classify most graphics as high-resolution or multicolor just by looking at them.

The four colors needn't all be different. Try, for example, POKE 36879,30 (sets the border blue), and POKE 36878,16 (sets auxiliary color white). Now POKE 646,14 so the character color is blue. Typing a few characters gives a new character set, resembling normal characters, which can be freely mixed with normal characters by POKE 646,6. All these characters have a chunky appearance. They are useful for decorative borders and designs, and for graphics, and are easier to use than user-defined characters. They also take up no extra space in RAM. Finding characters which look right—a frog or whatever—may be difficult though.

The next BASIC program displays most of these multicolor characters, neatly separated by spaces, by POKEing them into the unexpanded VIC-20's screen. Function key f1 toggles the border color between black and white; f3 toggles the auxiliary color between black and white; and f5 toggles the character color. The background color is always white. If you set all four colors white, the characters disappear. There are 16 different character sets, allowing for lowercase and uppercase too, which means there are about four thousand new characters available. They cannot be mixed together on the screen, of course. If you wish, the program can be easily modified to analyze the effects of color changes of the character color, border, auxiliary, and background.

Program 12-26. Multicolor Characters

```
10 PRINT"{CLR}":POKE 36879,24:REM BLACK BORDER, WH
   ITE SCREEN
20 FOR J=7680 TO 7680+511 STEP 2
30 POKE J,X:POKE J+30720,8:X=X+1:NEXT
40 GET X$
50 IF X$="{F1}" THEN POKE 36879,49-PEEK(36879):REM
    F1 TOGGLES BORDER
60 IF X$="{F3}" THEN POKE 36878,16-PEEK(36878):REM
    F3 TOGGLES AUXILIARY
70 IF X$="{F5}" THEN P=PEEK(38400)AND15:P=17-P:FOR
    J=38400TO38911 STEP 2:POKE J,P:NEXT
80 GOTO 40
```

The Reverse Bit

Bit 3 in $900F, as we saw in Chapter 6, is the reverse bit. This is conceptually tricky, and I have never found software which uses it. When the bit is set to 0, by POKE 36879, PEEK (36879) AND 247 or some equivalent, each character, either 8 × 8 or 8 × 16, is reversed—the background and foreground colors switch. If several colors have been used, this inevitably results in a set of rectangular patterns disrupting the display. However, if only two colors have been used, the effect is a straightforward reverse display.

Changing the Physical Position and Dimensions of the Screen Display

Four VIC chip registers, the first four, control the position of the rectangular display within the screen, and its number of columns and rows. Another bit controls the character size, and can select 8 × 8 or 8 × 16 pixels. Between them these registers determine the physical shape and position of the display. Often of course they are left at the default values set when the VIC is switched on—22 columns × 23 rows, and left margin 5, top margin 25. (UK equivalent margins are 12 and 38. To be international, these values need to be modifiable within the program.)

The following BASIC program has four straightforward demonstrations which modify the distances from the top and left of the screen, allowing the whole screen to move in a circle, for example. Line 1 matches background and border colors to delete the rectangular boundary between the text and the border.

Program 12-27. Shift Screen Demo

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
1 POKE36879,68+8                                    :rem 110
10 PRINT"1=RANDOM":PRINT"2=CIRCULAR"                 :rem 48
15 PRINT"3=VERTICAL":PRINT"4=HORIZONTAL":INPUT X
                                                    :rem 169
20 ON X GOTO 100,200,300,400                         :rem 94
100 POKE 36864,8+9*RND(1):REM AVERAGE VALUE IS 12
                                                    :rem 184
110 POKE 36865,20+36*RND(1):REM AVERAGE VALUE IS 3
    8                                                :rem 28
120 GOTO 100                                         :rem 93
200 FORI=0 TO 2*↑ STEP .639                          :rem 57
210 POKE 36864,12+12*SIN(I)                         :rem 152
220 POKE 36865,38+38*COS(I)                         :rem 165
230 NEXT                                            :rem 212
240 GOTO 200                                         :rem 97
300 FOR I=160 TO 0 STEP -1                            :rem 4
310 POKE 36865,I                                     :rem 68
320 FOR J=1 TO 20:NEXTJ,I                           :rem 113
330 FOR I=0 TO 22                                    :rem 58
340 POKE 36865,I                                     :rem 71
350 FOR J=1 TO 20:NEXTJ,I                           :rem 116
360 GOTO 10                                          :rem 51
400 FOR I=1 TO 23:POKE 36864,I:NEXT                 :rem 156
410 GOTO 400                                         :rem 98
```

The next program introduces the idea of altering the number of columns and rows, and also altering the left and top distances to center the screen. Starting with two parameters, U and L, which stand for the number of characters the top left of the screen is to be moved up and left, each of the four VIC registers is altered.

The calculations are a little complex: Moving the display a single character left means reducing $9000 by 2*L, because each bit change only moves half a character. To be symmetrical, the number of columns must be increased by 2*L. Accuracy up and down is to one-fourth a character, so shifting the screen up a whole character means POKEing a new value 4*U less. Again, for symmetry, the number of rows has to be increased by 2*U, but because this parameter in $9003 starts at bit 1, not bit 0, this value has to be doubled.

Program 12-28. Nonstandard Size VIC-20 Screen

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
1 POKE 648,28:SYS58648:REM SCREEN AT $1C00 :rem 75
2 POKE 56,28:CLR:REM BASIC ENDS AT $1C00  :rem 180
5 U=5:REM UP 5 CHRS                          :rem 36
6 L=2:REM LEFT 2 CHRS                       :rem 156
7 REM 28 BY 33 SCREEN                        :rem 70
```

```
20 POKE 36864,PEEK(36864)-2*L:REM MOVE LEFT
                                          :rem 158
30 POKE 36866,PEEK(36866)+2*L:REM INCREASE #COLS
                                          :rem 221
40 POKE 36865,PEEK(36865)-4*U:REM MOVE UP  :rem 39
50 POKE 36867,PEEK(36867)+4*U:REM INCREASE #ROWS
                                            :rem 6
100 CR=37888                              :rem 117
110 FOR J=28*256 TO 28*256+(22+2*L)*(23+2*U)-1
                                          :rem 210
120 POKE J,5                              :rem 109
130 POKE CR+X,2:X=X+1                     :rem 188
140 NEXT                                  :rem 212
```

The program fills the new screen with the character E in red, by POKEing 5's into the whole newly designed screen area, and shows that an enlarged screen is in fact possible. It is also possible to PRINT in almost the usual way; the easiest method is to use a function to position the cursor before PRINT, like this:

```
10 SC=7168 + 26*V + H: REM ASSUMES SCREEN AT $1C00 AND 26 COLUMNS
20 POKE 209, SC AND 255: POKE 210, SC/255: REM POINT INTO THE SCREEN RAM
30 PRINT "HELLO!": REM HELLO! STARTS H ACROSS, V DOWN
```

Smooth Screen Scrolling

Sometimes it's nice to be able to display smoothly scrolling text, or landscape graphics shifting left or right. The scrolling we've seen so far shifts whole characters, and is therefore somewhat jerky. We can improve on this without much extra work with VIC's facility to move the screen.

Its finest resolution is half a character horizontally, and a quarter character vertically, controlled by locations $9000 (36864) and $9001 (36865), respectively. To perform an upward scroll, the screen's position is moved upward using $9001, and every fourth movement the screen is moved back to its normal position and scrolled. This gives a jiggling motion at the top and bottom borders, which can be reduced by matching the border and background colors, or removed entirely by expanding the screen outside the total TV area. The screen scroll must be as fast as possible. If it is slow, the TV will catch the instant when the screen is shifted. We can demonstrate in BASIC; later we present some ML routines which are faster.

Horizontal motion is also possible, although resolution to only half a character weakens the effect. VIC BASIC has no sideways scroll, so a pure BASIC demo isn't possible—use SYS 828 with one of the earlier side-scrolling subroutines.

Program 12-29. BASIC Demo of Smooth Text Scrolling

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 POKE 36879,25:PRINT"{CLR}"             :rem 210
20 D$="{HOME}{21 DOWN}"                   :rem 195
30 READ X$: IF X$="!" THEN RESTORE:READ X$  :rem 7
40 PRINTD$X$                              :rem 213
50 GOSUB 10000                            :rem 214
```

```
60 GOTO 30                                          :rem 2
70 DATA IN THE FAR OFF LAND, OF COMMODORIA,LIVED J
   ACK,!                                          :rem 125
10000 P=PEEK(36865):S=59765                       :rem 218
10010 POKE 36865,P-1:GOSUB 11000                  :rem 179
10020 POKE 36865,P-2:GOSUB 11000                  :rem 181
10030 POKE 36865,P-3:GOSUB 11000                  :rem 183
10040 POKE 36865,P:SYSS                            :rem 56
11000 FOR J=1 TO 100 :NEXT:RETURN                  :rem 88
```

Line 11000 allows the delay between one-fourth line scrolling to be varied. SYS59765, which scrolls the screen, is slow by ML standards, so there is some flicker.

The two ML subroutines following are faster versions which are free of flicker. SYS 828 with the first subroutine scrolls the screen up one-fourth character, so SYS 828: FOR J=1 TO 100: NEXT within a loop can replace the BASIC subroutine in the demonstration. A test such as: IF PEEK (36865)=22 THEN READ X$ allows printing every fourth scroll.

The other ML subroutine scrolls up one-fourth character but also rotates the top line to the bottom, giving an infinite repeat; to do this it keeps a table of 22 characters and 22 color RAM entries at each scroll. The result is an effective continual scrolling.

One way to properly eliminate jitter at the edges with this method is to expand the screen's boundaries. It's not as easy to write a routine which can allow for all the parameters of screen margin setting and altered numbers of rows and columns.

This technique works with double-sized characters, except that seven increments are needed before each overall scroll.

Program 12-30. Scroll Up One-Fourth Character

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
0 DATA 173,1,144,201,22,208,7,169,25,141,1,144,208
  ,4,206,1,144,96                                 :rem 105
1 DATA 162,6,189,16,30,157,250,29,189,16,150,157,2
  50,149,232,208                                   :rem 91
2 DATA{2 SPACES}241,189,250,30,157,228,30,189,250,
  150,157,228,150,232,208                          :rem 82
3 DATA 241,96                                     :rem 127
10 REM***** SYS 828 SCROLLS UP 1/4 OF A CHARACTER*
   *****                                          :rem 114
20 FOR J=828 TO 878: READ X: POKE J,X: NEXT:rem 26
30 REM REPLACE 22 25 IN LINE 0 WITH 35 AND 38 IN B
   RITAIN                                         :rem 127
```

Program 12-31. Scroll Up and Rotate Top Line

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
0 DATA 173,1,144,201,22,208,7,169,25,141,1,144,208
  ,4,206,1,144,96                                 :rem 105
1 DATA 160,21,185,0,30,153,145,3,185,0,150,153,167
  ,3,136,16,241,162                               :rem 199
```

```
2 DATA 6,189,16,3Ø,157,25Ø,29,189,16,15Ø,157,25Ø,1
  49,232,2Ø8,241                              :rem 9Ø
3 DATA 189,25Ø,3Ø,157,228,3Ø,189,25Ø,15Ø,157,228,1
  5Ø,232,2Ø8,241                              :rem 83
4 DATA 16Ø,21,185,145,3,153,228,31,185,167,3,153,2
  28,151,136,16,241                          :rem 223
5 DATA 96                                    :rem 19Ø
1Ø REM **** SYS 828 SCROLLS UP AND ROTATES SCREEN
   {SPACE}CONTENTS ****                       :rem 32
2Ø FOR J=828 TO 912:READ X:POKE J,X:NEXT    :rem 15
3Ø REM REPLACE 22 AND 25 IN LINE Ø WITH 35 AND 38
   {SPACE}IN BRITAIN                          :rem 82
```

Smooth Scroll Up Two Pixels

```
SCREEN    =    $1E00
COLRAM    =    $9600
TOP       =    $9001
TOPPOSN   =    25      ;BRITAIN = 38
               *=$033C
               LDA  TOP
               CMP  +TOPPOSN-3
               BNE  NOSCROLL
               LDA  +TOPPOSN
               STA  TOP
               BNE  SCROLLUP
NOSCROLL       DEC  TOP
               RTS
SCROLLUP       LDX  +6
LOOP           LDA  SCREEN+22-6,X
               STA  SCREEN-6,X
               LDA  COLRAM+22-6,X
               STA  COLRAM-6,X
               INX
               BNE  LOOP
LOOP2          LDA  SCREEN+228+22,X
               STA  SCREEN+228,X
               LDA  COLRAM+228+22,X
               STA  COLRAM+228,X
               INX
               BNE  LOOP2
               RTS
```

User-Defined Characters

The VIC Chip and the Screen Position and Character Table Position

We saw in Chapter 5 how the video interface chip is wired to the internal memory of the VIC-20. Memory location $9005 (36869) is the most important determinant of graphics display; two other bits, one from each of locations $9002 and $9003 (36866 and 36867), are also essential.

To understand the display system, let's review this information. Switch on the VIC-20 without memory expansion, and PRINT PEEK(36866) PEEK(36867) PEEK (36869). These values are 150, 46 or 174, and 240. PEEK(36867) is an even number, which means that bit 0 is zero. PEEK(36866) is greater than 127; therefore, bit 7 is set. Location $9005 (36869) contains 240 ($F0). The bit pattern of its contents is 1111 0000.

These bits can be used to change the location of the screen and character set. The VIC chip constructs a 16-bit address from these bits as follows:

```
               $9005 bits 6 54
                          ↓ ↓↓
Screen start address= 0001 1110 0000 0000
                             ↑
                   $9002 bit 7

               $9005 bits 32 10
                          ↓↓ ↓↓
Character set address= 0000 0000 0000 0000
```

Arranging these bit patterns in the usual hexadecimal format of four bits together gives the screen start as $1E00 and the character set start as $0000. But when bits 2 and 3 of $9005 are zero, the VIC chip sees the character set address as $8000 plus the value of bits 0 and 1. So in this case, the start of the character set is actually $8000.

The unexpanded VIC-20 sets itself with the screen RAM from $1E00 to $1FFF, and with characters defined in eight-byte sets from $8000 onwards, which as we've seen is the uppercase character set. You can check that pressing SHIFT and the Commodore key alters the contents of $9005 to 242 ($F2 or 1111 0010) which leaves the screen position unchanged but alters the character definitions to start at $8800, which is the lowercase character set.

Two tables (one decimal, one hex) in Appendix G show every usable combination of screen and character definitions. Note, for example, how 240 in 36869 and 150 in 36866 correspond to a screen starting address of 7680 and a character table start address of 32768. Moreover, the color RAM position is fixed by bit 7 in $9002 (36866), and has two alternative positions, at $9400 or $9600.

The Screen Position and How to Control It

To set the screen properly, it is necessary at least in BASIC to POKE 648 ($288) with the correct screen page, so that the screen editor can work properly. PRINT uses a location to hold the current color, and in the same way location 648 points to the part of memory to which characters are to be POKEd, and this should normally be the screen.

An unexpanded VIC-20 has 30 ($1E), pointing to $1E00; an 8K or 16K VIC-20 has 16 ($10). With the aid of the table, it is possible to put the screen wherever you like within the limits imposed by the system. For example, POKE 36869,226: POKE 36866,22: POKE 648,24 puts the screen at 6144 and its color RAM at 37888, with the lowercase character set, as can be verified by clearing the screen and typing a few characters. The two screen positions used by the VIC-20 are only a subset of the total available number. Naturally, BASIC pointers may have to be changed to avoid the screen; simply shifting the screen is unlikely to work if a program uses strings, for example. Later in this section we'll have examples illustrating exactly what to do.

The VIC-20 automatically assigns 512 bytes to the screen. For example, an 8K expanded VIC-20 has its screen start address at $1000 (4096), and BASIC defined to start at $1200 (4608), leaving a limit of 512 bytes. It is important to realize that this apparent limit of 512 bytes is not inherent in the VIC chip, which can address far more bytes. 1024 is the practical maximum.

This is why I've used the phrase "Start of Screen Position" and variations on it. If the screen is redefined to be only 10 × 10 characters, only the first 100 bytes after the address as defined by the VIC chip will be used; if the screen is 25 × 25, then 625 consecutive characters after the address will be used. Provided this is borne in mind, there should be no problem with corruption caused by overlap of programs, characters, and screens.

The Character Set Position

Like the screen, the character set position defines the start of an area of memory. From it, the VIC chip will take its character information. Since 256 characters maximum can be defined, the total extent of the character set can be 256 x 8 = 2048 ($0800) bytes, or if double-sized characters are used, 256 x 16 = 4096 ($1000) bytes. For this reason, POKEing 36867 with 47 to set double-sized characters means that the entire character set from $8000 to $8FFF can be accessed, and reversed and lowercase characters can appear with text characters (but only in pairs), in a way which is normally impossible. Try

```
POKE 36867,47: FOR J=7680 TO 7935: POKE J,X: POKE J+30720,0: X=X+1: NEXT
```

with an unexpanded VIC-20.

Unlike the screen table, the characters need not form a continuous area of RAM. Character definitions can be started at $0, for example, and only certain selected characters actually used, with their definitions in the region $0 to $03FF. This is a tricky technique most suited to ML programs.

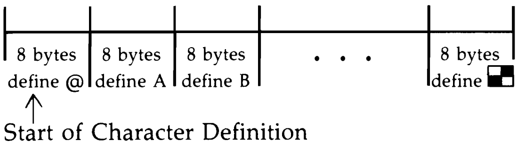
How Graphics Appear on the Screen

Figure 12-6 shows how 8 × 8 characters are stored in RAM and displayed. Figure 12-7 is the same as Figure 12-6 except each character is 8 × 16. Usually, 506 of these are displayed in one of the two ordinary text modes (Figure 12-6). Note that, because of screen RAM duplication of characters, typically there will be many spaces.

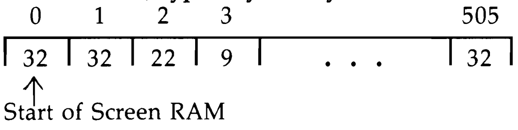
When characters are 8 × 16, they are upright, and the number of rows which they fill must always be equivalent to an even number of 8 × 8 character rows. Thus it is impossible to generate a screen the same size as the ordinary 22 × 23 screen. This mode is often characterized by a shallower picture than normal.

Figure 12-6. 8 × 8 Character Definitions and the Screen

Character definitions extend 2048 bytes (512 characters).



Screen RAM, typically 506 bytes.



Screen: Arrangement of screen RAM—22 across, 23 down

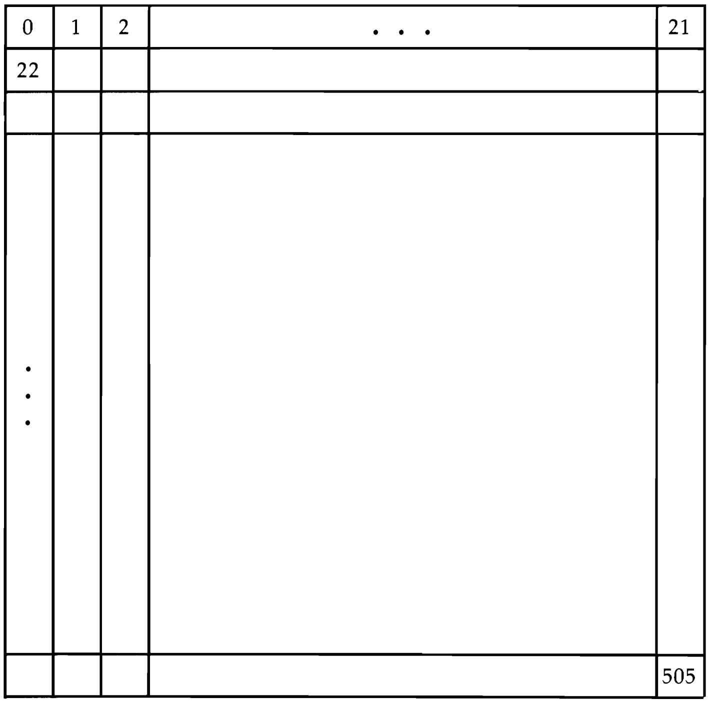
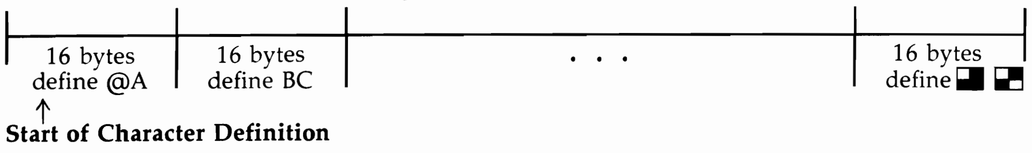
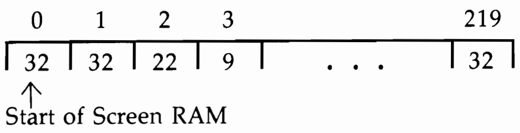


Figure 12-7. 8 × 16 Character Definitions and the Screen

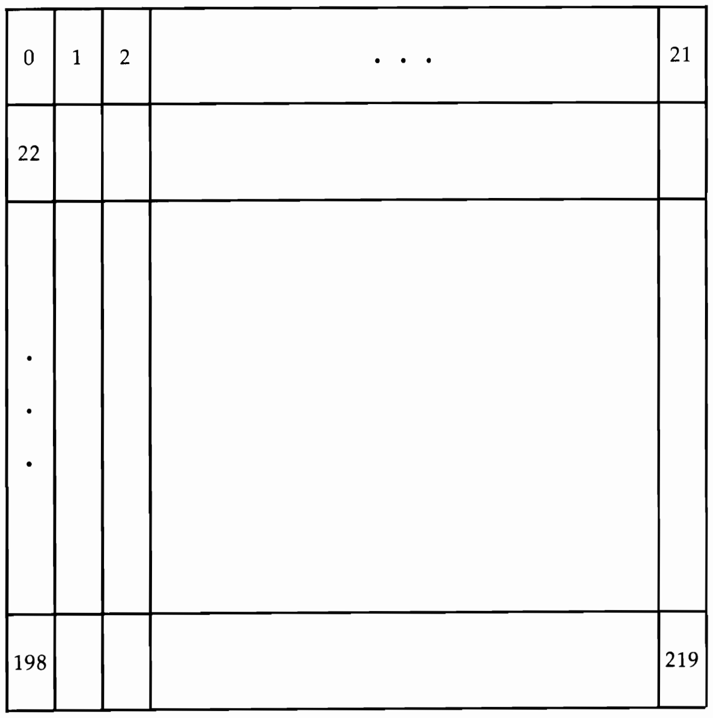
Character definitions extend 4096 bytes.



Screen RAM, typically 220 bytes.



Screen: Arrangement of screen RAM—22 across, 10 down



In either case, the VIC chip takes the RAM character from the screen, multiplies by 8 or 16, adds this to the start address as stored in VIC's registers, then uses the 8 or 16 bytes to construct the character's pattern of dots in the correct screen position.

In principle, the method of defining your own character set is straightforward. The VIC chip is reset to take its characters from RAM, not ROM. The usual area for storage of this information is $1000–$1FFF (4096–8191). The screen is nearly always in this area too; so is BASIC, in unexpanded VIC-20s. Therefore, several problems must be dealt with: shortage of space, keeping BASIC and the characters from overlapping, and moving the character information into RAM.

Why bother with double-sized characters? The reason is simple: They enable the entire screen to be bitmapped, so that true high-resolution graphics, in which any point on the screen can be turned on or off, is possible.

To be precise, the screen can hold only 256 entirely different characters, so double-size mode allows a screen the size of 512 normal characters to be 100 percent bitmapped, at a cost of doubling, or more than doubling, the RAM needed to hold the bits. The biggest fully bitmapped screens are 25 × 20 normal characters, or 23 × 22, or 21 × 24, or other combinations of similar size.

However, the VIC is connected internally so that the screen and character definitions normally coexist in VIC-20's built-in memory from $1000 to $2000. (It's also possible to put the screen at $200.)

What's the largest obtainable fully bitmapped screen? If the screen is at $200, all 4096 bytes from $1000 to $1FFF can hold character definitions. If both the screen and characters are to be stored in $1000–$1FFF, the character space is less. In fact, a 240-byte screen can be achieved with either layout, which is 24 columns × 10 double-sized rows, resembling 24 × 20 normal characters, or 20 columns × 24 rows. Only with the screen starting at $200 are 22 columns × 22 rows possible.

Fitting a 24 × 20 screen and characters into $1000–$1FFF can be done only by starting the screen and the characters at the same address, $1000, and using only the 240 characters from 16 to 255, so that the first character, CHR$(16), starts after the screen.

Double-sized graphics is another reason for double-sized characters. Whenever it's more convenient to manipulate 8 × 16 rather than 8 × 8 user-defined characters, it makes sense to define them in this size from the start; double-sized lettering is a good example.

Incomplete High-Resolution Graphics

Often a picture or other graphic display on the screen doesn't need 100 percent bitmapping; there may be large spaces, which can all be mapped by a single space character without wasteful duplication. An unexpanded VIC-20 can plot graphs (even with an enlarged screen) like this.

Memory Maps of the VIC-20 in Its Graphics Modes

How Many Configurations?

Many books and magazine articles give the impression that there are perhaps one or two, but no more, graphics configurations of the VIC. In fact, the number of permutations is enormous and can't be exhaustively covered.

This section describes configurations and their uses and advantages, in ascending order of complexity and sophistication.

Some unexpanded VIC-20 configurations.

Figure 12-8. Character Definitions at $1800, Screen Unchanged at $1E00

$1000	$1800	$1E00
BASIC	Characters	Screen

The difference from the normal unexpanded VIC-20 is that the character set starts in RAM at $1800 (6144), and therefore the top of BASIC RAM must be lowered to $1800.

Use POKE 56,24: CLR: POKE 36869,254 when VIC-20 is originally configured normally. This can be included as the first line of a program, since the start of BASIC itself isn't affected.

BASIC space is now 2048 bytes. There are 1536 ($600) bytes available for character definitions, space for 192 normal characters.

Figure 12-9. Character Definitions at $1800; Enlarged Screen Starting at $1C00

$1000	$1800	$1C00
BASIC	Characters	Screen

Character definitions and the start of screen are both moved. Top of BASIC RAM must be lowered. POKE 56,24: CLR: POKE 36869,254: POKE 36866,22: POKE 648,28 (direct or program mode) gives this configuration.

The entire screen can now be filled, after POKEing the parameters controlling the position of the window and its number of rows and columns. BASIC space is again 2048 bytes, and there's room for 128 user-defined 8 × 8 characters.

Figure 12-10. Loading User-Defined Characters with a BASIC Program

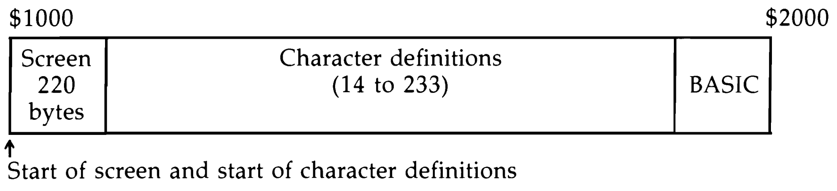
$1000	$1800	$1C00	$1E00
BASIC	Character Definitions	Variables	Screen

This more unusual configuration allows BASIC to include character definitions (and/or machine language) after the program, in such a way that LOAD automatically loads both together. This is very convenient, though tricky.

BASIC must not extend above $1800. When the program is saved, the pointers to the start of variables must be moved up to include the character definitions within what will be taken to be BASIC. POKE 45,0: POKE 46,28 in the example.

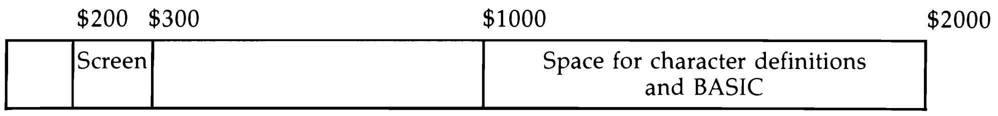
BASIC space is 2048 bytes, excluding variables, which have 512 bytes. (To check whether this is sufficient, remember that simple variables take seven bytes; then add the length of strings.) There's room for 128 user-defined characters.

Figure 12-11. 22 × 20 High-Resolution Graphics


$1000
$2000
Screen 220 bytes
Character definitions (14 to 233)
BASIC
Start of screen and start of character definitions

We need double-sized characters for full bitmapped high resolution. In this mode, 22 columns × 10 rows is equivalent to 22 × 20 normal characters. The screen must have 220 bytes, and we'll need 220 characters too; the diagram shows the most efficient way to get this, and uses only CHR$(14) through CHR$(233). This leaves 352 bytes for BASIC; just enough for a five- or six-line demonstration program.

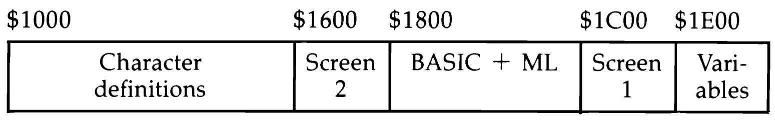
Figure 12-12. Arrangement with the Screen at $200



This is a technique for skilled programmers only. It has the advantage of making more RAM available for graphics definitions.

In BASIC, double-sized graphics are necessary to avoid the table of pointers at $0300.

Figure 12-13. Exotic Arrangement



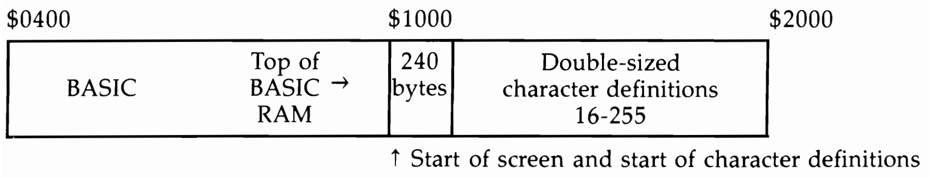
This is not intended as a recommendation or suggestion; simply to show how the soft nature of VIC allows a lot of flexibility. It has two alternate screens, neither in the conventional position, and each with its own color RAM, plus space for user-defined characters, and BASIC (including some machine language at the end) separated from its variables by a screen.

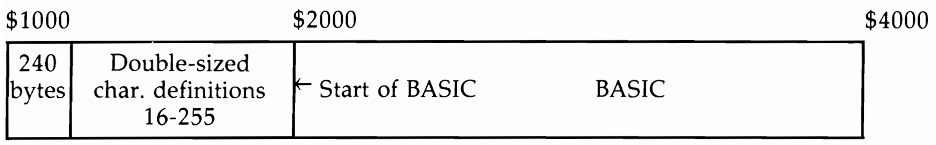
Because the start of BASIC has been moved from $1000, a configuration like this needs a loader, a short program that does the POKEs to reconfigure memory, then loads the main program into the correct place.

Expanded VIC-20

An expanded VIC-20 is of course an unexpanded VIC-20 with either 3K of RAM just before $1000, and/or 8K, 16K, or 24K of RAM just after $2000. 16K and 8K expansions are most popular. Configurations follow the same rules as the unexpanded VIC-20. The extra RAM is usable only for BASIC or machine language storage; character definitions and the screen have to occupy the usual positions, typically $1000 to $1FFF. (This is why 3K with 16K on an expansion board doesn't automatically enlarge BASIC.)

Figure 12-14. Expanded VIC-20


$0400
$1000
$2000
BASIC
Top of BASIC RAM →
240 bytes
Double-sized character definitions 16-255
↑ Start of screen and start of character definitions



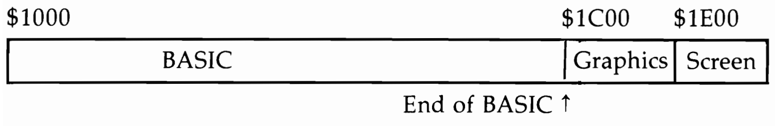
Full bitmapping of a 24 × 20 screen has this configuration. BASIC can be stored in 3K, 8K, or other expansion; BASIC pointers must be set correctly as shown.

Programs for Exploring VIC-20's Graphics Potential

Character Editor

Program 12-32 is a useful character editor, which you can use to design your own graphics. The editor assumes you are using an unexpanded VIC-20; if you are using an expander, POKE 648,30: SYS 64818. Its memory configuration is shown in Figure 12-15.

Figure 12-15. Short High-Resolution Character Editor



The program lowers the top of BASIC, sets the character definition pointers to $1C00 (7168)—giving room for 64 individual graphics—and copies the first 64 characters from ROM into RAM. You will be able to watch as they progressively build up. RUN/STOP–RESTORE returns to normal—so that BASIC can be LISTed without appearing as frogs, trucks, or whatever—as does POKE 36869,240.

Using the Character Editor

The character editor will display 64 of the current graphics characters in the middle of the screen. You are asked, WHICH CHARACTER? Respond with the character you wish to redefine.

As redefinition of the characters proceeds, it will become difficult to know exactly which key corresponds to which character; it is therefore advisable to note which key goes with which new character.

You are now asked, DELETE? and you're expected to type Y (Yes) to erase the character and start afresh, or N (No) to modify a character without deleting it. In either case, a pattern of 8 × 8 dots is printed at the top left of screen; asterisks represent bits which are on, and each byte's decimal equivalent provides a record of the character's structure.

Incidentally, the bit pattern of VIC's own character set can be scrutinized. The cursor position is marked by auxiliary color mode, which should be more or less visible in most settings of the program. Enter new asterisks or spaces by moving around with HOME, RETURN, and other cursor control keys, and watch the whole character change. It's printed in each color, and in multicolor mode, near the bottom of the screen, to give a better idea of its appearance in bulk. Note that defining the asterisk or the period will change the appearance of the 8 × 8 array.

You can move to the right of the screen with the cursor and use this area as a scratch pad—to hold new graphics characters and to check that they match properly.

The function keys: Press f2 when you've finished with a character; you will be returned to WHICH CHARACTER? F8 is intended as an exit so you can save the created characters either with a BLOCK SAVE, or with a change of BASIC pointer enabling BASIC and graphics to be saved together (see "Saving Graphics Sets to Tape or Disk" below). F1 cycles through character colors, in the current character; f3 through background; f5 through border; and f7 through auxiliary color.

About the program. If the function keys are omitted, this program can be compressed into about 11 lines. Line 5 configures the top of BASIC and the start of the character definitions; the screen is assumed at $1E00. Line 6 copies 64 characters from ROM into the new character definition area. Line 30 puts 64 black characters in midscreen, and lines 40–45 get the key to be edited and find its position in the table of characters, offset from the start by a multiple of 8. The ASCII value of alphabetic keys isn't identical to the POKE value, hence line 44. Line 60 clears this character if Y is pressed. Line 70 puts in 80 characters in 16 colors, including multicolor mode.

Lines 80–86 print the 8 × 8 array of dots and asterisks, with the contents of each byte, at the top left. Lines 100–200 are all concerned with inputs—space, asterisk, function keys, or cursor control keys are assumed. Note that locations 211 and 214 keep track of the cursor position in a relatively simple manner. The POKEs into M in lines 100 and 120 control multicolor/normal modes in the cursor. P in line 130 is the byte corresponding to the row being edited.

We can increase the number of characters by lowering the character set start to $1800, at the expense of reducing space for BASIC to 2K. All that's needed is POKE 36869,254 and POKE 56,24 in line 5. This makes space for 128 characters.

Program 12-32. Character Editor

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
5 POKE36869,255: POKE 55,Ø: POKE56,28: CLR:rem 240
6 FOR J=7168 TO 7679: POKE J,PEEK(32768+M): M=M+1:
   NEXT                                   :rem 183
2Ø PRINT "{CLR}{14 DOWN}WHICH CHARACTER?"; :rem 46
3Ø FOR J=Ø TO 63: POKE 79ØØ+J,J: POKE 3862Ø+J,Ø: N
   EXT                                    :rem 231
4Ø GET C$: IF C$="" GOTO 4Ø               :rem 249
42 PRINT C$                                :rem 9Ø
44 C=ASC(C$): IF C>63 THEN C=C-64         :rem 179
45 C=C*8                                  :rem 142
5Ø PRINT "DELETE? (Y/N)";                 :rem 138
52 GET D$: IF D$="" GOTO 52                 :rem 1
54 IF D$<>"Y" GOTO 7Ø                      :rem 23
6Ø FOR J=7168+C TO 7168+7+C: POKEJ,Ø: NEXT:rem 233
7Ø PRINT: FOR J=Ø TO 79: POKE 646,J/5: PRINT C$;:
   {SPACE}NEXT: POKE 646,6                 :rem 15
8Ø PRINT "{HOME}";: FOR Y=Ø TO 7: P=7168+C+Y: FOR
   {SPACE}X=Ø TO 7                        :rem 151
82 IF (PEEK(P) AND 2↑(7-X))>Ø THEN PRINT"*";: GOTO
    86                                     :rem 52
84 PRINT ".";                             :rem 166
86 NEXT: PRINT TAB(8) PEEK(P): NEXT        :rem 19
9Ø PRINT "{HOME}";                        :rem 136
1ØØ X=PEEK(211): Y=PEEK(214): M=384ØØ+22*Y+X: POKE
     M,1Ø                                 :rem 168
11Ø GET G$: IF G$="" GOTO 11Ø              :rem 93
12Ø POKE M,6                              :rem 113
13Ø P=7168+C+Y                            :rem 233
14Ø IF G$="{F2}" GOTO 2Ø                   :rem 52
142 IF G$="{F1}" THEN POKE 646,PEEK(646)+1: POKE M
    -1,PEEK(M-1)+1                        :rem 24Ø
144 IF G$="{F3}" THEN POKE 36879,PEEK(36879)+16 AN
    D 255                                 :rem 189
146 IF G$="{F5}" THEN POKE 36879,(PEEK(36879) AND
    {SPACE}247)+1 OR 8                    :rem 181
148 IF G$="{F7}" THEN POKE 36878,PEEK(36878)+16 AN
    D 255                                 :rem 193
15Ø IF G$="{F8}" GOTO 2ØØØØ               :rem 2ØØ
16Ø IF G$<>" " AND G$<>"*" THEN PRINT G$;: GOTO 1Ø
    Ø                                     :rem 219
17Ø IF G$="*" THEN PRINT "*{LEFT}";: POKE P,PEEK(P
    ) OR 2↑(7-X)                           :rem 4Ø
18Ø IF G$=" " THEN PRINT ".{LEFT}";: POKE P,PEEK(P
    ) AND NOT 2↑(7-X)                      :rem 38
19Ø X=PEEK(211): Y=PEEK(214): POKE 646,Ø: PRINT TA
    B(8) PEEK(P)"{LEFT}{2 SPACES}";       :rem 132
2ØØ POKE 211,X: POKE 214,Y: GOTO 1ØØ       :rem 98
```

Often the easiest way to use characters of this sort is simply as individual new characters, accepting the VIC size constraint. Many games do this; it is the line of least resistance and avoids the hassles involved in dealing with several bunched characters or with fractions of characters. Applications include special alphabet sets, perhaps with accents, in which case several versions of a letter will need to be stored separately; games pieces; math or music symbols; and of course the familiar arcade-style pictures. Below is some data for characters you might find useful.

A knight: 8,28,54,67,81,107,17,63
A bishop: 8,20,42,62,42,28,93,99
Trucks and cars: 15,21,21,63,63,45,18,0; 0,0,56,40,127,127,34,0; 63,113,81,113,127,127,54,0; 0,0,0,30,106,127,93,34
A "not equal" sign: 0,20,34,65,34,20,0,0

Program 12-33 creates an Islamic pattern using only four graphics "tiles" selected in a random order and repeatedly printed to the screen, which is enlarged. This is an example of effective use of these relatively simple user-defined graphics.

Program 12-33. Islamic Designs

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 POKE 648,28: POKE 36869,254: POKE 56,24: CLR
                                             :rem 137
20 C=1: FOR J=37888 TO 38688: POKE J,C: NEXT: REM
   {SPACE}COLOR RAM WHITE                     :rem 26
30 POKE 36879,8                                :rem 7
40 FOR J=6144 TO 6144+39: READ X: POKE J,X: NEXT:
   {SPACE}REM GRAPHICS                       :rem 119
50 POKE 36864,3: REM LEFT MARGIN (9 IN UK):rem 198
60 POKE 36866,25: REM 25 COLUMNS           :rem 219
70 POKE 36867,(PEEK(36867) AND 129) OR 30*2: REM 6
   0 ROWS                                     :rem 67
80 POKE 36865,15: REM TOP MARGIN (24 IN UK)
                                             :rem 242
100 R=1+RND(1)*25: DIM A%(R): REM MAX PERIOD OF RE
    PETITION IS 25                           :rem 242
110 FOR J=1 TO R-1                           :rem 132
120 A%(J)=RND(1)*5: REM SELECT RANDOM CHARS
                                             :rem 166
130 NEXT                                     :rem 211
200 FOR J=7168 TO 7968                        :rem 87
210 K=K+1:IF K>R THEN K=1                     :rem 78
220 POKE J,A%(K):REM POKE REPEATED SEQUENCES OF CH
    ARACTERS 0 TO 4                          :rem 250
230 NEXT                                     :rem 212
300 GET X$: IF X$="" GOTO 300                :rem 129
310 RUN 100                                   :rem 26
1000 DATA 36,72,144,33,66,132,9,18: REM 8 BYTES DE
     FINING CHR$(0)                           :rem 39
1010 DATA 36,18,9,132,66,33,144,72: REM CHR$(1)
                                              :rem 38
```

```
1020 DATA 36,36,255,0,0,255,36,36         :rem 77
1030 DATA 36,36,231,36,36,231,36,36      :rem 180
1040 DATA 36,66,153,36,66,36,153,66:REM CHR$(4)
                                         :rem 104
```

Saving Graphics Sets to Tape or Disk

Chapter 6 has routines which enable areas of memory to be dumped unchanged onto tape or disk, complete with a name which the user can type in.

Saving characters along with BASIC is a little tricky. This is how it's done with Program 12-32 above. Add these lines:

```
20000 POKE 45,0: POKE 46,30: POKE 55,0: POKE 56,32: CLR
20010 PRINT "{CLR}"
20020 INPUT "NAME"; N$: SAVE N$
```

These lines put the end of BASIC/start of variables pointer to $1E00. The other POKEs set the top of BASIC to $2000, in order to make room for the name to be stored and saved. Add *,8* to the end of line 20020 to save to disk instead of tape. This will now save the entire program plus characters. However, this isn't quite enough; when the program is loaded again, end of BASIC/start of variables is too high. So add this line too:

```
2 POKE 45,000: POKE 46,000
```

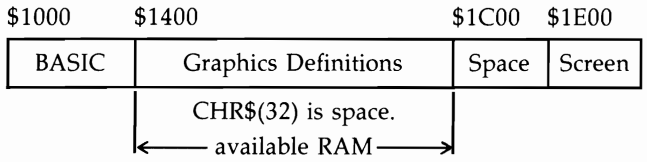
and when you've finished editing the program, so its final length is determined (for example, deleting line 6 so the characters are not overwritten), then put the values of PEEK(45) and PEEK(46) into this line. The reason for using three digits is so that we will not alter the program length (enter 15 as 015 or 9 as 009).

The trick, in short, is to remember that a program is saved from the start of BASIC to the end, as measured by pointers in 43 and 44 (start), and 45 and 46 (end). So POKEing and PEEKing locations 45 and 46 is what's needed.

Incomplete Graphics Examples

As we've seen, bitmapping the whole screen is impossible with an unexpanded VIC-20. The nearest approximation would be to map an area of about 24 × 20 normal characters, but this would still require memory expansion if room is to be left for any real BASIC program. A compromise solution exists, even for the unexpanded VIC-20, and the memory maps shown in Figure 12-16 illustrate two implementations. The point of this method is that very often at least half the screen holds spaces anyway, which can all be represented by one character, while the user-defined graphics deal individually with the other half.

Figure 12-16.Incomplete Graphics Memory Maps



(1) RAM for 256 normal (8 × 8) characters with $IC00–$IDFF unused, or
(2) RAM for 160 double-sized characters (8 × 16) with nothing unused.

These configurations confine BASIC to 1K, keep the screen its usual size, and still have room to store 256 normal characters or 160 double-sized characters. Double-sized characters can therefore cover a larger proportion of the screen, about 63 percent as compared with about 50 percent with normal characters. If an enlarged screen is used, though, this advantage disappears, and expanded screens can have a coverage of 50 percent maximum with these configurations.

Here's a short example program which should help make this clear.

Program 12-34. Compressed Version of Incomplete Screen Hi-Res for Unexpanded VIC

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 POKE56,20:CLR:FORJ=828TO849:READX:POKEJ,X:NEXT:
   SYS828                                  :rem 111
20 POKE36869,253:PRINT"{CLR}":POKE36879,8 :rem 227
100 FORX=0TO175                            :rem 125
110 Y=175-170*EXP(-((X-80)/40)↑4)          :rem 211
120 SC=7680+INT(X/8)+22*INT(Y/8)            :rem 22
140 CH=PEEK(SC):IFCH=32THENCH=N:POKESC,N:N=N+1:IFN
    =32THENN=33                            :rem 101
150 BYTE=5120+8*CH+(YAND7):POKEBYTE,PEEK(BY)OR2↑(7
    ANDNOTX):NEXT                          :rem 238
500 GOTO 500                                :rem 99
1000 DATA 162,8,160,0,132,253,169,20,133   :rem 156
1010 DATA 254,152,145,253,200,208,251,230 :rem 208
1020 DATA 254,202,208,246,96                :rem 97
```

About the program. The machine language subroutine puts zero bytes into all locations from $1400 to $1BFF. POKEing is of course slower. Without this (try it), the graphics as they are defined will contain garbage. Also line 140 includes a test for the space character so it will not be changed. If the space definition is altered, the whole screen will change, as you can see by omitting the test "IF CH=32..." and waiting until the program has plotted 32 characters.

Lines 120 to 150 plot high-resolution dots corresponding to X and Y, where X is the horizontal position, Y the vertical. The screen is 22 × 23, and therefore has 176 × 184 dots. X values in the range 0–175 and Y values from 0 to 183 can be plotted with the routine as it stands, with 0,0 in the top left.

Variables: SC is the byte in screen memory corresponding to X across and Y down. CH is the character stored in the screen RAM which is redefined to hold the new dot. BYTE is a location within the character definitions, which is why it begins at $1400 (5120). The new dot position is added to the character definition by ORing the present contents of the byte with 1, 2, 4, 8, or whatever in line 150.

Line 100 scans across the screen from left to right; STEP controls the resolution. More dots look better but take longer. Line 110 calculates Y values according to a formula. The values are selected to keep in the range 0 to 183. The formula in line 110 plots a bell-shaped curve; Y=X gives a straight line; Y=90+90*SIN(X/10) a sine curve.

The next program (12-35), also for an unexpanded VIC, has some enhancements: The screen window can be selected (line 2) to fill your TV screen; lines 4 and 6 control the position. Double-sized text can put a title on top; lines 10000 and after show this. Lines 90–96 have a scaling feature which insures that line 90's function will nearly always fit the screen. (Dots can be drawn, for example, with a joystick.)

Program 12-35. Incomplete Graphics with Text

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
1 POKE56,20:CLR:FORJ=828TO849:READX:POKEJ,X:NEXT:S
  YS828                                        :rem 63
2 R=26:C=23:REM CHOOSE ROWS,COLS              :rem 207
3 POKE36867,R+1:REM ROWS & DOUBLE SIZE        :rem 207
4 POKE36865,17 :REM TOP (21 IN UK)            :rem 255
5 POKE36866,128+C:REM COLS                    :rem 245
6 POKE36864,4 :REM LEFT (8 IN UK)             :rem 217
20 POKE36869,253:PRINT"{CLR}":POKE36879,8     :rem 227
90 DEFFNY(X)=(SIN(X/7 )+COS(X/9)↑2)            :rem 6
91 FOR X=0 TO (C+1)*8 STEP5                   :rem 124
92 Y = FNY(X)                                 :rem 151
93 IFMAX<Y THEN MAX=Y                          :rem 33
94 IFMIN>YTHEN MIN=Y                           :rem 32
95 NEXT                                       :rem 173
96 D=MAX-MIN:F=8*R/1.1/D                      :rem 138
97 POKE36879,10                                :rem 61
98 GOSUB10000                                 :rem 226
100 FORX=0TO8*C-1                             :rem 227
110 Y=FNY(X)                                  :rem 190
115 Y=F*(Y-MIN):Y=8*R-Y                        :rem 98
120 SC=7680+INT(X/8)+C*INT(Y/16)               :rem 36
140 CH=PEEK(SC):IFCH=32THENCH=N:POKESC,N:N=N+1:IFN
    =32THENN=33                               :rem 101
150 BY=5120+16*CH+(YAND15):POKEBY,PEEK(BY)OR2↑(7AN
    DNOTX):NEXT                                :rem 26
500 GOTO 500                                   :rem 99
1000 DATA 162,10,160,0,132,253,169,20,133 :rem 197
1010 DATA 254,152,145,253,200,208,251,230 :rem 208
1020 DATA 254,202,208,246,96                   :rem 97
10000 REM PRINT MESSAGE                       :rem 103
10002 N$="UNEXPANDED VIC GRAPHICS":N=LEN(N$)+1
                                              :rem 200
10003 FORJ=1TOLEN(N$):A=ASC(MID$(N$,J))-64:IFA<0TH
      EN12002                                  :rem 32
10004 S=J*16+5120                              :rem 83
10005 Q=0:FORK=32768+A*8TO32775+A*8:POKES+Q,PEEK(K
      ):POKES+Q+1,PEEK(K):Q=Q+2:NEXT           :rem 67
12001 POKE7680+J-1,J                           :rem 65
12002 NEXT                                     :rem 52
20000 RETURN                                  :rem 210
```

Line 100's STEP size controls the resolution of the plot. Line 97 controls the color scheme; POKE 0 for black dots on white, for example. Variations in rows or columns are compensated for.

Finally, note that RAM is short; the program already fills most of BASIC RAM, there's room for only 160 separate graphics on the screen. If you're experimenting, add this line which warns of a character shortage by changing the screen color:

145 IF N>140 THEN POKE 36879,11

3 × 3 High-Resolution Screen Editor (Unexpanded VIC-20)

This character editor allows 3 × 3 characters to be edited simultaneously. It is keyboard-controlled (* = plot point, space = don't plot) and uses a screen window enlarged to 24 × 28 characters. Of course, 24 × 24 permits 3 × 3 characters to be mapped; the extra rows allow the real-size image to be displayed.

About the program. The large screen starts lower than usual at $1C00; the character definitions start at $1800. BASIC can therefore go up to 2K bytes. Most of the character definition space is wasted; the program uses only 12 characters. The first nine (characters 0–8, starting from $1800) start as blanks and are built up by the user as asterisks or spaces are typed in; the other three are a space character, a dot, and a solid character. The dots show the positions of individual bits; some are colored differently to mark the boundaries of the nine graphics.

In general terms, lines 0–14 set the configuration of memory and the screen. Lines 15–32 blank characters 0 to 10 in the new character definition table, and also put in a dot character and a solid character. Lines 60 to 62 plot blue dots on the screen, except for every eighth dot, which is yellow. Line 70 puts spaces into the parts of the screen that {CLR} can't reach. Lines 80 to 95 POKE characters 0 to 8 to create a black square for the bottom of the screen; they can be repeated elsewhere or in different colors.

Editing starts at line 100. Cursor pointers are just as in the earlier character-editing program. The routine starting at line 200 POKEs in the on or off bit whenever * or the space bar is pressed.

If you wish to store the defined characters, PEEK the bytes from 6144 to 6215 for the relevant data.

Program 12-36. 3 × 3 Character Editor

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
1 POKE 36869,254: POKE648,28: POKE 36866,22:REM SC
  REEN AT $1C00                              :rem 73
2 POKE 55,0: POKE 56,24: CLR: REM CHRS AT $1800 &
  {SPACE}BASIC ENDS $1800                    :rem 23
6 POKE 36866,24 :REM 24 COLS                :rem 185
8 POKE 36867,56 :REM 28 ROWS                :rem 223
10 POKE 36864,4 :REM LEFT (10 IN UK)         :rem 45
12 POKE 36865,17:REM TOP{2 SPACES}(24 IN UK)
                                             :rem 49
14 POKE 36879,24:REM BLACK BORDER           :rem 112
16 SCREEN=7168                               :rem 58
```

```
18 COLOUR=37888                                :rem 140
20 CHARACTERS=6144                              :rem 78
30 FORJ=CH TO CH+87: POKE J,0: NEXT: REM CHRS 0-10
    ARE BLANK                                  :rem 248
31 POKE CH+83,24:POKECH+84,24: REM CHR 10 IS NOW D
   OT                                           :rem 54
32 FORJ=CH+88 TO CH+95: POKE J,255: NEXT: REM CHR
   {SPACE}11 IS SOLID                           :rem 40
40 PRINT "{CLR}";                                :rem 3
50 FOR J=SC TO SC+24*24-1: POKE J,10: NEXT:
   {2 SPACES}REM DOT                           :rem 223
60 FOR J=CO TO CO+24*24-1: POKE J,6:IFJ=INT(J/8)*8
   THEN POKEJ,7                                :rem 217
62 K%=(J-CO)/24:IF K%=0 OR K%=8 ORK%=16 THEN POKE
   {SPACE}J,7                                  :rem 240
65 NEXT                                        :rem 170
70 FORJ=SC+24*24 TO SC+24*29:POKE J,9:NEXT: REM PO
   KE 0-9 INTO A                               :rem 218
80 POKE SC+586,0:POKESC+587,1:POKESC+588,2: REM SQ
   UARE NEAR BOTTOM                            :rem 153
82 POKE SC+610,3:POKESC+611,4:POKESC+612,5: REM PO
   KE BLACK COLOR                              :rem 191
85 POKE SC+634,6:POKESC+635,7:POKESC+636,8: REM IN
   TO COLOR RAM                                :rem 107
90 POKE CO+586,0:POKECO+587,0:POKECO+588,0:rem 161
92 POKE CO+610,0:POKECO+611,0:POKECO+612,0:rem 127
95 POKE CO+634,0:POKECO+635,0:POKECO+636,0:rem 148
100 OF=PEEK(211)+22*PEEK(214): REM OFFSET FROM TOP
     LEFT OF SCREEN                            :rem 191
110 POKE COLOUR+OFFSET,8: REM MULTICOLOR MODE CURS
    OR                                          :rem 22
120 GETG$:IFG$=""GOTO120                        :rem 95
125 POKE COLOUR+OFFSET,6                       :rem 239
130 IF G$=" " OR G$="*" GOTO200                :rem 145
140 IF G$="{HOME}" THEN OFFSET=0: REM HOME KEY
                                               :rem 182
142 IFG$="{UP}"THEN OFFSET=OFFSET-24: REM UP
                                               :rem 243
144 IFG$="{DOWN}"THEN OFFSET=OFFSET+24: REM DOWN
                                                 :rem 6
146 IFG$="{RIGHT}"THEN OFFSET=OFFSET+1 : REM RIGHT
                                                :rem 37
148 IFG$="{LEFT}"THEN OFFSET=OFFSET-1 : REM LEFT
                                                :rem 86
150 IF OFFSET<0 THEN OFFSET=0                  :rem 187
152 IF OFFSET>24*24-1THEN OFFSET=24*24-1         :rem 7
154 POKE 214,OFFSET/22: POKE 211,OFFSET-INT(OF/22)
    *22                                        :rem 245
156 GOTO100                                    :rem 102
200 ROW%=INT(OFFSET/24): COL%=OFFSET-24*ROW%: REM
    {SPACE}FIND ROW & COL                      :rem 218
```

```
210 CHAR%=3*INT(ROW%/8)+INT(COL%/8): REM FIND CHAR
    ACTER Ø TO 8                              :rem 216
22Ø BYTE%=CHARS + 8*CHAR%+ ROW% - INT(ROW%/8)*8:RE
    M BYTE IN CHAR                            :rem 9
3ØØ IF G$="*" THEN POKE SC+OF,11:POKE BYTE%,PEEK(B
    YTE%) OR 2↑(7-COL%+INT(COL%/8)*8)        :rem 83
31Ø IF G$=" " THEN POKE SC+OF,1Ø:POKE BYTE%,PEEK(B
    YTE%)AND NOT 2↑(7-COL%+INT(COL%/8)*8)   :rem 76
32Ø GOTO1ØØ                                   :rem 95
```

3 × 3 Multicolor Mode Screen Editor (Unexpanded VIC-20)

A multicolor bitmap editor is a bit more tricky than the previous editors. But it is worthwhile experimenting with a program like the following to get the feel of VIC-20's color capabilities.

This editor allows four colors at one time. Each of the nine characters to be defined is assumed to use the same character color; and the background, border, and auxiliary colors are the same for each. The first step is to input four colors by pressing a color key, or SHIFT-color if the lighter border or auxiliary color is desired. For example, press RED, GRN, BLK, and PUR to set up these four colors. When the enlarged screen is ready, use the cursor keys and HOME to move about the screen. Press one of the four color keys (the others are ignored) to draw a block of color. The cursor is a block striped with all four colors, and therefore reasonably visible. You must remove the REM statements so the program will fit in memory.

Inevitably this mode produces pictures made of little horizontal lines. Bear in mind that three of these small lines stack up to make an approximate square. Also, remember that some colors are brighter than others—white shows up well on black, but small amounts of red are barely visible. The function keys could be programmed to change colors if you want to be able to watch the effects of different color balances and color choices. Quite striking effects of light and shade or of colored lighting can sometimes be designed.

About the program. Only 15 characters are used. The first nine—characters 0 to 8—start blank and are built up as the user selects colors. Character 9 is a dot, and character 10 a blank—this takes the same color as the background.

The next four characters are designed around the peculiarities of multicolor mode. One character is made up of bit pattern 01010101: This appears as a solid character in the border color, because the bit-pair 01 is interpreted as the border color in multicolor mode. The next, CHR$(12), is made of 10101010 bytes, 170 in decimal, which take the character color; and there is also the auxiliary color of 11111111. CHR$(14) is made of bit pattern 11100100; this gives a vertical stripe in each color and acts as a cursor.

This program is similar to the previous one. However, line 105 converts cursor shifts to the leftmost of two possible positions, since in multicolor mode adjacent bits are counted together. A second difference is the four variables—BACKGD, BODER, CELL, and AUX—that store the four colors which are compared (lines 10040–10070) with keystrokes so the color's status can be determined.

Program 12-37. 3 × 3 Multicolor Character Editor

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
1 POKE 55,Ø:POKE56,24:CLR: REM TOP OF BASIC $18ØØ
                                            :rem 156
2 PRINT "BACKGROUND?";: GOSUB 5ØØØØ: BA=G-1:REM IN
  PUT FOUR COLORS                           :rem 1Ø3
3 PRINT "{4 SPACES}BORDER?";: GOSUB 5ØØØØ: BO=G-1:
  REM KEYS; SHIFT-COLOR                      :rem 87
4 PRINT " CHARACTER?";: GOSUB 5ØØØØ: CE=G-1:REM KE
  YS; SHIFT-COLOR                            :rem 3Ø
5 PRINT " AUXILIARY?";: GOSUB 5ØØØØ: AU=G-1:REM GI
  VES ORANGE ETC.                           :rem 251
6 POKE 36879,8+BO+16*BA: REM BORDER, BACKGROUND
                                            :rem 189
7 POKE36878,16*AU: REM & AUXILIARY COLORS :rem 151
8 POKE 36869,254: POKE648,28: REM CHRS $18ØØ & SCR
  EEN $1EØØ                                 :rem 25Ø
9 POKE 36866,24: REM 24 COLUMNS             :rem 172
1Ø POKE 36867,56: REM 28 ROWS                :rem 8
11 POKE 36864,4: REM LEFT MARGIN (1Ø IN UK)
                                            :rem 236
12 POKE 36865,16: REM TOP MARGIN (24 IN UK)
                                            :rem 238
14 PRINT "{CLR}";: SCREEN=7168: COLOUR=37888: CHAR
   ACTERS=6144                              :rem 148
3Ø FOR J=CH TO CH+87: POKE J,Ø: NEXT: REM CHARS Ø-
   1Ø BLANK                                  :rem 97
31 POKE CH+75,12:POKE CH+76,12: REM CHAR 9 IS NOW
   {SPACE}DOT                                :rem 75
32 FOR J=CH+88 TO CH+95: POKE J,85: NEXT: REM CHAR
   S 11,12,13,14                             :rem 38
33 FORJ=CH+96 TO CH+1Ø3: POKE J,17Ø: NEXT: REM ARE
    VERTICAL STRIPES                         :rem 8Ø
34 FORJ=CH+1Ø4TO CH+111: POKE J,255: NEXT: REM WHI
   CH GIVE SOLID                             :rem 55
35 FORJ=CH+112TO CH+119: POKE J,228: NEXT: REM COL
   OR IN MULTICOLOR                          :rem 7Ø
5Ø FOR J=SC TO SC+24*24-1: POKE J,9 :NEXT: REM DOT
   S FILL SCREEN                            :rem 241
6Ø FOR J=CO TO CO+24*24-1: POKE J,CE+8: REM OF COR
   RECT COLOR                                :rem 4Ø
65 NEXT                                     :rem 17Ø
7Ø FOR J=SC +24*24 TO SC+24*29: POKE J,1Ø: NEXT: R
   EM BLANK BOTTOM                          :rem 255
8Ø POKE SC+586,Ø: POKE SC+587,1: POKE SC+588,2:REM
    POKE SQUARE OF                           :rem 98
82 POKE SC+61Ø,3: POKE SC+611,4: POKE SC+612,5:REM
    CHARACTERS INTO                         :rem 2Ø6
85 POKE SC+634,6: POKE SC+635,7: POKE SC+636,8:REM
    THE BOTTOM OF                            :rem 29
```

```
90 POKE CO+586,8+CE: POKE CO+587,8+CE: POKE CO+588
   ,8+CE:REM SCREEN                              :rem 176
92 POKE CO+610,8+CE: POKE CO+611,8+CE: POKE CO+612
   ,8+CE:REM WITH                                 :rem 10
95 POKE CO+634,8+CE: POKE CO+635,8+CE: POKE CO+636
   ,8+CE:REM COLOR                                :rem 98
100 OF=PEEK(211)+22*PEEK(214): REM OFFSET FROM TOP
     LEFT                                        :rem 106
105 IF INT(OF/2)*2<>OF THEN OF=OF-1              :rem 182
110 P=PEEK(SC+OF): POKE SC+OFFSET,14             :rem 109
120 GET G$:IF G$="" GOTO 120                      :rem 95
125 POKE SCREEN+OFFSET,P                         :rem 245
126 G=ASC(G$)                                    :rem 176
127 IF G>48 AND G<57 THEN G=G-49: GOTO 200:rem 117
128 IF G>32 AND G<41 THEN G=G-25: GOTO 200 :rem 98
140 IF G$="{HOME}" THEN OF=0: REM HOME     :rem 155
142 IF G$="{UP}" THEN OF=OF-24: REM UP     :rem 143
144 IF G$="{DOWN}" THEN OF=OF+24: REM DOWN:rem 162
146 IF G$="{RIGHT}" THEN OF=OF+2: REM RIGHT
                                                 :rem 194
148 IF G$="{LEFT}" THEN OF=OF-1: REM LEFT :rem 242
150 IF OFFSET<0 THEN OFFSET=0: REM CORRECTION IF M
    OVE                                          :rem 151
152 IF OFFSET>24*24-1 THEN OFFSET=24*24-1: REM OFF
     SCREEN EDGE                                 :rem 213
154 POKE 214,OFFSET/22: POKE 211,OFFSET-INT(OF/22)
    *22: REM CURSOR                              :rem 241
156 GOTO 100                                     :rem 102
200 ROW%=INT(OFFSET/24): COL%=OFFSET-24*ROW%
                                                 :rem 159
210 CHAR%=3*INT(ROW%/8)+INT(COL%/8): REM FIND CHAR
    ACTER 0 TO 8                                 :rem 216
220 BYTE%=CHARS + 8*CHAR%+ ROW% - INT(ROW%/8)*8:RE
    M BYTE IN CHAR                                 :rem 9
10040 IFG=BACKGD THEN B=0:POKESC+OF,10:POKESC+OF+1
      ,10:REM SEARCH                              :rem 74
10050 IFG=BODER THEN B=1:POKE SC+OF,11:POKESC+OF+1
      ,11:REM KEYPRESS                           :rem 222
10060 IFG=CELL THEN B=2:POKE SC+OF,12:POKESC+OF+1,
      12:REM COLOR &                             :rem 197
10070 IFG=AUX{3 SPACES}THEN B=3:POKE SC+OF,13:POKE
      SC+OF+1,13:REM POKE AND                    :rem 244
10080 P=(7-COL%+INT(COL%/8)*8):IFINT(P/2)*2<>PTHEN
      P=P-1:REM PLOT                              :rem 20
10100 POKE BYTE%,(PEEK(BYTE%)AND(255-3*2↑P))OR B*2
      ↑P: REM IN 3*3                              :rem 34
20000 GOTO 100                                   :rem 188
50000 GET G$: IF G$="" GOTO 50000: REM CONVERTS CO
      LOR KEY                                     :rem 29
50005 G=VAL(G$): REM OR SHIFTED-COLOR           :rem 143
```

```
50010 IF G=0 THEN G=ASC(G$)-24: REM INTO A VALUE 1
      -16                                      :rem 237
50015 PRINT G: RETURN                          :rem 233
```

Full-Screen High-Resolution Screen Editor (Expanded VIC-20)

Program 12-38 allows plotting on a large screen (20 columns × 24 rows) in high resolution and in color. The program is joystick-controlled. Dots are plotted in any of the eight joystick directions, or erased if the delete mode is on. The joystick button toggles the delete feature on and off. If the joystick and button are active together, the plotting position moves without affecting the present graphics. Finally, pressing any color key (1 to 8) sets the current character's color. Double-sized characters are used.

The program requires a 3K, 8K, or 16K expansion cartridge or the *Super Expander*. With 3K expansion memory or the *Super Expander*, add this line before saving:

10 POKE 55,0: POKE 56,16: CLR

With 8K or 16K expansion memory, type POKE 642,32: SYS 64818 before loading and running.

There are severe limitations on the use of color with high-resolution graphics. The section on the VIC chip's reverse bit shows how each color change is confined to a rectangular area.

Pictures can be saved to disk or tape with the methods explained in Chapter 6. (Note, however, that saving color RAM to tape isn't straightforward.)

Program 12-38. Full-Screen Joystick-Controlled Plotter in High-Resolution Mode

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
20 POKE36879,8+16: REM BLACK BORDER        :rem 209
22 POKE36869,204:POKE648,16: REM CHRS, SCREEN AT 4
   096                                     :rem 165
24 POKE36866,20: REM 20 COLS               :rem 225
26 POKE36867,25: REM 12 ROWS; DOUBLE-SIZE   :rem 98
40 FOR J=320 TO 343: READ G: POKE J,G: NEXT: SYS 3
   20                                      :rem 174
42 FOR J=0 TO 239                           :rem 69
44 POKE 4096+J,J+16: REM POKE CHRS 16 TO 255
                                            :rem 10
46 POKE 37888+J,0: REM COLOR RAM SET BLACK :rem 66
48 NEXT                                    :rem 171
100 POKE 37154,127: REM READ JOYSTICK       :rem 52
110 PE=PEEK(37152): POKE 37154,255          :rem 17
120 R=(PE AND 128)/128: REM RIGHT          :rem 220
130 PE=PEEK(37137)                         :rem 225
140 U=(PE AND 4)/4: REM UP                  :rem 58
150 D=(PE AND 8)/8: REM DOWN               :rem 197
160 L=(PE AND 16)/16: REM LEFT              :rem 31
170 B=(PE AND 32)/32: REM BUTTON           :rem 195
200 IF B=0 THEN DEL=1-DEL: REM BUTTON TOGGLES DELE
    TE                                       :rem 6
```

```
210 IF B=Ø AND U*D*L*R=Ø THEN MOV=1: REM BUTTON +
    {SPACE}JOY                                :rem 1Ø8
22Ø IF B=1 THEN MOV=Ø: REM MOVES WITHOUT PLOTTING
                                              :rem 174
3ØØ K=PEEK(197): IF K=64 GOTO 34Ø: REM READ KEYBOA
    RD AND                                    :rem 27
31Ø IF K<6{2 SPACES}THEN CE=2*K: REM CONVERT COLOR
     KEY                                      :rem 34
32Ø IF K>55 THEN CE=2*K-111: REM TO NUMBER Ø TO 7
                                                :rem 6
33Ø POKE 37888+SCR,CE                         :rem 158
34Ø IF L=Ø THEN X=X-1: IF X<Ø THEN X=Ø: REM UPDATE
     X AND Y                                  :rem 63
35Ø IF R=Ø THEN X=X+1: IF X>159 THEN X=159: REM CO
    ORDINATES                                 :rem 24
36Ø IF U=Ø THEN Y=Y-1: IF Y<Ø THEN Y=Ø    :rem 233
37Ø IF D=Ø THEN Y=Y+1: IF Y>191 THEN Y=191:rem 175
5ØØ SCREENCHR = 2Ø*INT(Y/16) + INT(X/8)     :rem 76
51Ø ROW = Y AND 15: BIT= X AND 7            :rem 21
52Ø CHAR=4352 + 16*SCR + ROW: REM STARTS 4Ø96+256
    {SPACE}(CHR 16)                           :rem 182
53Ø PE=PEEK(CH): REM SAVE IF LINE 56Ø NEEDS
                                              :rem 121
54Ø POKE CH,PEEK(CH) OR 2↑(7-BIT): REM PUTS IN THE
     NEW BIT                                  :rem 24Ø
55Ø IF DEL THEN POKE CH,PEEK(CH) AND NOT 2↑(7-BIT)
                                              :rem 252
56Ø IF MOVE THEN POKE CH,PE: REM RESTORE VALUE IF
    {SPACE}NO PLOT                            :rem 53
57Ø GOTO 1ØØ                                  :rem 1Ø2
1ØØØ DATA 169,16,17Ø,133,252,169,Ø,168,133: REM CL
     EAR CHAR DEFNS                           :rem 35
1Ø1Ø DATA 251,145,251,153,Ø,148,2ØØ,2Ø8    :rem 1Ø8
1Ø2Ø DATA 248,23Ø,252,2Ø2,2Ø8,243,96       :rem 231
```

Full-Screen Multicolor Mode Screen Editor (Expanded VIC-20)

Like the last program, this allows a picture to be developed on a fully mapped 24 × 20 screen. However, this version uses multicolor mode. It is also keyboard-controlled. Background, border, and auxiliary colors are input at the start and remain the same during the program; however, local character colors can be changed. The keyboard controls are simple: Type in whichever colors you want whenever you wish to select a new color, using the SHIFT key for the lighter background and auxiliary colors. If the color is the border, background, or auxiliary color, the appropriate bit pattern will be set for this. Any other color will be treated as a character color, and if new, will replace whatever character color was previously in use. The space bar toggles between plotting and moving without plotting.

The program requires a 3K, 8K, or 16K expansion cartridge or the *Super Expander*. With 3K expansion memory or the *Super Expander*, add this line before saving:

10 POKE 55,0: POKE 56,16: CLR

With 8K or 16K expansion memory, type POKE 642,32: SYS 64818 before loading and running.

Program 12-39. Full-Screen Keyboard-Controlled Plotter in Multicolor Mode

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
20 PRINT "BACKGROUND?";: GOSUB 50000: BA=G-1
                                             :rem 219
22 PRINT "{4 SPACES}BORDER?";: GOSUB 50000: BO=G-1
                                             :rem 201
24 PRINT " AUXILIARY?";: GOSUB 50000: AU=G-1
                                             :rem 202
26 PRINT " CHARACTER?";: GOSUB 50000: CE=G-1
                                             :rem 147
30 POKE 36879,8+BO+16*BA: REM SET BORDER, BACKGND
                                             :rem 224
32 POKE 36878,16*AU: REM AUXILIARY COLOR    :rem 76
34 POKE 36869,204: POKE648,16: REM SCREEN & CHARS
   {SPACE}AT $1000                           :rem 245
36 POKE 36866,20: REM 20 COLUMNS             :rem 212
38 POKE 36867,25: REM 12 DOUBLE-SIZE ROWS    :rem 42
40 FOR J=320 TO 343: READ G: POKE J,G: NEXT:SYS 32
   0                                         :rem 174
42 FOR J=0 TO 239: POKE 4096+J,J+16: REM SCREEN CH
   ARS 16 & UP                               :rem 127
44 POKE 37888+J,CE+8: REM DEFAULT CELL COLOR
                                             :rem 247
46 NEXT                                      :rem 169
100 GET G$: G=ASC(G$+CHR$(0))                :rem 218
200 IF G>48 AND G<57 THEN G=G-49: GOSUB 40000: GOT
    O 500                                     :rem 30
210 IF G>32 AND G<41 THEN G=G-25: GOSUB 40000: GOT
    O 500                                     :rem 11
220 IF G$=" " THEN MOVE = 1-MOVE:REM SPACE TOGGLES
     PLOT ON/OFF                              :rem 204
230 IF G$ = "{RIGHT}" THEN X=X+2: IF X>159 THEN X=
    159: REM RIGHT                            :rem 163
240 IF G$ = "{LEFT}" THEN X=X-2: IF X<0 THEN X=0:
    {SPACE}REM LEFT                           :rem 243
250 IF G$="{UP}" THEN Y=Y-1: IF Y<0 THEN Y=0: REM
    {SPACE}UP                                 :rem 101
260 IF G$ = "{DOWN}" THEN Y=Y+1: IF Y>191 THEN Y=1
    91: REM DOWN                               :rem 79
500 SCREENCHR = 20*INT(Y/16) + INT(X/8)        :rem 76
510 ROW = Y AND 15: BIT=6-(X AND 7)           :rem 201
520 CHAR=4352 + 16*SCR + ROW: PE=PEEK(CH) :rem 148
530 POKE CH,(PE AND (255-3*2↑BIT)) AND NOT B*2↑BIT
     :REM FLASH...                            :rem 210
```

```
54Ø POKE CH,(PE AND (255-3*2↑BIT)) OR B*2↑BIT: REM
     AND PLOT                                  :rem 2Ø2
55Ø IF MOVE=1 THEN POKE CH,PE: REM REPLACE UNCHANG
    ED                                          :rem 31
56Ø GOTO 1ØØ                                   :rem 1Ø1
1ØØØ DATA 169,16,17Ø,133,252,169,Ø,168,33: REM CLE
     AR CHAR DEFNS                             :rem 242
1Ø1Ø DATA 251,145,251,153,Ø,148,2ØØ,2Ø8        :rem 1Ø8
1Ø2Ø DATA 248,23Ø,252,2Ø2,2Ø8,243,96           :rem 231
4ØØØØ IF G=BACKGD THEN B=Ø: RETURN: REM SEARCH FOR
       COLOR                                   :rem 213
4ØØ1Ø IF G=BODER THEN B=1: RETURN              :rem 1Ø9
4ØØ2Ø IF G=CELL THEN B=2: RETURN                :rem 35
4ØØ3Ø IF G=AUX THEN B=3: RETURN                :rem 243
4ØØ4Ø CE=(G AND 7) + 8: B=2: REM NEW DEFAULT COLOR
                                                :rem 57
4ØØ5Ø POKE 37888+2Ø*INT(Y/16) + INT(X/8),CE: REM P
      UT INTO COLOR RAM                        :rem 166
4ØØ6Ø RETURN                                   :rem 218
5ØØØØ GET G$: IF G$ = "" GOTO 5ØØØØ: REM CONVERT C
      OLOR KEY                                 :rem 2Ø2
5ØØØ5 G=VAL(G$)                                 :rem 29
5ØØ1Ø IF G=Ø THEN G=ASC(G$)-24                  :rem 18
5ØØ15 PRINTG:RETURN                            :rem 233
```

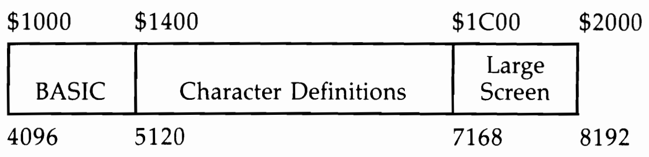
Defining Your Own Lettering

Large Lettering

Double-sized text, which is clear and readable even at a distance, is easy to achieve by redefining ROM characters as double-sized characters in RAM.

Several memory configurations are possible. We'll use the one shown in Figure 12-17, which applies to the unexpanded VIC and allows use of a large screen of, say, 26 columns × 16 rows of text. There's room for 1K of BASIC, which could be messages printed on the screen.

Figure 12-17. Large Lettering Memory Map (Unexpanded VIC)



This layout allows a generous 2K for the definitions of the characters. Since these are double-sized, they take 16 bytes each, so we've room for 128 characters (2048/16 =128). Try this short program for the unexpanded VIC which moves the lowercase set, excluding reverse characters, into RAM. (Note that the parameters in line 20 may need adjusting to square up your TV picture.)

Program 12-40. Large Lettering

```
10 POKE 36867,33:POKE 36866,154:REM 26 COLUMNS, 16
    DOUBLE-SIZE ROWS
20 POKE 36865,21:POKE 36864,8:REM SQUARE UP PICTUR
   E
30 POKE 56,20:CLR:POKE 36869,253:REM BASIC END=CHR
    START=$1400
40 FOR J=5120 TO 7168:POKE J,PEEK(J/2+32256):NEXT:
   REM EACH TWICE
```

In line 40, whenever J is odd, J/2 has an odd half which is ignored, so the same PEEK value is reused. The value 32256 is $8800 minus 5120/2.

All the normal keyboard characters print normally, but in giant characters. Colors behave normally, and the background and border can be POKEd normally in 36879, but reverse characters aren't defined. Also editing behaves abnormally—RETURN, for example, doesn't allow for the wider screen. This difficulty can be evaded by putting semicolons after strings so that RETURN isn't used, and juggling spaces and lengths to fit. This sort of thing is sometimes recommended for titling videos and so on.

Since there are plenty of characters to play with, we could include reverse characters or characters of our own. For example, we might move reversed uppercase text to replace the graphics characters in text mode. Then SHIFT-A prints as a reversed A, and SHIFT-space appears as a block. This is useful for messages, and easy to use, since there's no need for a conversion algorithm. These lines replace line 40:

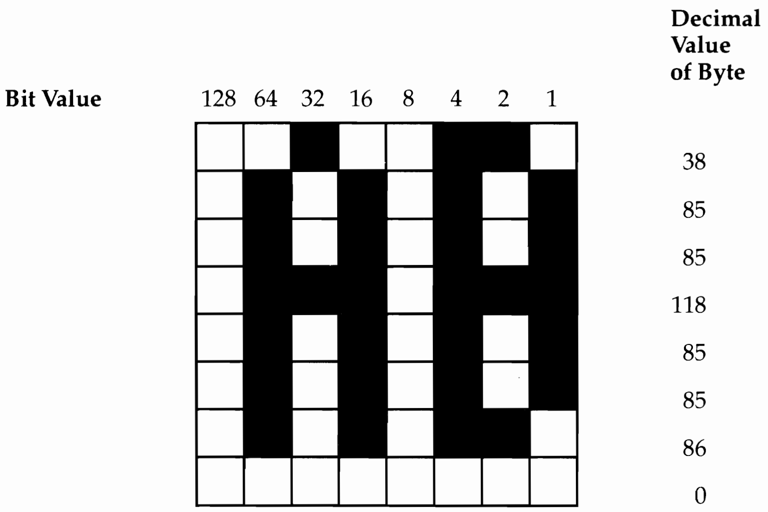
```
40 FOR J=5120 TO 6159: POKE J, PEEK (J/2 + 30208): NEXT
50 FOR J=6160 TO 7168: POKE J, PEEK (J/2 + 30720): NEXT
```

Small Lettering

Some commercial software has 40 columns of text with the VIC-20. How is this possible? Since a full character is 8 × 8 dots, a 40-column screen will require characters that are 4 × 8 dots. Obviously, though, they need to be readable when put next to each other, so in practice a full set of characters like this needs to be 3 × 7 dots. Figure 12-18 shows an uppercase A and B fitted into an 8 × 8 grid.

Three-dot wide lettering is surprisingly readable in both upper- and lowercase, although some letters—notably capital N—are impossible to approximate, and others, such as M, can only be loosely approached. Four-dot wide letters, in fact, are probably the best size visually. However, text has to be separated into blocks of eight characters, which are then redefined as a five-character set for display, which is tiresome. In addition, only 20 rows of these characters will fit into a VIC-20, whereas the narrower three-dot characters can fill 40 columns × 24 rows, a very useful size, almost compatible with PET/CBM and 64 machines. The rest of this section will therefore deal with three-dot wide graphics only.

Figure 12-18. Small Lettering: A and B



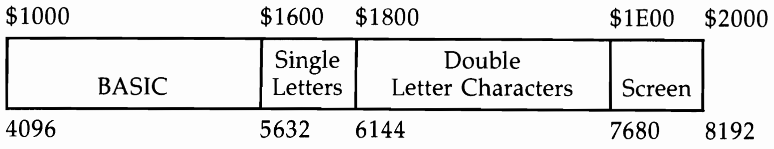
Narrow Lettering with 8 × 8 Characters (Unexpanded VIC-20)

Can we display lettering like this in the common 256-character format? It's possible to get a partial solution, but since we're forced to consider pairs of letters, the arithmetic is against us. The full uppercase alphabet set, plus space character and a period, can be combined into 784 combinations (28 * 28). So only one-third of all possible combinations could be covered. However, this remains a feasible approach since several letters and letter combinations are rare and could be ignored. And spare characters can be allocated letters when required.

The demonstration program uses an incomplete graphics technique; it simply converts a BASIC string of text into characters, starting at the beginning, and not attempting to check whether a combination already exists. Therefore, it is impossible to fill a full screen. At most, 255 8 × 8 characters are available (a space character has to be retained, making the full 256), and the screen can't be more than about half-full.

Its memory map is slightly more complex than some we've used. There are two areas of character storage. $1600–$17FF stores individual letter definitions.

Figure 12-19. Thin Lettering Memory Map



Character definitions actually start at $1800. When a pair of letters is to be displayed, the bits for the two characters are combined and put in the character definition area. A maximum of 198 new pairs can be defined, enough for about nine complete lines of text. I've used 26 letters and 8 punctuation symbols; these occupy 272 bytes (34*8). BASIC could end further up than $1600, but for relative convenience I've started the individual characters at a page boundary and kept the usual ASCII numbering of punctuation symbols.

To run the demonstration, first enter and run the loader, which puts individual definitions into $1600-$17FF, occupying the low four bits. Lines 330 and 340 do this for letters, separating the two letters as they're stored in DATA statements, and lines 420 to 450 do the same for punctuation.

Once the loader has been run, run the demonstration. GOSUB 500 prints any string TEXT$ (must have an even number of characters) in black at a position in the screen referred to as OFFSET. OFFSET = 0 means printing starts at the top left of the screen, OFFSET = 22 starts one line down, and so on.

Program 12-41. Thin Lettering BASIC Loader

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
1ØØ DATA 32,46,44,63,58,59,34,33           :rem 39
11Ø FOR J=1 TO 8: READ X: CH(J)=X: NEXT    :rem 4Ø
3ØØ FOR J=564Ø TO 5639 +26*8               :rem 63
31Ø FOR K=Ø TO 7                           :rem 13
32Ø READ X                                  :rem 9
33Ø POKE J+K, X/16                        :rem 159
34Ø POKE J+K+8,X AND 15                   :rem 166
35Ø NEXT K                                 :rem 34
36Ø J=J+15                                :rem 251
37Ø NEXT J                                 :rem 35
4ØØ FOR J=1 TO 8                           :rem 14
41Ø FOR K=Ø TO 7                           :rem 14
42Ø READ X                                 :rem 1Ø
43Ø POKE 5632+CH(J)*8+K, X/16             :rem 217
44Ø POKE 5632+CH(J+1)*8+K, X AND 15       :rem 217
45Ø NEXT K                                 :rem 35
46Ø J=J+1                                 :rem 199
47Ø NEXT J                                 :rem 36
5ØØ FOR J =64ØØ TO 64Ø7:POKEJ,Ø:NEXT      :rem 2Ø1
61Ø POKE 648,3Ø                           :rem 247
62Ø POKE 36866,15Ø                        :rem 15Ø
63Ø POKE 56,22: CLR                       :rem 222
1ØØØ DATA 38,85,85,118,85,85,86,Ø,38,85,69,69,69,8
     5,38,Ø:REM ABCD                       :rem 27
1Ø1Ø DATA 119,68,68,119,68,68,116,Ø,37,85,69,87,85
     ,85,37,Ø:REM EFGH                    :rem 135
1Ø2Ø DATA 115,33,33,33,33,37,114,Ø,84,84,1ØØ,1ØØ,8
     4,84,87,Ø:REM IJKL                   :rem 137
1Ø3Ø DATA 86,117,117,117,85,85,85,Ø,38,85,85,86,84
     ,84,36,Ø:REM MNOP                    :rem 158
```

```
1040 DATA 38,85,85,86,85,117,37,16,55,82,66,34,18,
     82,98,0:REM QRST                         :rem 128
1050 DATA 85,85,85,85,85,82,34,0,85,85,82,114,117,
     117,85,0:REM UVWX                        :rem 183
1060 DATA 87,81,82,34,36,36,39,0:REM YZ       :rem 253
1070 DATA 0,0,0,0,0,6,6,0,2,5,1,2,34,32,66,0:REM S
     PACE.,?                                   :rem 87
1080 DATA 0,0,34,0,0,34,2,4,82,82,82,2,0,0,2,0:REM
      :;"!                                    :rem 115
```

Program 12-42. Thin Lettering Demonstration

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 POKE36869,254                              :rem 103
100 PRINT"{CLR}"                              :rem 211
400 TEXT$="I AM BESIDE A DARK FOREST; A RAPID STRE
    AM{3 SPACES}RUNS NEARBY ... "             :rem 130
401 TE$=TE$+"FIERCE ELECTRIC STORMS RAGE.TINY FROG
    S HOP AROUND."                            :rem 140
402 TE$=TE$+" FAST MOVING CLOUDS{3 SPACES}OBSCURE
    {SPACE}THE MOON. A CLOSED                 :rem 184
403 TE$=TE$+" COFFIN LIES BY A SMALL HUT.  :rem 241
410 OFFSET=176: GOSUB 500                     :rem 134
420 TE$="I SEE NOTHING UNUSUAL.": OF=352: GOSUB 50
    0                                          :rem 97
430 TE$="WHAT SHALL I DO NOW?": OF=440: GOSUB 500
                                              :rem 127
499 GOTO499                                   :rem 133
500 Q=0:{2 SPACES}FOR J=1TOLEN(TE$)STEP2       :rem 42
520 A1=ASC(MID$(TE$,J))-64:IFA1<0THENA1=A1+64
                                               :rem 19
521 A2=ASC(MID$(TE$,J+1))-64:IFA2<0THENA2=A2+64
                                              :rem 116
530 FORK=0TO7:P1=PEEK(5632+8*A1+K):P2=PEEK(5632+8*
    A2+K)                                     :rem 121
560 POKE6144+8*N+K,P1*16+P2:NEXTK             :rem 153
600 POKE 7680+OF+Q,N                           :rem 80
610 POKE 38400+OF+Q,0                          :rem 93
614 Q=Q+1: N=N+1:IF N=32THENN=33               :rem 30
615 NEXT                                      :rem 219
620 RETURN                                    :rem 120
```

Narrow Lettering with 8 × 16 Characters (Expanded VIC-20)

The only way to fill an entire screen with three-dot wide characters is with double-sized characters, which map the entire screen. Each double-sized character contains up to four three-dot wide characters, one in each corner. As we've seen, 20 × 12 of these characters can be fit in, giving in effect 40 columns of text × 24 rows. This of course imposes a software overhead—the method can be used only with memory expansion.

One commerical product, *Superscreen,* fits the unexpanded VIC area. When run, it stores its version of the VIC's characters at $2000 and beyond, reconfigures the memory and stores characters 16, 17, 18, ...255 in screen memory, clears the character definitions, and changes two vectors (for input and output of characters) so that these intercept machine language routines at about $2800. These routines handle the work of calculating which quadrants of which double-sized characters require a character to be ORed in.

The start-of-BASIC pointers have to be at the end of all this. If the object is to write or use 40-column BASIC programs, BASIC must be higher than usual, with perhaps 13K of memory left from a 16K expansion. There are inevitably problems with color, screen, and other POKEs, but ordinary monochrome BASIC can work satisfactorily.

Other programs, notably for use with modems, are intended only to be used for a specific function, so problems of compatibility with other programs are unimportant.

Using the Raster Scan and the Interrupt

The VIC always knows which screen line is being scanned. This information can be used to obtain some effects that might otherwise be impossible to achieve. These effects have one thing in common: The screen can be treated as though divided horizontally into two or more independent parts. As we'll see, there are two programming approaches: the easy way, which is not very flexible, and the more difficult method, using VIC's interrupt to regularly modify the screen as each scan periodically occurs, which is very flexible. The second method is potentially very powerful, opening up such possibilities as mixing text on the bottom half of the screen with graphics on top.

Easy Example with the Raster Scan

The raster line is stored in nine bits; we'll ignore the smallest bit and use only location $9004. This is about the shortest demonstration program possible. (This is a US version; for Britain change the 128 in line 10 to 144.)

Program 12-43. Raster

```
10 DATA 169,128,133,0,169,40,141,15,144,165
20 DATA 0,205,4,144,208,251,169,56,141,15,144
30 DATA 173,4,144,208,251,198,0,208,230,96
40 FOR J=828 TO 858:READ X:POKEJ,X:NEXT
50 SYS 828
```

A background color change scrolls up the screen, as the program waits for the raster line to take a progressively reducing value. The assembly listing looks like this:

```
        LDA  #$80    ;IN BRITAIN #$90
        STA  $00     ;LOCATION $00 COUNTS DOWN FROM 144 TO 0
LOOP    LDA  #$28
        STA  $900F   ;RED BACKGROUND, BLACK BORDER
        LDA  $00
```

```
WAIT  CMP  $9004
      BNE  WAIT   ;WAIT UNTIL RASTER = CONTENTS OF $00
      LDA  #$38
      STA  $900F  ;CYAN BACKGROUND, BLACK BORDER
WAI2  LDA  $9004
      BNE  WAI2   ;AWAIT TOP OF SCREEN
      DEC  $00
      BNE  LOOP   ;REPEAT UNTIL TOP OF SCREEN REACHED
      RTS
```

Add line 60 POKE 833, RND(1)*256 OR 8: POKE 845, RND(1)*256 OR 8: GOTO 50 for a repeating display in random colors.

What other effects can this method give? Part of the screen can use lowercase, and part uppercase, or user-defined graphics can coexist with normal text, if location $9005 is altered with a loop as above. The screen dimensions can be changed. Double-sized and normal characters can be mixed.

The point is that almost the entire processing time of the 6502 chip is spent waiting for the raster scan. We'll now see how we can do better.

More Complex Example with the Raster Scan and Interrupt

The TV screen is scanned about each thirtieth of a second, and every scan interleaves the previous scan. Interrupts and screen scans aren't normally synchronized, but since the interrupt rate is programmable, we can arrange interrupts to coincide with any screen scan position. This makes it possible for one horizontal section of the screen to be processed differently from another section. This example program changes the border and background colors, by changing location 36879, but other possibilities include changing the entire character set, changing the screen size, and so on. Ordinary text could be used at the bottom, and graphics at the top.

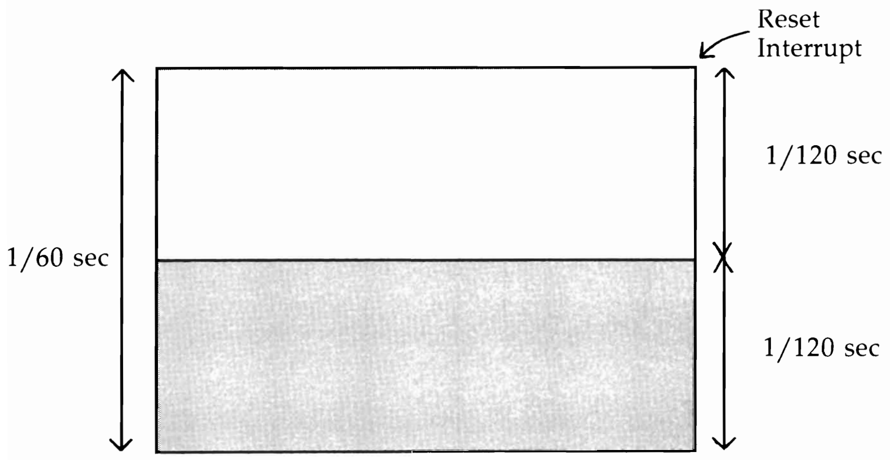
Program 12-44. Split Screen Demonstration: BASIC Loader

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
2 FOR J=828 TO 881: READ P: POKE J,P: NEXT:rem 212
4 SYS 828                                 :rem 213
6 PRINT"{CLR}POKE 849 AND 875 TO{4 SPACES}CONTROL
  {SPACE}COLORS                            :rem 89
8 PRINT "POKE 862 TO CONTROL{4 SPACES}POSITION OF
  {SPACE}SPLIT;{4 SPACES}867 FINETUNES    :rem 161
9 REM * BRITAIN: REPLACE 32 AND 200 (LINE 13) BY 4
  3 AND 66 *                              :rem 152
10 DATA 120,169,3,141,21,3,169,73,141,20  :rem 155
11 DATA 3,88,96,162,0,240,11,206,74,3,169 :rem 221
12 DATA 94,141,15,144,76,21,235,173,4,144,208
                                          :rem 163
13 DATA 251,169,32,141,37,145,169,200,141,36
                                          :rem 114
14 DATA 145,238,74,3,169,58,141,15,144     :rem 86
15 DATA 76,191,234                        :rem 121
```

How does this work? In Figure 12-20, the situation *after* the program is run shows interrupts at about twice the screen scan frequency. Interrupts alternate between the value they put into 36879. Also, each second interrupt waits for the TV to scan the top line, then reloads the timer with the same constant value. This means the display is very stable. It's not usually acceptable to simply hope the VIA will generate interrupts at exactly the right frequency to match the TV's display.

Figure 12-20. Screen Scan Frequency



The dividing line cannot be moved below about the halfway point. To do this satisfactorily, more interrupts are needed, perhaps four per screen scan. Try POKEing 862 with 64, twice its normal value; interrupts now coincide with the screen refresh, so the two patterns of color alternate, and therefore overlap on the screen. If the colors are very different, there'll be a noticeable flashing effect, but if they're similar, the flashing is reduced, though usually still visible.

POKE 849,9: POKE 875,24 shows a black and white dividing line; if the boundary isn't still, move it by changing the value in 867.

Loading or saving to tape or disk temporarily turns off the split, because the interrupt is used by the system.

Below is the assembly language. Note that JMP $EABF continues the interrupt just as usual; JMP $EB15 continues without updating the clock or scanning the keyboard, and this insures that the clock works normally. $034A holds a flag, alternately 0 and 1, to track the state of the interrupts. US values to set the VIA timer are about 4239, 8481, and other multiples; in Britain, 5536, 11074, and so on. There is a slight delay at each complete picture when the screen top is tested for, but generally of course routines like this will be adequately transparent.

```
033C  SEI           ;SYS 828 RESETS
033D  LDA  #$03     ;THE INTERRUPT
033F  STA  $0315    ;VECTOR TO
0342  LDA  #$49     ;$0349.
```

```
0344  STA  $0314
0347  CLI
0348  RTS
0349  LDA  #$01   ;TEST FOR 1ST
034B  BEQ  $0358  ;OR ALTERNATE
034D  DEC  $034A  ;INTERRUPT
0350  LDA  #$5E
0352  STA  $900F  ;SET BORDER, BCKGND
0355  JMP  $EB15  ;CONTINUE INTERRUPT
0358  LDA  $9004  ;1ST INTERRUPT
035B  BNE  $0358  ;WAIT FOR SCREEN TOP
035D  LDA  #$20   ;SET VIA TIMER
035F  STA  $9125  ;TO ABOUT 1/120 SEC.
0362  LDA  #$C8
0364  STA  $9124
0367  INC  $034A  ;NEXT INTERRUPT
036A  LDA  #$3A   ;SET BORDER, BCKGND
036C  STA  $900F
036F  JMP  $EABF  ;CONTINUE INTERRUPT
```

Interrupts and Repetition

Any repetitive task can be carried out with an extension added to the interrupt routine. Graphics examples include such things as clocks and countdown indicators when accurate timing is wanted. Pictorial, numerical, or digital clockfaces can be continually updated.

Motion

By *motion* I mean simulating movement by replacing characters with duplicate characters PRINTed nearby. Moving characters one character step in any direction is relatively straightforward. This sort of animation is inevitably jerky. An obvious improvement is to use intermediate characters made up of two halves of the original. For example, using ordinary BASIC:

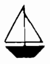
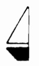
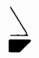
Program 12-45. Motion

```
10 M$="{C}{F}"
20 PRINT"{2 SPACES}";
30 FOR J= 1 TO 20
40 FOR K= 1 TO 30:NEXT
50 PRINT"{2 LEFT} {B}";
60 FOR K= 1 TO 30:NEXT
70 PRINT"{LEFT}"M$;:NEXT
```

Lines 40 and 60 control the speed of the checked block across the screen. This relies on the fact that the check pattern can be initiated by another pair of graphics. Smoothness of movement is improved over the simpler method. For general movement, exactly the same principle can be followed, but you'll need user-defined graphics. The character editor presented earlier is an easy way to define these. Obviously, for horizontal movement each original character will need an additional pair of characters, each holding half, as in this diagram:

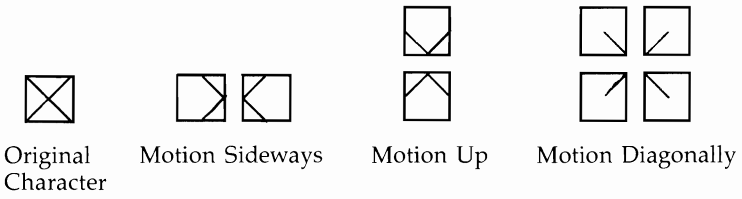
Figure 12-21. Half-Character Motion

Boat Character Diagrams

 is one character, then "  " and "  " will give smoother movement.

This method becomes unwieldy for a lot of movement in many directions, because each single character needs eight more characters to allow half-character motion in the eight main directions. This method imposes a maximum of about 25 characters to move in any of the main directions. If the movement is only one-dimensional, as in frog-type games, many more (80 or so maximum) mobile characters can be fit in, making the technique a very useful one.

Figure 12-22. Half-Character Motion in Eight Directions



If you want to experiment without using the editor, try this with any VIC:
1. Reset the VIC (turn it off then on again).
2. POKE 44,20: POKE 20*256,0: NEW to start BASIC at $1400.
3. Enter these lines:

```
10 POKE 36869,252:POKE 36866,150:POKE 648,30
20 FOR J=8*4096 TO 8*4096+4*256-1:POKE 4096+Q,PEEK(J):Q=Q+1:NEXT
```

4. Run.

This puts character definitions from $1000 (4096) on, so POKEing 4096 to 4103, for example, alters the first eight bytes, which define @. The next bytes define A, B, C, D, and so on, and you can POKE in your own character definitions. Obviously the left, right, top, and bottom half-characters are logically related to the original characters.

Motion of Groups of Characters Together

An interesting problem is moving a collection of characters all together on the screen. These could be in a close group or spread out in a pattern like space invaders, or spread out over the whole screen to imitate stars.

Although the machine language program that follows may seem long and complex, it could have been much longer. It has somewhat minimal features in order to keep its length reasonable. One SYS call takes a collection of characters anywhere on the screen and moves them by a selectable amount. A value of 1 scrolls the characters right one position; 22 scrolls vertically down; 129 scrolls left; 151 scrolls up and right, and so on. The more often the routine is called, the faster the apparent motion.

The demonstration keeps track of nine characters. Each character's position in the screen is stored in a table, and with each SYS call the table is updated. The characters are all assumed to be identical, for example, dots (which give a starlike effect). It is easy to add a table of characters and a table of colors to the addresses.

The routine sets the background and border black, so the characters are white on black. Extra code might be added to change color RAM if this is desired.

The program won't overwrite nonspace characters already on the screen: If a character other than a space is found, it is left untouched. A collision detection flag is set. You may want to add a delay loop to slow the movement. Press any key, then type in 1, 22, 23, or 129, and watch the direction of motion change. Numbers not related to the screen width of 22 characters give a random "rain."

The screen is assumed to start at $1E00, so the program will work with an unexpanded VIC-20.

Program 12-46. Star Motion

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
Ø DATA169,8,141,15,144,16Ø,Ø,132,255,162,16,189,21
  3,3,133,253,189                             :rem 174
1 DATA212,3,133,252,177,252,2Ø1,46,2Ø8,4,169,32,14
  5,252,2Ø2,2Ø2                               :rem 58
2 DATA16,232,162,16,169,22,48,39,133,254,24,189,21
  2,3,1Ø1,254,157                             :rem 178
3 DATA212,3,189,213,3,1Ø5,Ø,2Ø1,32,2Ø8,1Ø,189,212,
  3,1Ø5,5,157,212                             :rem 14Ø
4 DATA3,169,3Ø,157,213,3,2Ø2,2Ø2,16,221,48,39,73,1
  28,133,254,56                               :rem 72
5 DATA189,212,3,229,254,157,212,3,189,213,3,233,Ø,
  2Ø1,29,2Ø8,1Ø                               :rem 66
6 DATA189,212,3,233,6,157,212,3,169,31,157,213,3,2
  Ø2,2Ø2,16,221                               :rem 59
7 DATA162,16,189,213,3,133,253,189,212,3,133,252,1
  77,252,2Ø1,32                               :rem 75
8 DATA2Ø8,9,169,46,145,252,2Ø2,2Ø2,16,232,96,169,1
  ,133,255,2Ø8,245                            :rem 24Ø
9 DATA68,3Ø,12Ø,3Ø,163,3Ø,191,3Ø,252,3Ø,17,31,77,3
  1,116,31,187,31                             :rem 155
11Ø FORJ=828 TO 997: READ X: POKE J,X: NEXT:rem 76
1ØØØ PRINT "{CLR}{2 DOWN}{RED}DEMONSTRATION"
                                              :rem 74
1Ø1Ø SYS 828                                  :rem 99
1Ø2Ø GET X$: IF X$>"" THEN INPUT X: POKE 866,X
                                              :rem 142
1Ø3Ø IF PEEK(255)=1 THEN PRINT "{HOME}COLLISION"
                                              :rem 146
1Ø4Ø IF PEEK(255)=Ø THEN PRINT "{HOME}{9 SPACES}"
                                              :rem 23Ø
1Ø5Ø GOTO 1Ø1Ø                                :rem 193
```

Source code is given for programmers who'd like to modify this program, to increase the number of objects, for example, or to allow a set of objects to be displayed in different colors.

```
;MULTIDIRECTIONAL SCROLL
;ASSUMES
;CHARACTERS CAN BE VARIED (DOT IS 46)
;COUNTER IS 2*(NO. OF ADDRESSES -1)
;COLLISION RECORDED AS 1 (0 IF NO COLLN)
SPACE          = 32
CHARACTER      = 46
POINTER        = $FC
TEMP           = $FE
COLL           = $FF
OFFSET         = 22
COUNTER        = 16
BACKGND        = $900F
;SET BLACKGROUND AND Y=0
               *=$33C
               LDA  +8
               STA  BACKGND
               LDY  +0
               STY  COLL
;ERASE EXISTING CHARACTERS
               LDX  +COUNTER
ERASE          LDA  TABLE+1,X
               STA  POINTER+1
               LDA  TABLE,X
               STA  POINTER
               LDA  (POINTER),Y
               CMP  +CHARACTER
               BNE  NOERASE
               LDA  +SPACE
               STA  (POINTER),Y
NOERASE        DEX
               DEX
               BPL  ERASE
;UPDATE ADDRESSES OF EACH CHAR
               LDX  +COUNTER
               LDA  +OFFSET
               BMI  SUBTRACT
;TEST FOR PLUS OR MINUS CHANGE
               STA  TEMP
ADD            CLC
               LDA  TABLE,X
               ADC  TEMP
               STA  TABLE,X
               LDA  TABLE+1,X
               ADC  +0
               CMP  +$20
               BNE  NOOVERFL   ;STILL WITHIN SCREEN
               LDA  TABLE,X    ;ALLOW FOR 506<>512
```

```
                ADC  +5
                STA  TABLE,X
                LDA  +$1E
NOOVERFL        STA  TABLE+1,X
                DEX
                DEX
                BPL  ADD
                BMI  PLOTCHARS
;REMOVE BIT 7 WHICH DENOTES NEGATIVE
SUBTRACT        EOR  +%10000000
                STA  TEMP
SUB             SEC
                LDA  TABLE,X
                SBC  TEMP
                STA  TABLE,X
                LDA  TABLE+1,X
                SBC  +0
                CMP  +$1D
                BNE  NOUNDERFL    ;IF STILL IN SCREEN
                LDA  TABLE,X
                SBC  +6        ;ALLOW FOR 506<>512
                STA  TABLE,X
                LDA  +$1F
NOUNDERFL       STA  TABLE+1,X
                DEX
                DEX
                BPL  SUB
;REPLACE CHARS ON SCREEN
PLOTCHARS       LDX  +COUNTER
PLOT            LDA  TABLE+1,X
                STA  POINTER+1
                LDA  TABLE,X
                STA  POINTER
                LDA  (POINTER),Y
                CMP  +SPACE
                BNE  COLLFOUND
                LDA  +CHARACTER
                STA  (POINTER),Y
CONT            DEX
                DEX
                BPL  PLOT
                RTS
COLLFOUND       LDA  +1
                STA  COLL
                BNE  CONT
;TABLE OF ADDRESSES
TABLE           .WO  $1E44,$1E78,$1EA3,$1EBF
                .WO  $1EFC,$1F11,$1F4D,$1F74
                .WO  $1FBB
```

Motion by Dynamic Redefinition of Characters

We saw how motion to the nearest half-character needed eight extra partial-character definitions. Obviously, with this method, smooth motion to the nearest dot—the smoothest VIC is capable of—in any direction is unrealistically expensive in terms of memory use. Motion sideways or up and down only, often used in shoot-out type games, is more practicable, needing about 16 extra characters for every original.

A better but very advanced method is to alter the actual character definitions in RAM to give the impression of motion, so each partial character is generated when it's needed rather than having to be stored. This means that machine language is likely to be necessary: 16 or so POKEs, just to move one character, will be too slow.

A certain amount of planning is necessary to redefine characters dynamically. First, it's desirable to keep the original, unaltered characters in memory. These can be modified, then put into the character definition area, without themselves being changed. Second, programming is made easier if the characters are numbered conveniently.

Program 12-47 shows how a character can be plotted anywhere on the screen; a square arrangement of four definable characters is necessary for this, because obviously the character will nearly always straddle adjacent cells. This type of thing in machine language can put a character anywhere on the screen, and so give very smooth motion.

Program 12-48, "Vertical Motion," is a short machine language demonstration, which redefines consecutive characters. Adding delay loops in lines 100 and 110 may make things clearer. Note the effect on numerals at the top of the screen—a realistic odometer effect. As it stands, the example uses a whole batch of characters. If you add:

```
35 FOR J=7168+41*8 TO 7679:POKEJ,0: NEXT
36 FOR J=7168+41*8 TO 7168+41*8+15: POKE J, RND(1)*256: NEXT
```

an 8 × 16 random block is controlled by the SYS calls.

Program 12-47. Plot Character Anywhere on Screen

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 POKE56,24:CLR:POKE36869,254:POKE36866,150:POKE6
   48,30                                    :rem 186
20 FOR J=6144 TO 7679:POKEJ,PEEK(32768+Q):Q=Q+1:NE
   XT: REM $1800-$1DFF                      :rem 69
30 Q=0:FORJ=6144 TO 6151: INPUT Z(Q):POKEJ,Z(Q):Q=
   Q+1:NEXT:PRINT"@"                        :rem 226
40 INPUT "X (0-175)";X                      :rem 110
50 INPUT "Y (0-183)";Y:PRINT"{CLR}"          :rem 14
60 S=7680+22*INT(Y/8)+INT(X/8):C=38400 + 22*INT(Y/
   8)+INT(X/8): REM SCREEN,COLOR POKES       :rem 69
70 XO=X-8*INT(X/8)                           :rem 45
80 YO=Y-8*INT(Y/8)                           :rem 49
90 XF=2↑XO                                  :rem 123
95 FORJ=6144+8TO6144+39:POKEJ,0:NEXT        :rem 159
```

```
100 FOR J=0TO7:POKE6144+8+J,Z(J):NEXT      :rem 179
150 POKES,1:POKES+1,3:POKES+22,2:POKES+23,4: REM C
    HR IN POSN 1 OF 1,2,3,4 SQUARE         :rem 248
160 POKEC,0:POKEC+1,0:POKEC+22,0:POKEC+23,0
                                           :rem 248
200 FOR J=0 TO 7: REM MOVE RIGHT           :rem 221
210 POKE6144+8+J,Z(J)/XF: REM DIVIDE BY 1,2,4,8,..
    .                                      :rem 206
220 POKE6144+24+J,(Z(J)-XF*INT(Z(J)/XF)) * 256/XF:
     REM PUT REMAINDER INTO NEXT DOOR CHR :rem 201
230 NEXT                                   :rem 212
300 FOR J=15 TO YO STEP-1: REM MOVE DOWN   :rem 217
310 POKE6144+8+J,PEEK(6144+8+J-YO): REM 1&2:rem 47
320 POKE6144+24+J,PEEK(6144+24+J-YO): REM 3&4
                                           :rem 144
330 NEXT                                   :rem 213
340 FOR J=YO-1 TO 0 STEP-1: REM DELETE LEFTOVER
                                           :rem 176
350 POKE 6144+8+J,0: REM 1&2               :rem 113
360 POKE 6144+24+J,0: REM 3&4              :rem 164
370 NEXT                                   :rem 217
400 GOTO 40: REM PLOT IN ANOTHER POSITION :rem 171
```

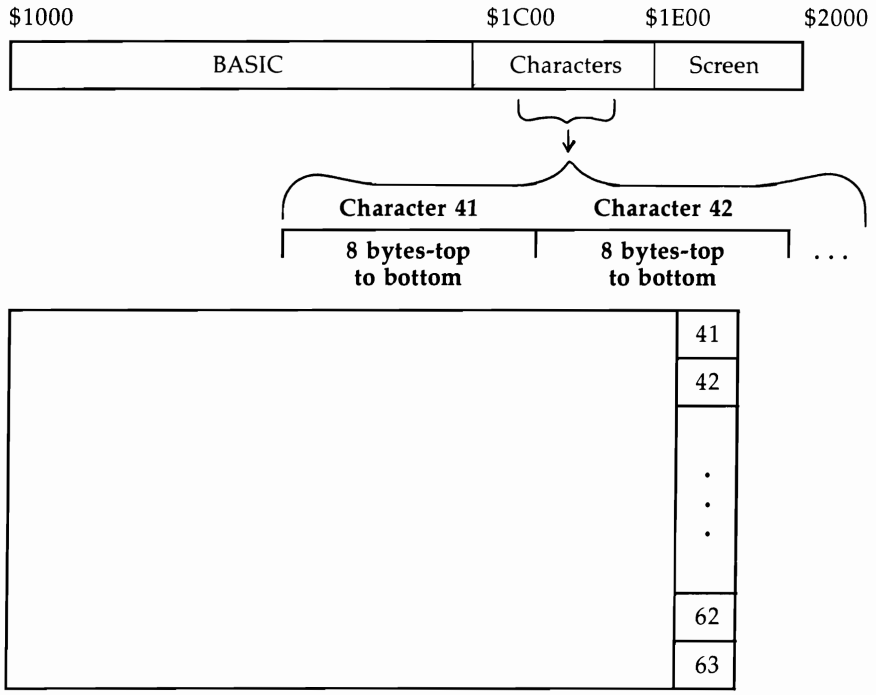
Program 12-48. Vertical Motion Using Character Definition Changes

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 POKE56,28:CLR                           :rem 172
20 POKE36869,255                           :rem 105
30 Q=0:FORJ=7168 TO 7679:POKEJ,PEEK(32768+Q):Q=Q+1
   :NEXT                                   :rem 232
40 POKE36879,8                               :rem 8
50 PRINT "{CLR}{WHT}0123456"                :rem 51
60 Q=41:FORJ=7680+21 TO 7680+21+22*23STEP22:POKEJ,
   Q:Q=Q+1:NEXT                             :rem 70
70 FOR J=828TO853:READ X:POKEJ,X:NEXT       :rem 24
100 FOR J=1 TO 80:SYS 828:NEXT             :rem 143
110 FOR J=1 TO 80:SYS 840:NEXT             :rem 138
120 GOTO 100                                :rem 93
1000 DATA 162,183,189,71,29,157,72,29,202,208
                                           :rem 176
1010 DATA 247,96,162,1,189,73,29,157,72,29 :rem 38
1020 DATA 232,224,183,208,245,96            :rem 40
```

Vertical motion is the easiest to implement. In our example, 64 character definitions are stored after $1C00, and characters 41 to 63 are POKEd into the right edge of the screen, so 41 is followed by 42 and so on. These are defined by a consecutive block of RAM locations, as shown in Figure 12-23.

Figure 12-23. Vertical Motion



Screen

All we need to do to move selected characters in this range up or down one row of dots is to move the corresponding bytes in the character definitions one byte along. So SYS 828 uses this simple loop:

```
       LDX  #183 decimal
LOOP   LDA  $1D49,X
       STA  $1D48,X
       DEX
       BNE  LOOP
       RTS
```

and SYS 840 is similar, but moves RAM the other way.

Horizontal motion is achieved by numbering adjacent horizontal screen memory locations consecutively. The programming is more complex because every eighth byte of the relevant character definitions must be rotated one bit, and this bit must be stored while 8 is added or subtracted from the current offset.

Program 12-49. Horizontal Motion Using Character Definition Changes

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
0 DATA 160,7,152,170,24,8,40,126,0,28,8,24,138,105
  ,8,170,201,184                              :rem 57
1 DATA 144,242,40,136,16,234,96,160,175,152,170,24
  ,8,40,62,0,28                               :rem 17
2 DATA 8,56,138,233,8,170,176,244,40,136,192,167,2
  08,234,96                                   :rem 109
20 FOR J=828 TO 877: READ X: POKE J,X: NEXT:rem 25
10000 POKE56,28:CLR: POKE36869,255: POKE 36879,8:P
      RINT"{CLR}"                             :rem 249
10010 Q=0: FOR J=7168 TO 7679:POKEJ, PEEK(32768+Q)
      :Q=Q+1:NEXT                             :rem 119
10020 Q=0:FOR J=7680 TO 7701: POKEJ,Q:Q=Q+1:NEXT
                                              :rem 190
10030 FOR J=1 TO 50: SYS 828:NEXT             :rem 239
10040 FOR J=1 TO 50: SYS 853: NEXT:GOTO10030
                                              :rem 85
```

Diagonal motion is more complex. The previous BASIC program illustrates the method. It is necessary to implement a form of local bitmapping, with numbering of characters as shown in Figure 12-24.

Figure 12-24. Local Bitmaps

1	3
2	4

17	14	11
18	15	12
19	16	13

The size of the bitmapped area depends on the type of motion. If the object moves out of the mapped screen area, it must be remapped. Some provision may be necessary for overlap of characters. A relatively easy solution is to exclusive-OR character definitions together where they overlap, so the full characters are recoverable.

Super Expander

The *Super Expander* is a Commodore cartridge which fits in the back of the VIC-20 or into an expansion board. It supplements BASIC with graphics and sound commands, and adds a few other BASIC commands—KEY, RJOY, RPEN, and RPOT.

KEY allows the function keys to output text; RJOY reads the joystick, RPEN the light pen, and RPOT a paddle. Music commands, and those dealing with the games port, are dealt with in Chapters 13 and 16.

So the intention of the *Super Expander*, presumably, was to provide a general-purpose aid to writing BASIC games and similar applications. The *Super Expander* uses the VIC-20's built-in features, and uses no special tricks. Anyone who says graphics are impossible without *Super Expander* is talking nonsense.

Technical Description

Hardware. The *Super Expander* is a 3K RAM pack with a ROM mounted on it. Its RAM occupies $0400–$0FFF, and its ROM occupies $A000–$AFFF, so $B000–$BFFF is left available for another utility. $A000 is the autostart area: When the VIC is turned on with the cartridge in place, the system is automatically initialized. The function keys are set up with various keywords, so pressing any of these keys will show that it is ready. The *Super Expander* can coexist with *VICMON* (at $6000), *Programmer's Aid* (at $7000), and 8K or 16K memory expansion.

There are four memory configurations possible with the *Super Expander*, depending on the absence or presence of 8K or 16K expansion, and whether or not graphics are used (see Figure 12-25).

Memory. The number of bytes free indicated when the machine is turned on is misleading. As soon as graphics are used, the entire RAM area from $1000 to $1FFF is dedicated to graphics definitions and screen storage, which instantly removes 4K of memory. *Super Expander* then limits the top of BASIC's memory to $1000. When 8K or 16K is used too, the BASIC program in RAM and all its variables is shifted up to start at $2000. This can be important: If you develop a long program you may find that it won't run because of an OUT OF MEMORY error.

The amount of memory free if graphics are to be used can be printed out like this:

```
GRAPHIC 1: F=FRE(0): GRAPHIC 0: PRINT F
```

which temporarily switches into a graphic mode. GRAPHIC 0 leaves BASIC where it is. GRAPHIC 4 alters BASIC to what it was before the GRAPHIC command. With graphics in use, the *Super Expander* alone has 3062 bytes for BASIC; with 8K, 8046 bytes; with 16K, 16238 bytes. Because BASIC must have continuous memory, and because the system puts the screen at either $1000 or $1E00, the *Super Expander*'s 3K can't be used as part of BASIC when 8K or 16K is added, although it can be used to store ML or data.

Software description. The *Super Expander* adds BASIC commands by storing them as tokens like ordinary BASIC. A wedge, to intercept BASIC, is not used. Instead, during initialization of the system, the *Super Expander* alters many of the indirect vectors around $0300, including the first six which handle tokenization, the input and output vectors, and LOAD and SAVE.

Note that *Super Expander* can be software-disconnected by POKE 783,181: SYS 64815, which bypasses the usual ROM check. It now functions as a plain 3K RAM expansion pack. SYS 64802 will reconnect it. The number of bytes free is 136 larger than reported when the cartridge is active. This is because function keys are allocated an average of 16 characters each, and eight bytes store the lengths of each string. On power-up, the keys are defined as GRAPHIC, COLOR, DRAW, SOUND, CIRCLE, POINT, PAINT, and LIST. This is a nice cosmetic touch, not necessarily of much practical use. KEY allows f1 to f8 to be reprogrammed in direct or program

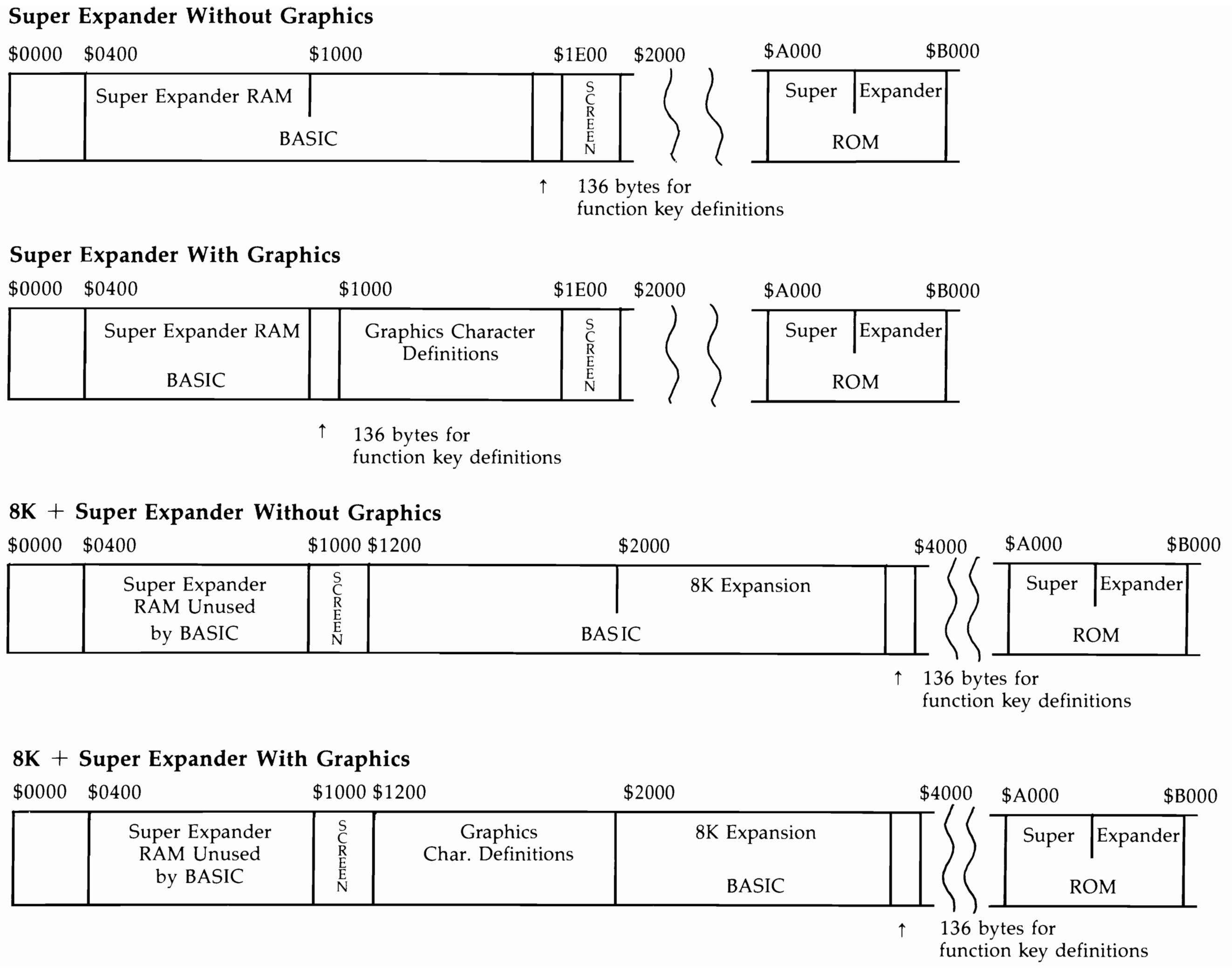

Super Expander Without Graphics
$0000 $0400 $1000 $1E00 $2000 $A000 $B000
Super Expander RAM
BASIC
SCREEN
Super Expander ROM
136 bytes for function key definitions
Super Expander With Graphics
$0000 $0400 $1000 $1E00 $2000 $A000 $B000
Super Expander RAM
BASIC
Graphics Character Definitions
SCREEN
Super Expander ROM
136 bytes for function key definitions
8K + Super Expander Without Graphics
$0000 $0400 $1000 $1200 $2000 $4000 $A000 $B000
Super Expander RAM Unused by BASIC
SCREEN
8K Expansion
BASIC
Super Expander ROM
136 bytes for function key definitions
8K + Super Expander With Graphics
$0000 $0400 $1000 $1200 $2000 $4000 $A000 $B000
Super Expander RAM Unused by BASIC
SCREEN
Graphics Char. Definitions
8K Expansion
BASIC
Super Expander ROM
136 bytes for function key definitions

Figure 12-25. Four *Super Expander* Memory Maps

mode. The total length of the strings cannot exceed 128 bytes, but there is no special limit on any one string.

The keywords introduced by the *Super Expander* are these:

204	**KEY**	Define function keys or list current definitions
205	**GRAPHIC**	Set graphic mode or return to normal
206	**SCNCLR**	Clear graphic screen
207	**CIRCLE**	Draw circles, ellipses, or arcs
208	**DRAW**	Draw straight lines between points
209	**REGION**	Change the current character color
210	**COLOR**	Select background, border, character, auxiliary colors
211	**POINT**	Plot points
212	**SOUND**	Set four voices and volume
213	**CHAR**	Print ordinary VIC text or graphics characters
214	**PAINT**	Fill an area with solid color (within limitations)
215	**RPOT**	Return position of paddle
216	**RPEN**	Return position of light pen
217	**RSND**	Return the value in a sound register
218	**RCOLR**	Return color currently in registers 0 to 3
219	**RGR**	Return graphic mode 0 to 3, but excluding 4
220	**RJOY**	Return position of joystick
221	**RDOT**	Read color register, 0 to 3, of a point

These keywords can be abbreviated on entry like ordinary keywords, typically with an initial followed by a SHIFTed second letter.

A program using these commands, saved to tape or disk, can be loaded into a computer without the *Super Expander*, but will not run, and will LIST oddly. A *Super Expander* program can be run only if the host computer is fitted with a *Super Expander*. Does this mean that a cartridge has to be present? No, a copy of the *Super Expander* in an 8K RAM pack at $A000 will run any such program, but it requires an expansion board with two RAM packs at least.

Super Expander relocated lower down in RAM, and included in the software, is feasible, but cannot be a solution in programs for resale without Commodore's permission. The best way to relocate *Super Expander* is to assume 8K or 16K expansion and use a loader to: load the relocated *Expander* into $2000 to $2FFF; put the start of BASIC at $3000; initialize the RAM version of the *Expander*, which will reserve 136 bytes at the top of memory; and load and run the BASIC program.

Because of these complications, very little commercial software depends on the *Super Expander*.

Graphics software. The *Super Expander* graphics have the serious drawback of providing only 160 × 160 pixels, equivalent to 20 rows × 20 columns of normal characters. As we've seen, the maximum bitmapped VIC-20 screen is 24 × 20, which is 20 percent larger than this utility can offer.

Other features of this utility may appear disappointing, but most are inescapable, due to limitations of the VIC-20. For example, differently colored high-resolution lines drawn at random are eventually submerged in a colored rectangular grid, but this is only a consequence of the fact that high-resolution graphics can't display more than two colors within a character.

The x and y coordinates have nominal values from 0 to 1023. The true resolution is 0 to 159 in each direction, or 0 to 79 horizontally in multicolor mode.

BASIC on small computers is generally slow with graphics, especially if individual dots are plotted. Moreover, if dots make up a picture, not calculable by a formula, storage in BASIC is likely to be inadequate. For example, DATA 123,456 takes at least seven bytes just to store a single dot, so an *Expander* with its 3K can't store more than about 350 dots out of 160 × 160, which is only 1.5 percent of the picture area. It may be best to use the *Expander* as an editor and to save the screen RAM.

Screen memory is arranged with values from 0 to 199, in columns from left to right. If you press STOP and HOME with graphics present, keys @, A, B, C, etc., are defined as the top left, top-but-one left, and so on.

BASIC Programming with *Super Expander*

The *Super Expander* could be more user-friendly. The problem is that its graphics commands rely on four color registers that store the familiar character, background, border, and auxiliary colors; these have to be referred to as 0, 1, 2, and 3. All four have to be defined with COLOR, even where, with high-resolution graphics, two aren't very relevant. And a special command (REGION) has to be used to change the character color currently being plotted. So the programmer is always a step removed from the actual colors, having to remember which register holds what color, rather than being able to simply select and use any color.

Modes. There are three graphics modes, turned on by GRAPHIC 1, GRAPHIC 2, and GRAPHIC 3. GRAPHIC 0 returns to normal text, with 22 columns, 23 rows, white background, cyan border, and blue lettering, so a program can be listed. GRAPHIC 4 is identical, except that it returns BASIC to the condition it was before going into graphics. This increases the amount of free memory, but at the risk of OUT OF MEMORY errors as explained earlier. A syntax error in a program exits with GRAPHIC 0 to print its message.

There's very little difference between the graphics modes, which are multicolor, high-resolution, and mixed. If you've understood the earlier material, you will appreciate that the only differences are in color RAM: GRAPHIC 1 has bit 3 set, GRAPHIC 2 does not have bit 3 set, and GRAPHIC 3 may have bit 3 set. The bit is always inserted or removed in the first two modes, and left alone in the third. If you switch modes in midprogram, you'll clear the screen.

Plotting. X and y coordinates are 0–1023 in all three modes—values larger than these are ignored, while negatives give ?ILLEGAL QUANTITY ERROR. The origin is top left. Converting the actual coordinates of 0–159, which are often easier to work with since they involve distinct points, requires a factor of 6.4, or 12.8 in the case of x coordinates in multicolor mode. In practice, 6.395 and 12.795 avoid rounding errors.

Colors. Color parameters are identical to those POKEd into the VIC chip:

0 Black	**6 Blue**
1 White	**7 Yellow**
2 Red	**8 Orange**
3 Cyan	**9 Light Orange**
4 Purple	**10 Light Red**
5 Green	**11 Light Cyan**

12 Light Purple	**14 Light Blue**
13 Light Green	**15 Light Yellow**

Border and character colors can only take values of 0–7; the full range is reserved for background and auxiliary colors. COLOR takes parameters in this order:

COLOR background, border, character, auxiliary.

The order is that of the bit-pairs in multicolor mode, where 00 is interpreted as background, 01 as border, 10 as character, and 11 as auxiliary. This shows how the VIC chip intrudes itself unnecessarily into *Super Expander*'s commands.

BASIC Syntax and Commands

Parameters are evaluated, so GRAPHIC X and similar constructions are valid. This sometimes confuses people. I've seen it claimed that DRAWOVER x,y,z or DRAWX x,y,z are *Super Expander* commands; in fact 'OVERx' or 'Xx' are evaluated as variables OV or Xx, which are likely to be zero, so the command is treated as DRAW 0,y,z. Table 12-5 shows how the commands GRAPHIC, COLOR, and REGION are related to the three graphics modes.

Table 12-5. *Super Expander* Modes

Register:	0	1	2	3
COLOR	Background,	Border,	REGION 0-15 Character,	Auxiliary
GRAPHIC 1 (multicolor)	0-15	0-7	0-7	0-15
GRAPHIC 2 (high-res.)	0-15	0-7	0-7	Unusable
GRAPHIC 3 (either)	0-15	0-7	8-15	0-15
	0-15	0-7	0-7	Unusable

Notes:
1. "0-7" means that 8-15 are converted to 0-7. Example: In GRAPHIC2 mode, COLOR 10,10,10,10 is indistinguishable from COLOR 10,2,2,10.
2. REGION changes the character color, and is usable in any of the modes. Use it to switch between high-resolution and multicolor modes in GRAPHIC3.

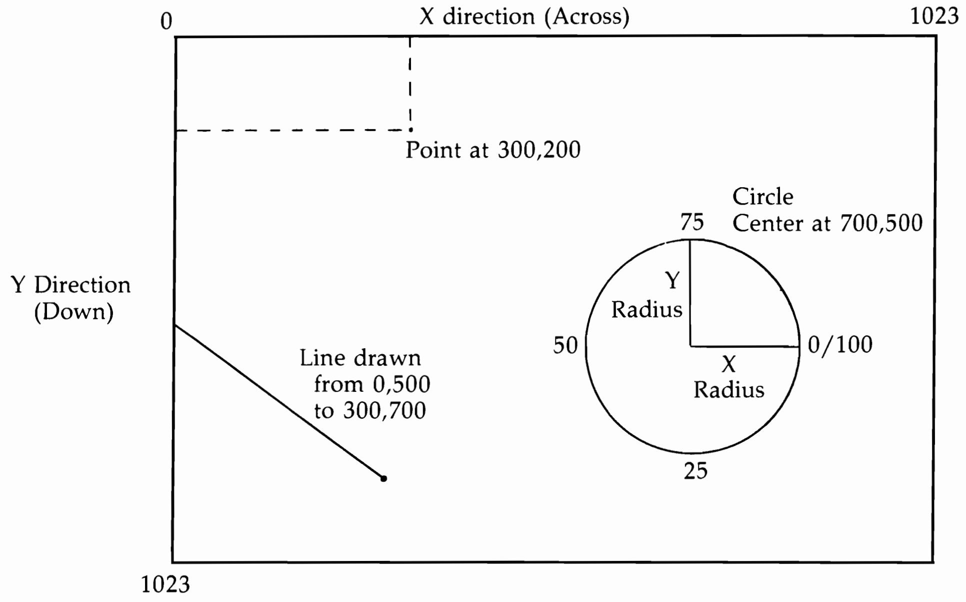
Plotting commands. POINT, DRAW, and CIRCLE plot dots. DRAW and CIRCLE construct straight lines and circles or part-circles, respectively. POINT is useful in setting starting points for lines and for drawing individual dots. The syntaxes are as follows. The portion within brackets is optional.

POINT color register 0–3, X1 coordinate 0–1023, Y1 coordinate 0–1023 [,X2,Y2,X3,Y3...]
One command can plot several points.

DRAW color register 0–3 [, X coordinate 0–1023, Y coordinate 0–1023] TO X1,Y1 [TO X2,Y2 TO X3,Y3 ...]
The line's start defaults to the previous point drawn. A set of lines can be specified, each starting at the end of the previous line.

CIRCLE color register 0–3, X coordinate 0–1023, Y coordinate 0–1023, radius across 0–1023, radius down 0–1023, [start of arc 0–100, end of arc 0–100]
This command plots ellipses, so circles can be round even in countries with different TV systems. Arc parameters are treated as 0–100, and arcs are drawn clockwise. The maximum parameter actually accepted is 255.

Figure 12-26. Examples of *Super Expander* Plotting



Other commands. SCNCLR erases graphics. Switching graphics modes has the same effect.

PAINT color register 0–3, x coordinate 0–1023, y coordinate 0–1023 fills the region surrounding x,y with solid color. In high-resolution mode, adjacent areas are likely to change color, since there just aren't enough colors to spare. There is a bug in PAINT: Its algorithm assumes that any bit which is on marks a boundary, so an area already PAINTed can't be cleared with PAINT 0,X,Y. PAINT isn't intelligent enough to be able to find any other type of boundary.

CHAR row 0–19, column 0–19, string expression puts ordinary text and characters on the screen in the current character color or in multicolor mode. CHAR allows ordinary BASIC to be mixed with the *Super Expander*'s graphics commands, and can be very much more useful than appears at first. The obvious use is to put messages on the screen, but CHAR also allows VIC's ordinary characters to be used for decorative borders, animation, and other BASIC tricks. Try a string of Commodore-+ in multicolor mode, for example.

RCOLR (register 0–3) is a function which returns the value in color register 0–3. PRINT RCOLR (2) prints the character color. RDOT (X,Y) returns the color register value, 0 to 3, of a dot. And RGR (padding) returns the graphics mode. All these commands do is simple machine language PEEKs—they are useful when drawing random graphics.

We'll look at KEY here despite its irrelevance to graphics. KEY prints out all eight current key definitions, as stored in RAM. Note that RETURN characters and quotes are both treated specially, emerging as CHR$(13) and CHR$(34).

KEY definitions use this syntax: KEY parameter 1–8, string expression. So KEY X, "HELLO"+M$ is acceptable, if X is 1 to 8. The null string is accepted, as are KEY 1, "{RED}", KEY 2,"{CLR}". If all the keys are set null, perhaps by a loop involving KEY J, a 128-character key can be defined with KEY 1,LEFT$(Y$,128) where Y$ is very long. The system isn't designed for longer strings than this.

KEY is usable within programs. 10 FOR J=1 TO 8: INPUT X$: KEY J,X$: NEXT assigns all eight keys. Fancy constructions are possible, although the average 16-byte limit is severe. KEY 1, "CHAR 0,0" + CHR$(34) + X$ + CHR$(34) + CHR$(13) assigns key 1 so that X$ prints at the top left in graphics mode.

Program 12-50. BASIC Example

```
10 GRAPHIC 2
20 COLOR 1,2,3,4
30 X= RND(1)*1000:Y=RND(1)*1000
40 DRAW 2 TO X,Y
50 GOTO30
```

This example sets high-resolution mode, white background, red border, and cyan characters, then draws lines which join random points in the screen. DRAW 2 uses the character color, so the lines are cyan. In fact, DRAW 1 or DRAW 3 also gives cyan lines: The character color is assumed to hold except for DRAW 0, which uses the background color, in effect erasing a line.

Add 35 REGION RND(1)*16 to change the character color. You'll see how the colors are all in the range 0–7, indicating that the pastel colors have been disregarded; and you'll see how the color RAM changes, making distinct lines impossible. The screen ends as a collection of colored rectangles with all the bits set.

Change line 10 to GRAPHIC 1, multicolor mode. The lines are coarser, as expected, but the effect is similar—there's no sign of more color. This is because line 40 only draws with the character color. Replace line 35 with:

```
35 C=INT(RND(1)*16):IF NOT(C=RCOLR(0) OR C=RCOLR(1) OR C=RCOLR(2) OR
   C=RCOLR (3)) THEN 35
```

and line 40 with:

```
40 DRAW C TO X,Y
```

which insures that only one of the four possible colors is selected. There is now no problem with plotting, except that the lines are coarse and the color range limited.

37 REGION RND(1)*16 increases the color range by varying the character color, without producing too blocklike an effect, since many proper lines are still plotted. You may feel the effect is enhanced if COLOR's first two parameters are identical; this reduces the palette by one color, but deletes the boundary between plottable screen area and border.

GRAPHIC 3 allows high-resolution and multicolor modes to coexist. The mode depends entirely on the character color; if it is 8 or more, you have multicolor mode. Try GRAPHIC 3 in line 10. Provided you've included line 37 with REGION, some lines emerge as hi-res, others as multicolor.

Finally, we'll see why REGION alters character colors, but leaves the others alone. Replace line 37 with

37 COLOR 1,2,11,RND(1)*15

and line 40 with

40 DRAW 3 TO X,Y

Now line 37 selects multicolor mode, and the auxiliary color varies randomly. Line 40 draws with this auxiliary color, so the screenful of lines continually changes color. The border and background can be changed like this too with

37 R=RND(1)*8: COLOR R,R,11,RND(1)*15

so the whole screen flashes.

Program 12-51. Multicolor Mode Concentric Ellipses (Four Colors)

```
10 GRAPHIC 1
20 COLOR 0,2,5,6
30 FOR R=0 TO 500 STEP 12.8
40 J= (J+1) AND 3
50 CIRCLE J,500,500,R,R:REM ,.7*R,R FOR CIRCLES
60 NEXT
```

In Program 12-51, line 40 cycles J from 0 to 3. In multicolor mode, all four colors can always be displayed. Replace line 10 with

10 GRAPHIC 2

Line 40 becomes irrelevant—delete it and change line 50:

50 CIRCLE 1,500,500,R,R

This gives a moire pattern.

Program 12-52. High-Resolution Mode Random Ellipses

```
10 GRAPHIC 2
20 X=512*RND(1):CX=X:IF RND(1)>.5 THEN CX=1023-CX
30 Y=512*RND(1):CY=Y:IF RND(1)>.5 THEN CY=1023-CY
40 RX=X*RND(1):RY=Y*RND(1)
50 CIRCLE 2,CX,CY,RX,RY
60 GOTO 20
```

In Program 12-52, lines 20 and 30 select centers and radii which fit within the screen.

55 REGION RND(1)*8: PAINT 2,CX,CY

shows PAINT and its limitations.

Program 12-53. Drawing in Three Dimensions

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 DEF FNZ(X)=SIN(X/3+1/(X+1))*10*EXP(15/(X+12))
                                              :rem 72
20 GRAPHIC 2:COLOR1,0,0,0:S=1:H=1023+6.4*S :rem 80
30 FOR X=-80TO0STEPS:M=-80:H=H-6.4*S:FORY=80 TO -8
   0 STEP -1.6                               :rem 205
40 J=FNZ(SQR(X*X+Y*Y))-Y:IF J>M THEN M=J:V=1024-6.
   3*(J+80)                                   :rem 41
50 IF V<0 GOTO 70                             :rem 86
60 POINT 2,H,V,1023-H,V:REM PLOT SYMMETRICAL PAIR
   {SPACE}OF BLACK ON WHITE DOTS              :rem 30
70 NEXT Y,X : REM ON A WHITE SCREEN WITH A BLACK B
   ORDER                                      :rem 88
```

Here S controls the step size across; the larger S is, the faster the picture is plotted, but with fewer dots. For each value of X, a series of Y values is calculated and plotted, provided each value isn't less than the minimum, M, so far. Line 50 avoids negative values; line 60 plots dots. Note that line 40 uses a square root formula to insure circular symmetry.

Chapter 6 explains how these pictures can be saved to disk and tape.

Super Expander: Machine Language Entry Points

A011	Table of 7GRAPHIC, 5COLOR, 4DRAW, etc.
A044	Initialization routine called on power-up
A047	Lower top of BASIC memory by 136 bytes
A077	RUN/STOP–RESTORE routine
A08D	KEY X, string processed here
A0BF	KEY alone processed here
A1BF	Table of all 18 *Super Expander* keywords, with high bit set
A214	Table of 18 addresses (add 1) called from $A515
A238	Initialization subroutine—puts pointers into RAM
A2A2	Table of new vectors for $0314
A372	New IRQ interrupt routine to process keyboard
A395	New input routine processed here
A454	New PRINT routine, processes new keywords, etc.
A626	Music mode—handles O, T, S, V, R, P, Q, etc.
A6C2	Music mode—plays notes A to G
A72C	**GRAPHIC** jumps to A88A
A740	**CIRCLE** " " AC93
A7A5	**POINT** jumps to AAE7
A7BD	**COLOR** jumps to AA29
A7CF	**REGION** jumps to AA6B

A7D9	**SCNCLR**	jumps to AAF2
A7DD	**SOUND**	jumps to AB35
A7EA	**CHAR**	jumps to AE57
A80A	**PAINT**	jumps to AD6C
A811	**RPOT**	jumps to AB6A
A818	**PEN**	jumps to AB77
A81C	**RSND**	jumps to AB55
A820	**RCOLOR**	jumps to AA85
A824	**RGR**	jumps to AA23
A828	**RDOT**	jumps to AA8C
A843	**RJOY**	jumps to AEDA

These are provided as a basis for exploration. Some of the parts of code—for example, RJOY to read the joystick and SCNCLR to clear the character definitions—can be used in isolation, so that SYS 43762 clears the characters, but mostly the commands are rather intertwined.

Locations 707–711 hold the first four VIC registers controlling the screen position and size; 712 ($2C8) holds the graphics mode, 0–4. (POKE 712,2: GRAPHIC 3 crashes, for example, because of this.) Locations 715–718 hold the four colors currently defined. When the cartridge is switched on, PRINT PEEKs of these give values 1, 3, 6, and 0, corresponding to white, cyan, blue, and black. So POKE 717,2: GRAPHIC 2: GRAPHIC 0 sets characters to red. *Super Expander* is lavish in its use of RAM, including the start of the cassette buffer, parts of page 2, and some zero page areas. Machine language programming is therefore liable to be tricky. A good way to try it is to use an expansion board with *VICMON* and 8K or 16K of expansion, so machine language can be tucked into the now untouched 3K from $0400 to $9FFF.

Some Final Words About Graphics

Many VIC programs use the 22 × 23 format. A few extend the number of rows—*Night Ride* and Commodore's *Jelly Monsters* are 22 × 28. Some software houses confine themselves to screens obviously designed to be 512 bytes or fewer: 16 × 32 or 25 × 20. Occasionally—for example, "Moons of Jupiter" by D. Byrden—a full screen is used. The moving objects vary in size, of course, but 2 × 2 and 3 × 2 composite graphics are common. Generally it's easier with VIC than other computers to produce largish objects, but more difficult to produce small ones, because of the quite coarse resolution.

Designing screen-graphic layouts is just a matter of starting with a grid and fitting things into it.

Provided 256 different characters are sufficient to fill the entire screen, the normal character-plotting mode used by BASIC is fine. It also allows expanded screens to be used, which is impossible in high-resolution mode; there just aren't enough bits to go around.

Note that a split-screen method, though tricky, can increase the available number of characters to 512. For example, with the screen starting at $1000, graphics might be stored in $1200–$17FF and text definitions in $1800–$1FFF. When the interrupt triggers a changeover between the top to the bottom halves of the picture, the start-of-graphics pointer would switch from $1000 to $1800. Enlarged screens become less of a problem too.

The easiest way to mix high-resolution graphics with text is to use the entire RAM area from $1000 through $2000 to define a 24 × 20 screen (using characters 16 to 256). Text and numerals can be taken from the usual ROM definitions and inserted so the whole screen is bitmapped. Mixing in narrow lettering is possible, but consumes some space (the letters' bit patterns have to be stored) and processing time.

Chapter 13

Sound

Sound

This chapter deals with music and noise generation on the VIC-20 and includes program examples in BASIC and ML. Music theory is outlined in a straightforward way where it can be applied to the VIC. It also includes a summary of *Super Expander* commands, as well as a discussion of the techniques needed to play music without slowing programs.

After working through the chapter, you will have a good appreciation of VIC's sound capabilities and limitations.

How Does VIC Generate Sound?

Chapter 5 explained that the VIC has five registers controlling sound output to the TV. This chapter shows you how to program those registers, which are listed in Table 13-1.

Table 13-1. VIC Sound Registers

Hex	Decimal		
$900A	36874	37002	Low notes
$900B	36875	37003	Medium notes
$900C	36876	37004	High notes
$900D	36877	37005	White noise
$900E	36878	37006	Volume (0–15)

All music aids, like *Super Expander*, basically provide fancy ways of POKEing values into these locations; thus, any sound effect which they can make can be made without them, too. Either decimal address is usable, although you may find the second set easier to remember. Commodore's interface chips are always fairly high in memory, so you're stuck with five-digit POKEs and PEEKs unless you use variables to cut down on space (for example, R1=36874).

Summarizing Chapter 5, values of 128 or more in any of the sound registers turn the registers on, although no sound is heard unless the volume is also on. Note that $900E shares with auxiliary color, so a simple POKE isn't always a suitable way to change volume.

The larger the number, the higher the tone. However, 255 produces the lowest tone and is not often used. The frequency is controlled by the crystal clock; with each pulse the register value is (in effect) increased, and when 255 is reached the TV output is reversed. Square wave sound results.

Chapter 5 explains how to calculate exact frequencies produced by the VIC. Unfortunately, they rarely coincide with musical notes, so VIC music is inevitably out-of-tune. The exception is octaves; tones of exactly twice or four times another's frequency can be produced.

Try POKE 37006,5: POKE 37004,230: POKE 37003,230, which puts two high notes in the upper tone registers and plays the result at a low volume. This is a richer sound than either register can produce separately, as you'll see if you cursor up and change one tone to 0.

Alter the POKEs to 140. The sound is much deeper; it also tends to be more distorted, which helps explain the odd sounds VIC may produce at low frequencies. Alter one POKE to 141, and you'll hear some beating, caused by interaction between notes of similar frequencies. This is another reason for some of VIC's inharmonious effects when notes played together fail to harmonize because their pitches don't quite match. Pressing RUN/STOP–RESTORE returns VIC to normal.

Program 13-1 rapidly switches two notes in a register; try, for example, 245 and 235. The pulsing effects vary with the numbers and their order in an unpredictable way; it's easier to experiment than to say what to expect. Hit any key to change the values.

There are many ways to modify these sounds. For instance, add lines 35 and 55 to POKE the volume with different values (a form of amplitude modulation) to increase the pulsing. The possibilities are almost endless, although the square waveform inevitably limits the VIC to low-fidelity applications. You'll never quite get orchestral quality.

Program 13-1. Switching Notes

```
10 POKE 37006,4
20 INPUT X,Y
30 POKE 37004,X
40 GET X$:IF X$ GOTO 20
50 POKE 37004,Y
60 GOTO 30
```

Program 13-2 changes register values systematically. Try inputting 1, 2, or 3 for glissandi (slides) up, and 255, 254, or 253 for glissandi down. The values 32 or 64 produce rapid alternation. Press any key to change X. You've certainly heard sounds like these before, and you can build your own sounds by adding several registers at one time or including random numbers.

Program 13-2. Changing Register Values

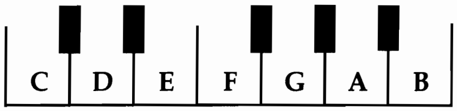
```
10 POKE 37006,4
20 INPUT X
30 POKE 37004, K OR 128
40 K=K+X AND 255
50 GET X$:IF X$ GOTO 20
60 GOTO 30
```

Before looking at music, look briefly at the white noise register at 37005. Change 37004 in line 30 to 37005 and listen to the results. The value 177 gives a frantic bubbling noise.

Music and VIC

Figure 13-1 shows a short section of the piano keyboard, starting at C. It has 12 keys in half-tone steps, tuned to frequencies in constant ratio to each other. (Two half tones together make up a whole tone, or whole step.) Since octaves differ in frequency by a ratio of exactly 2, the ratio of frequencies between adjacent keys is 2↑(1/12), or roughly 1.059463. American standard pitch sets A at 440; international standard pitch is 435. Whichever scale is used, all the other notes' frequencies can be calculated from any one note.

Figure 13-1. Part of a Keyboard



The black notes are designated "sharps" (♯) moving up or "flats" (♭) moving down. Thus, C-sharp is identical to D-flat. The white notes are "naturals."

A scale consists of seven notes taken from these 12, plus the eighth octave note. The starting note of a scale is its key. There are many possible scales; those with notes divided fairly evenly along the range are normal.

C major may be the best-known scale. It starts with C and uses only white notes. All major scales have the same basic pattern (starting tone, whole step, whole step, half step, whole step, whole step, whole step, half step). There are 12 half steps between any two notes that are an octave apart; compare this sequence with the keyboard diagram to see why C major uses white keys only.

Natural minor scales have a different pattern (starting tone, whole step, half step, whole step, whole step, half step, whole step, whole step). Again, note the total of 12 semitones. An example is A minor, which starts with A and uses white notes. There are other minor scales, plus unusual minor modes like Dorian and Phrygian, which can complicate matters even more.

To put a tune into programmable form, you must start with a list of the relevant notes. Converting notes to POKE values can be done with the help of Table 13-2; however, given the small number of values in the VIC sound registers, the result is inevitably going to sound out-of-tune. The lower POKE values offer more scope for fine-tuning but tend to sound worse overall.

It's best to avoid some of the less accurate notes. You should also be careful when selecting values that should harmonize, unless you pay careful attention to absolute pitch. For example, the major scale made of POKEs 130, 144, 156, 162, 172, 181, 189, and 193 gives the least overall errors of frequency of any major scale.

Probably the best approach, though, is to use Pythagorean harmonies, which VIC can generate exactly. You've seen how to select register values to generate octaves: 250 and 245, for example, differ by 5 and 10 respectively from 255.

Table 13-2. Note Conversions

Approx. Note	US Value	UK Value	Approx. Note	US Value	UK Value
C	135	128	G	215	213
C#	143	134	G#	217	215
D	147	141	A	219	217
D#	151	147	A#	221	219
E	159	153	B	223	221
F	163	159	C	225	223
F#	167	164	C#	227	225
G	175	170	D	228	227
G#	179	174	D#	229	228
A	183	179	E	231	230
A#	187	183	F	232	231
B	191	187	F#	233	232
C	195	191	G	235	234
C#	199	195	G#	236	235
D	201	198	A	237	236
D#	203	201	A#	238	237
E	207	204	B	239	238
F	209	207	C	240	239
F#	212	210	C#	241	240

It's not difficult to extend the principle. Table 13-3 shows the relationships between half tones and root notes; all, as it happens, are accurate approximations of true chromatic values. Under some tuning systems they can actually be treated as correct values. Some approximations, for example thirds and fifths, are very close indeed.

Notes which make up a scale (naturals) are marked with an X. A number was selected into which all or most of the scale's ratios will divide. Then this number was divided by each of the ratios, giving the columns headed 120 and 144.

The first POKE column is simply 255 minus these values. The second and third POKE columns are the values for the next two octaves. Figures marked with 1/2 could either be rounded up or down; asterisked values are not exact. Note that these figures differ occasionally from those given by the exact formula 255−N/2↑(J/12). Thus, N=120 and J=9 give 183.6, not 183. But the theory is that the integer ratio harmonies will stand together and shouldn't be mixed with chromatic harmonies.

Program 13-3, a short excerpt from a well-known Bach piece, plays three-note chords using the major scale sequence just calculated. A deeper bass and variable note-lengths are easy to put in. In addition, an economical way to control timing is to precede each set of notes with a constant and use line 50 READ T: T=TI+T and line 55 IF TI<T GOTO 55.

Table 13-3. Ratios of Half Tones and Root Tones

Ascending Note No.	Approx. Frequency Ratio	Values for VIC-20									
		Major Scale					Minor Scale				
		Notes	N=120	POKEs			Notes	N=144	POKEs		
0	1	x	120	135	195	225	x	144	—	183	219
1	18/17										
2	9/8	x	107*	148*	201½*	228*	x	128	—	191	223
3	6/5						x	120	135	195	225
4	5/4	x	96	159	207	231					
5	4/3	x	90	165	210	232½*	x	108	147	201	228
6	7/5										
7	3/2	x	80	175	215	235	x	96	159	207	231
8	8/5						x	90	165	210	232½*
9	5/3	x	72	183	219	237					
10	9/5						x	80	175	215	235
11	15/8	x	64	191	223	239					
12	2	x	60	195	225	240	x	72	183	219	237

* Not exact

Program 13-3. Jesu

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
1 REM ** 'JESU JOY OF MAN'S DESIRING' PART **
                                            :rem 171
2 REM ****** NOTE: Z CAN BE USED TO END THE TUNE,
  {SPACE}OR (AS HERE) TO LOOP               :rem 176
3 REM ****** NOTE: 'L' IS TREATED AS QUIET, SO 2,
  {SPACE}1, OR Ø VOICES CAN BE USED         :rem 139
1Ø R=37ØØ2: V=37ØØ6                         :rem 185
2Ø S(1)=225: S(2)=228: S(3)=231: S(4)=232.5: S(5)=
   235: S(6)=237: S(7)=239: S(8)=24Ø        :rem 211
3Ø B(1)=195: B(2)=2Ø1.5: B(3)=2Ø7: B(4)=21Ø
   {2 SPACES}: B(5)=215: B(6)=219: B(7)=223: B(8)=
   225                                       :rem 66
4Ø POKE V,3                                  :rem 72
5Ø GOSUB 1ØØØ                               :rem 166
55 POKE R,B(N){2 SPACES}:REM LOW REGISTER :rem 1Ø9
6Ø GOSUB 1ØØØ                               :rem 167
65 POKE R+1,S(N)                            :rem 1Ø2
7Ø GOSUB 1ØØØ                               :rem 168
75 POKE R+2,S(N):REM HIGH REGISTER           :rem 11
76 PRINT                                    :rem 25Ø
8Ø FOR J=1 TO 2ØØ: NEXT                     :rem 181
9Ø GOTO 5Ø                                    :rem 7
1ØØØ READ N$                                 :rem 79
1Ø1Ø IF N$="Z" THEN RESTORE: GOTO 1ØØØ       :rem 37
1Ø2Ø PRINT N$;                              :rem 253
1Ø3Ø N=ASC(N$) - 66                         :rem 13Ø
1Ø4Ø IF N<1 THEN N=N+7                      :rem 121
1Ø5Ø RETURN                                 :rem 166
1ØØØØ DATA C,E,C, C,E,D, C,G,E, C,E,G, D,A,F, G,B,
      F, G,B,A, D,B,G, C,E,G                :rem 184
1ØØ1Ø DATA E,G,C, D,G,B, E,G,C, C,E,G, C,G,E, F,A,
      C, F,A,D, C,G,E                       :rem 1Ø9
1ØØ2Ø DATA D,A,F, D,B,G, D,F,A, C,E,G, A,D,F, A,C,
      E, G,B,D, C,G,E, E,G,C, C,C,C           :rem 3
2ØØØØ DATA Z                                :rem 1Ø2
```

VIC as a Keyboard

It's easy to simulate a piano keyboard with the VIC, since each register has about three usable octaves. However, the registers only overlap for about one octave. Thus, it's not really feasible to simulate a keyboard in the sense of allowing three keypresses to play chords over several octaves. For example, where three notes are low-pitched, the highest register may be unable to play any of them, even when set to play its lowest note.

Program 13-4 lets you use your VIC as a keyboard. First, it defines two rows of keys to act as the keyboard, with Y being note C, then it computes an array of POKEs to correspond to the keys, turns on the volume, and plays notes depending on which key is pressed. Y, U, I, and so on are a major scale. Line 30's parameter of 60 controls the POKE values.

There's no problem in adding another double row of keys; just extend line 100 and alter line 130's parameter to 120, for example, so the lower keys play bass and the upper treble (or vice versa). Use X=PEEK(197) in place of GET to test for actual key depressions.

Program 13-4. VIC Keyboard

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
100 K$="1QW3E4R5TY7U8IO0P+@-*↑"                  :rem 54
110 DIM K%(100)                                 :rem 190
120 FOR J=1 TO LEN(K$): X=ASC(MID$(K$,J)) :rem 159
130 K%(X)=255.5 - 60/(2↑(J/12))                 :rem 185
140 NEXT                                        :rem 212
150 R=37004: V=37006                            :rem 240
200 POKE V,3                                    :rem 118
300 GET K$: IF K$="" GOTO 300                   :rem 103
310 POKE R,K%(ASC(K$))                          :rem 153
320 GOTO 300                                     :rem 97
```

Chords with the VIC

If you modify line 310, and add lines 315 and 316 as shown, you will be able to play chords:

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
310 POKE R-N,255.5-(255-K%(ASC(K$)))/2↑N   :rem 103
315 N=N+1:IF N=3 THEN N=0                    :rem 63
316 PRINT"{CLR}"PEEK(37002)PEEK(37003)PEEK(37004)
                                             :rem 87
```

Now the contents of each tone register print on top of the screen. N counts from 0 to 2, and the registers are POKEd, in turn, with the correct value appropriate to their outputs. The key Y typically prints 247, 238, and 221, so each register plays an approximately equal tone. However, tuning cannot be precise and some beating effects are unavoidable. Try pressing the space bar three times, putting 191, 128, and 0 into the registers, for a weird pulsing sound.

Chords can be constructed fairly simply. A "root chord" is the tonic (or starting) tone, plus the major third tone, plus the major fifth tone. A major third is four half steps away from the tonic, and a fifth is seven half steps away. Thus, the chord C major consists of the tones C, E, and G.

A minor third is three half steps up, so C minor is C, E-flat, and G. Other chords include the dominant and subdominant; these are built of three notes in just the same way but start at the fifth and fourth notes of the scale, respectively.

"Inversions" occur when the lowest note is moved up an octave, moving it to the top of the three-note pile.

VIC's three voices are fairly well-suited to these straightforward methods. Note that major thirds' and fifths' frequencies are in constant ratio to their tonics, of 2↑(1/3) and 2↑(7/12) respectively, so POKE 37003,X and POKE 37004,255−(255−X)/1.4983 are fifths.

Program 13-5 turns your VIC into a chord organ and lets you play a set of carefully chosen chords using the keys Q, W, E, and so on. It is easily modified.

Program 13-5. VIC Chord Organ

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 REM -- SET UP POKE VALUES FOR NOTES Ø TO 16 --
                                             :rem 51
20 DIM RV%(16): FOR J=Ø TO 16: READ F: RV%(J)=255-
   F: NEXT                                   :rem 166
30 DATA 9Ø,8Ø,72,67.5,6Ø,54,48,45,4Ø,36,34,3Ø,27,2
   4,22.5,2Ø,18                              :rem 227
40 REM ---- READ IN NOTE VALUES FOR EACH REGISTER
   {SPACE}----                               :rem 198
50 DIM MA%(2,94)                              :rem 14
55 K$="QWERTYUIOP@*↑"                         :rem 75
60 FOR Y=Ø TO 2                              :rem 232
65 FOR J=1 TO 13: K=ASC(MID$(K$,J)): READ MA%(Y,K)
                                              :rem 89
70 NEXT J,Y                                  :rem 117
80 REM ---- VOLUME ON, THEN PLAY 3-NOTE CHORD: ---
   -----                                     :rem 174
90 POKE 37ØØ6,2                              :rem 246
100 GET X$: IF X$="" THEN GOTO 1ØØ           :rem 172
110 X=ASC(X$)                                :rem 2Ø3
120 FOR Y=Ø TO 2                              :rem 21
130 POKE 37ØØ4-Y,RV%(MA%(Y,X))               :rem 114
140 NEXT Y                                    :rem 45
150 PRINT PEEK(37ØØ2) PEEK(37ØØ3) PEEK(37ØØ4) :REM
     PRINT CHORD                             :rem 151
160 GOTO 1ØØ                                  :rem 97
8999 REM ---- DATA FOR NOTES: --------       :rem 167
9ØØØ DATA Ø,1,2,3,4,5,6,7,8,9,1Ø,11,12        :rem 38
91ØØ DATA 4,6,4,7,6,7,8,11,11,7,7,8,7         :rem 15
92ØØ DATA 9,11,7,12,8,1Ø,11,16,13,11,12,13,1Ø
                                             :rem 122
```

Super Expander Sound

The *Super Expander* cartridge offers two music aids. One is a set of commands that rely on the user to put in register values; the other uses notes (so far as possible) and converts them.

SOUND

This is an easy, uncomplicated command, which simply POKEs the four sound registers with pitch and volume. For example, SOUND 0,0,128,0,8 POKEs 128 into the high tone register, at a volume of 8, and turns off the other tones. The parameters must follow the usual rules: The values 128 to 255 must be used if the sound is to be audible; 0 to 15 must be used to set volume.

CTRL-BACK ARROW

This command can be thought of as calling "music mode." In direct or program mode it converts notes written as A, B, C, D, E, F, or G (with # and $ for sharp and flat) into sound. It can play only one octave in direct mode, so it is virtually useless as a keyboard as it stands. Also, if S is accidentally pressed (it's located on the keyboard near A, B, C, and so on), it's interpreted as a register selection and turns on noise. Pressing RETURN cancels the mode.

In program mode, this command modifies PRINT, so a printed string can output music. The extra commands in the string, in addition to notes, sharps, and flats, are summarized below. As long as strings are terminated with semicolons, music mode remains in force.

P	Prints to screen during play
Q	Do not print
V	Set volume
T0–T9	Set tempo; an interrupt-driven counter is used
S1–S4	Select bass, mid, high, or noise voice
O1–O3	Select pitch within a voice
R	Rest (silence)

Machine Language and VIC-20 Sound

Any program with BASIC POKEs can be directly converted into ML. The ML versions will be significantly faster, so in the simplest examples, notes from a table can be consecutively stored in the sound registers, and a delay loop waits a suitable time between notes.

Musical notes typically have a hard-to-synthesize attack, in which the note is established, then a short delay followed by a relatively long sustain in which the characteristic harmonics of the instrument appear. Finally, during the release phase, the note fades. All this happens very quickly. Envelope shaping is possible in ML, but such ML synthesis is not really possible with the VIC.

Interrupts

Chapter 8 discussed ML which plays music as a program runs. The BASIC loader is given in Program 13-6. When run, 256 bytes are set aside in memory to store notes; their processing is controlled by POKEing memory.

Program 13-6. Interrupt Music

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 DATA 120,169,28,141,21,3,169,21,141      :rem 61
11 DATA 20,3,88,165,250,133,249,165,248    :rem 132
12 DATA 133,247,96,165,251,208,6,141,12    :rem 125
13 DATA 144,76,191,234,173,14,144,41,240   :rem 172
14 DATA 5,251,141,14,144,198,249,208,239   :rem 185
15 DATA 166,248,189,0,29,141,12,144,230    :rem 127
16 DATA 248,165,248,197,252,144,4,165,247 :rem 249
17 DATA 133,248,165,250,133,249,76,191,234 :rem 33
100 POKE 56,PEEK(56)-2: CLR                :rem 178
110 S=PEEK(55) + 256*PEEK(56)              :rem 213
120 FOR J=S TO S+71: READ X: POKE J,X: NEXT:rem 59
130 R=S+21                                   :rem 4
140 POKE S+2,R/256                         :rem 190
150 POKE S+7,R-INT(R/256)*256               :rem 70
160 R=S+256                                 :rem 65
170 POKE S+49,R/256                        :rem 252
180 POKE S+48,R-INT(R/256)*256             :rem 126
190 SYS S                                  :rem 236
1000 PRINT "{CLR}POKE START POSITION{4 SPACES}(0-2
     54) IN 247."                            :rem 2
1010 PRINT "POKE TEMPO IN 250{6 SPACES}(SMALLER =
     {SPACE}FASTER)."                        :rem 6
1020 PRINT "POKE VOLUME (0-15) IN{2 SPACES}251."
                                            :rem 12
1030 PRINT "POKE LENGTH FROM START IN 252."
                                           :rem 166
1040 PRINT "POKE NOTES IN " S+256 "TO" S+511
                                           :rem 252
2000 POKE 250,1: POKE 251,3: POKE 252,255: POKE 24
     7,0: REM DEMO OF 255 RANDOM BYTES      :rem 45
```

Enter and run the program. POKE 252,20 (to set up a 20-byte set of notes). POKE the start of the 256-byte area (7424 with an unexpanded VIC) with 200. A bleep will repeat. POKE 250 with different values to change tempo; POKEing with 1 gives the fastest rate. Something like FOR J=7424 TO 7524: POKE J,128+Q: Q=Q+1: NEXT will put ascending values in the reserved buffer, so a smoothly increasing tone will play without significantly affecting BASIC.

Locations 247 and 252 allow the 256 bytes to be partitioned, so several tunes can coexist. Obviously, a tune can be played at any tempo (notes are of fixed duration, from 1/60 to about 4 seconds), so there's considerable scope with this technique for good-quality sound programming. There's no great problem in extending the method to include all three (or four) voices and to allow for variable tempos. It is also possible to allocate more memory for storage.

Program 13-7 is a shorter example that puts ML into low memory. It can be activated at any time with a simple POKE, provided the ML isn't overwritten. It also has no significant slowing effect on BASIC.

Program 13-7. Interrupt Noise

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
2 REM ** NEEDS ONLY POKE 84Ø,X (YOU CHOOSE X) TO A
  CTIVATE **                                :rem 25
3 REM ** LARGE X VALUES CAUSE REPEAT **    :rem 132
5 REM ** POKE 849 AND/OR 85Ø WITH 234 TO REDUCE RE
  PEATS **                                  :rem 25
6 REM ** POKE 846,238 FOR INCREASING SOUND **
                                           :rem 175
7 REM ** POKE 854,13 FOR NOISE, NOT TONE **
                                           :rem 219
1Ø FOR J=828 TO 863: READ X: POKE J,X: NEXT:rem 19
2Ø SYS 828: LIST 2-7                        :rem 15
1ØØ DATA 169,3,141,21,3,169,71,141,2Ø,3,96,169,Ø,2
    4Ø,1Ø,173,72,3                          :rem 54
11Ø DATA 2Ø6,72,3,1Ø,1Ø,9,128,141,12,144,169,4,141
    ,14,144,76,191,234                     :rem 253
```

Chapter 14

Tape

Chapter 14

Tape

Tape is the most-used storage medium for the VIC-20. This chapter discusses tape operations completely; it will give you the information you need to handle any tape operation.

The chapter is arranged in six parts:

Using LOAD and SAVE with BASIC programs. Includes notes on validation, the end of tape marker, time taken, and relocation.

Handling tape files. How to store and retrieve data with tape.

Loading and Saving blocks of data. Looks at saving ML programs on the screen. Includes methods for saving at addresses above $8000 (32768).

Hardware notes on the C2N recorder. Discusses programming the motor, the cassette keys, and the VIA.

Software notes. Explains how programs and files are stored and how they can be retrieved and examined.

Security methods. Copy protection for tapes.

Loading and Saving BASIC Programs with Tape

LOADing and SAVEing programs with tape are both, in their simplest forms, very easy to use. The command LOAD prompts you with the message PRESS PLAY ON TAPE; when that is done, the next program is located and loaded. Holding down the left SHIFT key and pressing the RUN/STOP key enters LOAD and RUN into the VIC-20; it is the method that uses the minimum of keystrokes.

The command SAVE prompts with PRESS PLAY & RECORD ON TAPE. When this is done, the BASIC program currently in memory is saved on tape.

Tape, as operated by VIC, is not very fast. Table 14-1 shows approximate times needed to load or save BASIC programs. Obviously, longer programs take more time. It also indicates the number of programs which can be expected to fit onto one side of a cassette; as you might expect, longer tapes can store more programs.

Table 14-1. Time Required to Load or Save Programs to Tape

Length of Program	Approx. Time to Load or Save	Approx. Number of Programs, One Side of Cassette			
		C5	C10	C20	C30
1K	1/2 min.	4	9	19	29
4K	1-1/4 min.	1	3	7	11
8K	2-1/4 min.	—	2	4	6

Before considering the full syntax of LOAD and SAVE, it is helpful to look at a few aspects of BASIC storage. These commands have the function of loading RAM from tape and of dumping RAM to tape, respectively. In fact, they use the start- and end-of-BASIC pointers, in locations 43 and 44 (start) and 45 and 46 (end). This is why variables can't normally be stored along with BASIC. The zero byte at the very start of all BASIC programs is not used; neither is the byte at the end-of-BASIC position.

With VIC, BASIC can start in several positions, depending on memory expansion. LOAD automatically relinks BASIC so that it can work properly regardless of memory configuration, assuming BASIC doesn't use POKEs to screen and other expansion-dependent features. Append (Chapter 6) takes advantage of this mobility of BASIC on loading.

The cassette recorder (sometimes called the "Datassette") is not under full computer control, which is why screen prompts are necessary. In particular, there's only one line to test for a keypress on the cassette, so VIC cannot distinguish PLAY from RECORD. Even the fast forward and rewind keys are detected as though PLAY or RECORD were being pressed. Thus, if you want to rewind a tape and record from the start, rewind before pressing RETURN after SAVE.

In addition, be sure to press both RECORD and PLAY to save to tape. PLAY looks the same on the screen but of course doesn't work. If PLAY and RECORD are accidentally pressed for the LOAD command, the program on tape will be erased, unless the write-protect tabs at the back of the cassette are missing.

Tape operations use the interrupt, locking out the keyboard. However, the RUN/STOP key and TI clock subroutine are called at intervals, so RUN/STOP and RUN/STOP–RESTORE still work. Without this, if tape reading failed in some way, the VIC would have to be switched off. Note that the clock is updated about ten times faster than usual during tape operations.

Several programs can be stored consecutively on each side of a tape; however, the simple LOAD syntax can't distinguish between them. So the system allows BASIC programs to be named. The complete syntax for SAVE is SAVE *"filename"* which saves the program along with a name. The corresponding LOAD *"filename"* searches for the named program and also (so you know where you are) lists any other programs it may find. For example, following the command LOAD "CHECKERS" the screen may show something like this:

```
LOAD "CHECKERS"
PRESS PLAY ON TAPE
OK
SEARCHING FOR CHECKERS
FOUND CHESS
FOUND CHECKERS
LOADING CHECKERS
READY.
```

The maximum length of a name, as it appears after FOUND, is 16 characters. Provided the found program name matches, the program is loaded. LOAD "CH" loads CHESS if it finds that program on the tape before CHECKERS. LOAD "CHEC" loads CHECKERS. This is why LOAD alone always loads the first program it finds.

Full Syntax of LOAD and SAVE

The full syntax introduces two new concepts: the forced LOAD address and the end-of-tape marker. A forced LOAD means that the starting address is the same as that specified on tape; no relocatability is allowed. This is primarily important with ML programs and hardly applies to BASIC.

An end-of-tape marker signals that there are no more programs on a tape. The idea is to avoid the situation where time is wasted in reading blank tape. The marker needn't be near the physical end of the tape, and if you choose to do so you can record programs beyond it. When LOAD finds such a marker, it prints a message which should be ?END OF TAPE but is instead ?DEVICE NOT PRESENT ERROR.

Full syntax for LOAD is LOAD *String Expression, Device Number, Type-of-load Number*, where String Expression is the program name (e.g., "CHESS" or X$). Device Number is 1 or expression evaluating to 1 (tape is always device #1). Type-of-load is 0 for a relocating load and 1 for a forced load, or an expression evaluating to 0 or 1. Only bit 0 counts; a parameter of 16 is treated as 0.

As you've seen, forced LOADs are seldom used with BASIC. Also, if the middle parameter is not specified, it is assumed to be 1, so the simpler syntax of LOAD *"filename"* is usually enough.

Full syntax for SAVE is SAVE *String Expression, Device Number, Type-of-save Number*. The type-of-save parameter uses two bits; 0 means SAVE allowing relocation, while 1 means SAVE with a forced LOAD address. This too is seldom used in BASIC. If the parameter is 2, it means SAVE with an end-of-tape marker; a value of 3 means SAVE with both forced LOAD address and end-of-tape marker.

Some examples will make this clearer. SAVE "TEST PROGRAM",1,2 stores TEST PROGRAM on tape, followed by an end-of-tape marker. SAVE CHR$(18) + CHR$(28) + "PROGRAM" adds a RVS ON and a RED character to the program's name. When it's found, this will generate FOUND PROGRAM, and the name will be reversed and in red characters.

SAVE "EXCEPTIONALLY LONG NAME" stores the program in memory onto tape with the full name as it is given. Although LOAD checks only the first 16 characters, the others are in fact saved; as you'll see, they can be put to use in program protection.

Direct and Program Modes

So far discussion has focused on direct mode. However, both LOAD and SAVE work from within programs too. SAVE has the same effect as it does in direct mode. LOAD has a chaining effect; generally, the newly loaded program overwrites the older program and is automatically run.

Where a long program can be split into smaller programs (for example, in BASIC tutorials, where the earlier lessons may be no longer needed), this feature helps compensate for the VIC's small memory. It also decreases the loading delay for any particular program. Screen prompts don't appear when PLAY is pressed on the cassette; this also helps to keep the screen layout clean.

Validation and Errors with LOAD and SAVE

SAVE, although very reliable, isn't 100 percent foolproof. The tape may be faulty, for example. The best protection is to save your program twice, perhaps with names like PROG and PROGCOPY to distinguish them.

An alternative is the VERIFY command. This has syntax identical to LOAD and

SAVE, so VERIFY *"filename"* or simply VERIFY is acceptable. VERIFY works much like LOAD, except that the bytes aren't loaded into memory but are instead compared with the present memory contents. If the two are not equal, ?VERIFY ERROR results. To use VERIFY, the tape must be rewound to the start of the program being verified; note too that VERIFY takes at least as much time as saving a second copy.

If you use the VERIFY command, you will get the following screen display:

```
SAVE "GRAPHICS DEMO"
PRESS PLAY & RECORD ON TAPE
OK
SAVING GRAPHICS DEMO
READY.
```

(Rewind tape at this point.)

```
VERIFY (or VERIFY "GRAPHICS DEMO" or VERIFY "GR")
PRESS PLAY ON TAPE
OK
VERIFYING (or VERIFYING GRAPHICS DEMO or VERIFYING GR)
OK
READY.
```

VERIFY also works within a program, but if you use it in that way it is necessary to include a message telling the user to rewind.

LOAD is generally reliable, but errors are possible for reasons explained in the hardware section of this chapter. The message ?LOAD ERROR signals that the system found uncorrectable errors on tape. PRINT ST prints the value of the error status variable and gives a clue as to what happened; PRINT PEEK (159) indicates the number of errors found.

?LOAD ERROR doesn't always signify a failure to load. If a program is loaded partly into ROM, or into an area where there's no RAM, the system will find an error. These situations won't normally apply to BASIC unless a program has a forced load address into the 3K expansion area and 3K expansion is not present.

If you experiment with short test programs, deliberately recording over small sections to corrupt them, you'll be able to generate LOAD errors. Note that the resulting program is usually meaningless. Sometimes you'll generate an ?OUT OF MEMORY ERROR instead; this happens if the header (at the start of the program) is corrupted.

Programs may very occasionally seem to have disappeared from the tape. Either the program hasn't been recorded (this can be checked most easily with an ordinary audio tape recorder) or, more likely, the record/playback head needs to be cleaned or demagnetized.

Handling Files of Data on Tape

Files are more difficult to understand than programs. A file is a collection of stored data—in this case, data stored sequentially on tape. A typical use is with programs that give multiple-choice tests; once the program is in memory, a tape on a particular subject can be read and its information used. In principle, there's no limit to the number of subjects. Tapes can be changed indefinitely, so the tape files are a storage system which is independent of the program.

VIC's tape system is slower in this mode than it is with program storage. Even in the best cases it's about half as fast. To put this in perspective, Table 14-2 shows the approximate amount of data which can be stored as a file on one side of a cassette.

Table 14-2. Tape Storage Capacity

Cassette Type	C5	C10	C20	C30
Maximum Length of File	5K	10K	20K	30K
Minimum Time to Read or Write	3 min	6 min	12 min	18 min

Because of this slow speed, data file handling may be absurdly slow. It may be worthwhile saving data along with programs, although this is a tricky technique, requiring the end-of-program pointer to be moved to include variables and the first line of the program to POKE in the correct value. In addition, strings are hard to save. The whole of BASIC memory needs to be saved, plus the pointers which handle strings. Generally, then, particularly with unexpanded VICs, files offer the only method of storing reasonable amounts of data.

Files need a buffer. Unlike a program, which has a place allocated in memory at one time, files need to be written piecewise, with data accumulating until the buffer is full. It's not possible to write directly to tape, because the motor needs to pick up speed, so there's a stop-start option with files.

VIC tape files are inevitably sequential. Data has to be written (and read back) in order, so the only way to access data that is part of the way into a file is to read the whole file from the start. Moreover, there's no way to alter file information, without reading everything into memory, altering it, and writing it back—and this is usually impossible with small memory VICs. Thus the system has severe limitations, although to be fair these are largely constraints of the system and are unavoidable without advanced methods like those described later in this chapter.

There are three stages in file use. First, the file must be opened, meaning that preparations are made in memory to write data to tape. Second, you write the file to tape. Finally, close the file, meaning that the file is correctly terminated.

When the file is to be read back, perhaps by a different program, three other steps are necessary. First, open the file for reading, which prepares the VIC for input from tape. Second, read the file from tape; it may be read in parts. Finally, close the file. The final step is often not really necessary; since nothing is being written to tape, the file will be left unchanged.

File Handling Syntax

The full syntax of OPEN is OPEN *file number, device number, type of OPEN, filename.* The file number is an expression evaluating a number from 1 to 255; the device number must evaluate to 1, corresponding to tape. The filename is a string expression, usually something like "TEST DATA."

The type of OPEN is an arithmetic expression, usually 0, 1, or 2. Use 0 to open a file for reading, 1 to open a file for writing, and 2 to open a file for writing with an end-of-tape marker after the file.

The rules for default values (values assumed by the system when not specifically set) are similar to those for LOAD and SAVE. For example, when no name is given to the file, it's saved without a name; when a file is opened for read, the first file conforming to the name in the OPEN statement is taken to be the correct file.

Note that OPEN defaults to read; this prevents accidental overwriting of files. Also note that there's no type 3. Reading a file, then writing an end-of-tape marker, isn't allowed.

Because the VIC only supports one tape drive, the normal VIC setup never has more than one file open. Thus OPEN 1 is a typical command, assigning file number 1 to tape. Other file numbers are seldom used.

To see how this works, consider the following example. Type in OPEN 1,1,1, "TESTING" in direct mode and press RETURN. This opens file number 1 ("logical file 1" is another name for it) to tape, for writing. You'll be prompted PRESS RECORD & PLAY ON TAPE. When you do this, there's a delay of 12 seconds or so. A preparatory block of information, giving the filename, has been written to tape. Now type PRINT #1,"HELLO" and press RETURN. Nothing happens, although the buffer has stored the word HELLO.

But there's room for more. Type CLOSE 1 and press RETURN. The buffer is written to tape. If you don't type CLOSE, no data is written; generally, if a file isn't closed, the last batch of data will be missing. In addition, the system won't recognize that the file has ended.

Program 14-1. Using Files

```
10 OPEN 1
20 INPUT #1,X$
30 PRINT X$
40 CLOSE 1
```

To read this back, use the INPUT# command. This can only be used from within a program. Therefore, most file reading is done in program mode. Type in Program 14-1, rewind the tape, and run it. After PRESS PLAY ON TAPE there'll be a delay while the header is found and read; then the word HELLO should be printed on the screen. The word was recovered from tape, showing in miniature how files work.

Using Files Effectively

Note the 10 OPEN 1,1,0,"TEST" is necessary if you wish to name the file to be read; the default parameters have to be put in. PRINT# is usually used to write to tape. The alternative is CMD 1, which causes PRINT to output to file #1. However, it has the drawback of sometimes working in unpredictable ways. In particular, GET prevents it from working.

Either INPUT# or GET# will let you read from a tape file. GET# takes in individual characters, exactly like GET, and therefore tends to be slower than INPUT#. But it is able to treat the various special characters of INPUT (comma, colon, quotes, return) as ordinary characters.

Generally, use INPUT# when you're sure of the format of each data item. Don't try to read a string with INPUT#1,X, for example, or try to input a string longer than

88 characters. Null strings are also a problem and should be avoided.

Note, too, that number storage is inefficient. For example, ten bytes are taken up storing 1234.56. When possible, write ASCII values to tape with PRINT#1,CHR$(X), and use GET# to read them back.

CLOSE's full syntax is identical to that of OPEN, but only the file number is actually used. Therefore, CLOSE 1 is typical.

The Status Variable, ST

You can use ST to detect the end of a file. ST changes from 0 to 64 when the last record of a file is read with INPUT#. However, it isn't necessary. Alternatives are to arrange the data as it's written into a definite pattern (for instance, 100 strings alternating with 100 numerals), then read back using the same pattern, so no problems should arise. You could also write an end-of-file marker of your own (such as"****") which can be checked on readback.

ST will become 4 or 8 if a program is mistakenly read as a file. The errors mean that the program is too short, or too long, to fit the buffer.

Saving and Loading Machine Language

Programs in machine language are unlike BASIC in that they need to be positioned in a fixed place in memory. Otherwise, they generally won't work. The same applies to character definitions and screen memory; usually it's easiest to keep these at fixed locations. All these examples occupy continuous chunks of RAM, so LOAD and SAVE can be used. Files aren't necessary.

BLOCK LOAD and BLOCK SAVE, discussed in Chapter 6, provide methods (with examples) for doing this reliably. There are only two further points which need to be made here.

Forced LOAD Addresses. If memory is saved with the forced LOAD parameter, for example by SAVE "GRAPHICS",1,1, then LOAD will always position GRAPHICS back in the area it was saved from. SAVE "GRAPHICS" allows repositioning; LOAD will now load starting at the BASIC pointer area. But LOAD "GRAPHICS",1,1 forces a LOAD back into the original area. Thus, it is SAVE which determines whether LOAD always puts data back where it came from.

Therefore, when saving ML, it is usual to insure a forced LOAD by using syntax SAVE "ML",1,1.

Memory above $8000 (32768). The VIC has a limitation, carried over from CBM machines, of being unable to save data above $8000. Therefore, saving color RAM or the ROM area to tape isn't too simple. However, there are two ways it can be done, as explained here. Note that loading above $8000 isn't a problem.

Saving above $8000 with files. This relies on PEEKing the data and writing it as a tape file. The efficient way to do this is to use the routine given in Program 14-2. Any other way is likely to be much slower; this method takes about 90 seconds to save the 2K of ROM from $C000 to $C7FF.

Program 14-2. Saving above $8000

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 OPEN 1,1,1,"ROM AREA FILE"                    :rem 21
20 FOR J=12*4096 TO 12.5*4096                   :rem 150
30 P=PEEK(J)                                    :rem 176
40 PRINT#1,CHR$(P);                              :rem 78
50 NEXT                                         :rem 164
60 CLOSE 1                                       :rem 13
```

Reading back uses the same process in reverse, except that RAM of course is needed. Program 14-3 reads the file from tape and stores it in RAM beginning at $1600.

Program 14-3. Reading ROM Files from Tape

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
5 POKE56,22:CLR                                 :rem 122
10 OPEN 1                                       :rem 196
20 FOR J=5632TO7679                              :rem 32
30 GET#1,P$                                      :rem 55
40 POKE J,ASC(P$+CHR$(0))                        :rem 82
50 NEXT                                         :rem 164
```

Saving above $8000 as a program. A more sophisticated method saves this area with a forced LOAD header, so LOAD is enough to put the program back into RAM. This is faster than a file, of course. The ROM area from $C000 to $C800 is copied into RAM starting at $1600, then a forced LOAD header is written to tape, and finally the RAM is stored. All this is possible only with advanced ML methods.

Program 14-4. Saving the ROM Area as a Program

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 POKE 56,22:CLR                               :rem 166
20 FOR J=320 TO 381:READ D:POKE J,D: NEXT :rem 218
30 INPUT "NAME";N$                                :rem 5
40 N$=N$+"{18 SPACES}"                          :rem 244
50 POKE 828,3                                   :rem 149
60 POKE 829,0:POKE 830,22                        :rem 40
70 POKE 831,0:POKE 832,30                        :rem 35
80 FOR J=1 TO 16 :POKE 832+J,ASC(MID$(N$,J)):NEXT
                                                :rem 144
90 SYS 320                                      :rem 253
100 SYS 357                                      :rem 47
110 POKE 321,20                                 :rem 229
120 POKE 325,22:POKE 329,0                       :rem 79
130 POKE 333,30:POKE 337,0                       :rem 77
140 SYS 320                                      :rem 41
1000 DATA 169,105,133,171,169,3,133,194,169,60,133
     ,193,169,3,133,175                          :rem 30
```

```
1010 DATA 169,252,133,174,169,1,133,184,133,186,16
     9,0,133,183,169,255                     :rem 86
1020 DATA 133,185,76,234,248                 :rem 102
1030 DATA 162,0,189,0,192,157,0,22,232,208,247,238
     ,105,1,238                              :rem 125
1040 DATA 108,1,173,105,1,201,200,208,234,96
                                             :rem 94
```

A forced LOAD changes BASIC pointers; NEW or SYS 64802 will set them back to normal.

Cassette and Tape Hardware Notes

Virtually all VIC owners also have a Commodore C2N Datassette recorder (also marketed with the model number VIC-1530). These units have undergone several redesigns, both externally and internally. But all of them, from the early black PET/CBM models to the newer, compact rounded version, are compatible in most programming situations.

The compact C2N has a tape counter and a SAVE light (lit when recording onto tape). It also has a braided ground strap on its connector, which is not used with the VIC. Pressing RECORD also presses PLAY; earlier models have the standard security feature of requiring two separate RECORD and PLAY keys to be pressed independently.

The C2N takes its power from the VIC through the same connector that handles data transfer, although it could be powered separately with a small modification. The connector can be plugged in only one way; most recorders cannot be connected incorrectly. Cable lengths vary between models but are generally adequate.

C2Ns use an ordinary 1-7/8-inch-per-second tape transport mechanism, plus additional circuitry to control the VIC specific features like keypress detection. All tape recorders use similar principles: The C2N has an erase head, to remove signals (if any) from tape, followed by the record head, which records vertical magnetic stripes on the tape. On playback, the same head acts in reverse to play back the signals on the tape, generating induced voltage when the tape is drawn past the head.

The VIC uses a square-wave system, alternately changing the direction of magnetization. Square waves are relatively difficult to copy with ordinary audio equipment, which tends to round them off, and this provides some protection against unauthorized tape copying. However, commercial tape duplication is done by recording on tape from an original with equipment designed to preserve the original signal shapes.

The C2N's record/playback head is mounted so that its angle to the tape is variable, although it's not usually advisable to alter it. However, if the angle isn't reasonably perpendicular, read errors are possible with tapes made on other recorders. The newest C2Ns have a small hole through which the relevant screw can be turned with a plastic screwdriver.

A more common source of problems is a magnetized head. Demagnetizers are simple coils which use alternating house current to magnetize the heads alternately in opposite directions; as the demagnetizer is moved away, the inverse square law insures that the remaining magnetism is minimal. Tapes played with magnetized heads may be partly erased; if your recorder doesn't read tapes which it should be able to read, demagnetize it immediately.

The capstan is the metal spike which drives the tape at a fairly constant speed. Tape is trapped between it and a hard rubber pinchwheel when reading or writing. It's best not to leave the PLAY key pressed with the recorder off, or the tape may become dented by the capstan and give irregular playback.

When rewinding or running fast forward, the pinchwheel is disengaged and one or the other spool is driven directly. When playing at normal speed, the right-hand spool is kept under tension so the tape is wound tightly.

The tape counter is connected by a belt to the right-hand spool. One turn of the counter therefore indicates more tape when the right-hand spool is full than it does when the spool is nearly empty. Actual tape length is a quadratic expression of the counter reading, so the counter readings corresponding to programs of equal length on the same tape show progressively smaller differences.

Routine recommended maintenance involves cleaning the heads, typically with a cleaning kit consisting of cotton swabs and cleaner. The cleaner is a liquid like isopropyl alcohol, never a plastic solvent like trichloroethane.

The best type of tape is ordinary ferric oxide (not chromium) tape of reasonable quality. A screw-type cassette casing is preferable, since it can be taken apart if the tape gets tangled. Very long tapes are good for storage, but shorter tapes, perhaps with only one program each, save search time. It's not really possible to test tapes; this is far too time-consuming.

All cassettes have write-protect tabs at the back left of the cassette case, one tab for each side of the tape. If these tabs are removed, the recorder won't save to that tape, so much commercial software is packaged in cassettes like this. Put a piece of masking tape over the gap if you wish to record over a protected tape.

Since both sides of the tape are usable, only half the width (1/16 inch) is used at one time. Thin tape is prone to problems with print-through: in fact, a tightly wound spool left for some time may degrade as magnetism is transferred between adjacent turns of tape. However, even short tapes may be thin, and thus prone to this problem; there's no easy way to be sure which tapes may have trouble and which will not.

Most tapes start with a nonmagnetic leader to take the strain at the end of fast winding. The tape operating system allows for this, with seven or eight seconds of tone before actual recording proper starts, but you may prefer to manually wind forward so all recording begins on the magnetic part of tape. Another tip: Rewind brand-new tapes before recording on them. High-speed manufacturing equipment stretches tape to some extent; by rewinding the tape first, you relieve the stretch and make the tape more stable.

Non-Commodore Tape Hardware

Can you connect an ordinary tape recorder to your VIC? It is not particularly easy to do. Commodore claims several advantages for its dedicated tape system: There are no problems with recording levels, automatic or otherwise, or with tone controls and other potential incompatibilities; control over motor stop/start makes file handling possible; and the system is monaural, using a full 1/16 inch of tape rather than the 1/32 inch used on stereo recorders.

If building an interface to drive an ordinary tape recorder seems like too much trouble, you might experiment with connecting the tape write line (next to right-hand pin of the cassette port, looking from the VIC's back) to the microphone socket, and the read line (third pin from the right—next to write—looking from the back) to the earphone socket of an ordinary recorder. Remember to connect the common, or ground, connection too.

It's actually possible to interface two (or more) recorders to the VIC, with the possibility of file merges and updates.

Tape Operating Systems

Alternative ROM or RAM operating systems have been designed and are commercially available. *Rabbit* and *Arrow* are two of them; each is far faster (by about six times) than the VIC's tape system. Each has commands to LOAD, SAVE, and VERIFY, and each has a BLOCK SAVE command. Syntax is typically something like *S "PROGRAM" or *S "SCREEN",1E00,2000.

Rabbit occupies $7000–$7FFF, the same area as *Programmer's Aid*, which reduces the maximum BASIC RAM. *Arrow* occupies $A000 up, and therefore has no effect on BASIC, but makes ROM area loading and saving difficult. *Arrow* automatically initializes on power-up; *Rabbit* needs a SYS call.

The speed improvement with these systems is enormous; even 8K programs load in only about 25 seconds. This is not so far removed from disk speeds. But tapes recorded with these systems aren't compatible with ordinary programs. In spite of the attractive speed performance, little software is written for them.

Programming the Recorder

The tape port pinout is diagrammed in Figure 14-1. Pins C–F are connected to the VIA chips. Chapters 5 and 6 have examples of the programming, and the program "Tape Talker" includes a routine to read the signal from tape.

Figure 14-1. VIC Tape Port

Pin	A	B	C	D	E	F
Function	Ground	+5 Volts	Motor On/Off	Read	Write	Cassette Key On/Off

The tape motor is programmable. Its VIA location (shared with the user port and the RESTORE key) is 37148 ($911C). With the PLAY key pressed, POKE 37148,4 turns the motor off; POKE 37148,12 turns the motor on. These work in direct or program mode. Remember to leave 12 in this location, or the motor won't start.

The cassette keypress can be detected with location 37151 ($911F). Bit 6 is 1 when the key isn't pressed and 0 when the key is pressed. The PEEK values are usually 126 and 62 respectively. To test, use WAIT 37151,64 (which waits until the cassette key isn't pressed) or WAIT 37151,64,64 (which waits until it is). In ML programming, use JSR $F8AB followed by BEQ to branch if the key is pressed, or BNE to branch if the key isn't pressed.

An edge connector fitted into the tape port, with a wire from pin B (second from the left looking from the back), can provide limited power to external equipment. Small amplifiers and even printer interfaces can be powered in this way.

Advanced Tape Programming

In this section you'll see how programs and files are stored on tape and how you can manipulate them. You'll see how the headers and their programs or files are programmable independently, which means that you will be able to write tape programs which can load anywhere.

Storage at Bit Level

The VIC's tape system uses three separate square wave frequencies; the actual values vary internationally. If you call them long, medium, and short (L, M, and S), then each byte is made of patterns of L, M, and S. Bit value 0 is represented as SSMM; bit 1 as MMSS. An odd parity bit is added as an internal check (the total of 1's is made odd). LLMM marks the start of a byte. The system also has a standard tone which is used to allow for differences in tape motors.

Storage of Programs on Tape

Try recording a short program on tape and replaying it through an ordinary recorder. You will hear several seconds of a constant tone, then about four seconds of header, then two seconds of tone, then the program. The header and the program are each written twice; you'll hear a short pause midway in each. Any program records or loads in about 15 seconds plus 15 seconds per K (kilobyte).

Storage of Files on Tape

Data files are stored on tape as a sequence of fixed-length buffers. There are two reasons why files are slower than programs: One is the extra time spent writing or reading the tones; the other is the extra time spent starting the cassette motor to read each buffer. If BASIC does the reading or writing, that slows things too.

Error Correction

As data is read, errors in the first copy of the recording are noted (and corrected, if possible, by reading the second copy). Only 30 errors are allowed; these are logged in RAM at \$0100–\$013D (at the bottom of the stack area). PEEK(159) gives a count of the errors after a full read; this should be zero. Small ML routines to be put in the stack area are best started after \$013D if tape is to be used.

Headers in More Detail

Tape storage relies on headers; if you understand them, you understand most of what you need to program tape.

There are five types of headers, as diagrammed in Figure 14-2. Only the first 21 bytes are normally used, unless you wish to add ML or program protection.

Figure 14-2. Types of Headers

1	Start Address	End Address	Name	

Program Header—Relocatable

3	Start Address	End Address	Name	

Program Header—Forced LOAD Address

4	Start Address	End Address	Name	

Data File Header

2	DATA

Data Buffer

5	Start Address	End Address	Name	

End of Tape

The tape buffer, which holds headers and file data, normally extends from $033C to $03FB (828–1019), a total of 192 bytes. It can be changed by POKEing into $B2 and $B3 (178 and 179). As it happens, OPEN 1 accepts any of these header types, not just files, and provides a simple way to look at buffers.

Put a VIC program tape in the recorder, type OPEN 1, and press RETURN. Then, when the header is found, type FOR J=828 TO 850: PRINT PEEK(J);: NEXT. Now, the first byte is 1–5, the second and third bytes are the start address, the fourth and fifth are the end address, and the following bytes are the name.

Using SYS 63680 in place of OPEN 1 loads the first 192 bytes of any tape data into the buffer, so use this if you want a tape directory which allows you to examine the start of ML or BASIC programs, or read the whole of data files.

Tape Directory

Program 14-5 identifies and lists programs and files on tape and shows how the header system works. Another way to inspect header storage, with the unexpanded VIC, is to POKE 36879,8: PRINT CHR$(147) CHR$(14): POKE 178,0: POKE 179,30: OPEN 1: CLOSE 1 which puts the header into the screen. The very first character will be a, b, c, d, or e, and the name will start four characters later.

Program 14-5. Listing Tape Programs and Files

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 OPEN1:CLOSE1                              :rem 165
20 IFPEEK(828)=1THENPRINT"PROGRAM--RELOCATABLE"
                                              :rem 7
30 IFPEEK(828)=2THENPRINT"PROGRAM--FORCED LOAD"
                                             :rem 190
40 IFPEEK(828)=3THENPRINT"DATA FILE"         :rem 181
50 IFPEEK(828)=4THENPRINT"DATA BUFFER"        :rem 81
60 IFPEEK(828)=5THENPRINT"END OF TAPE"        :rem 21
70 PRINT"START ADDRESS:";                     :rem 65
80 PRINTPEEK(829)+256*PEEK(830)               :rem 17
90 PRINT"END ADDRESS:";                      :rem 140
100 PRINTPEEK(831)+256*PEEK(832)              :rem 53
110 PRINT"NAME: ";:FORJ=833TO848:PRINTCHR$(PEEK(J)
    );:NEXT                                   :rem 19
```

Consequences of This Method of Storage

Programs can be made to load into any area, even places normally impossible to load into, if the header is altered. The header itself can be used to store ML programs. However, note that saving ML starting at $033C with a monitor can't work, because the program will be overwritten by its header before it can be saved.

Since a program's start and end are defined, there's no need for an end-of-program identifier. However, files are saved as chunks, and the last chunk written (on CLOSEing the file) has a zero inserted after the data. This zero byte causes ST=64 to be set if INPUT# reads the file back. The start and end addresses with file data are simply the start and end addresses of the buffer. Because "2" identifies a buffer of data, only 191 bytes are actually storable in the buffer.

Several tricks to make programs harder to copy are explained at the end of this chapter.

ML Routines to Save, Load, and Run Tape Programs

ML programmers may want to do the equivalent of LOAD and SAVE. Conversely, programmers might want to decipher LOAD and SAVE instructions in ML programs. There are too many locations and subroutines for exhaustive listing here, but many of the most common can be outlined.

LOAD's ROM entry address is $E165. It uses the Kernal LOAD routine at $FFD5, which jumps to $F542. All the parameters are set, and several branches test for the device number of 1. Tape LOAD is at $F5D1. Normally, all programs have a header, and $F867 finds a named header. $F7AF finds any (that is, the next) header. $F8C9 reads the program itself from tape into the correct part of RAM.

A typical loader, POKEd from BASIC and designed to run an ML program, is given below. This loads whatever program it finds next on tape with a forced load, then jumps to address $1000, the start of your ML program.

```
LDA #1
TAX
TAY
JSR  $FFBA   ; file #, device, sec. addr. all 1
LDA #0
JSR  $FFBD   ; filename irrelevant
JSR  $FFD5   ; forced load to stored address
JMP  $1000
```

SAVE's ROM entry point is E153; its Kernal routine is $FFD8, which jumps to $F675. Tape saving is handled from $F6F8. $F7E7 writes the header, and $F8E6 writes the program.

ML saving might look like this:

```
LDA  #1
STA  $BA     ; device #1 (i.e., tape)
STA  $B9     ; sec. addr.=1 (i.e., forced LOAD in header)
LDX  #0
LDY  #$20    ; end address is $2000 here
LDA  #$FB    ; start address presumed in ($FB)
JSR  $F675   ; save
```

All conventional LOAD and SAVE routines use both a header and its subsequent program. Before seeing how to operate these separately, however, note the useful RAM locations in Table 14-3.

Table 14-3. Useful RAM Locations

$90	144	ST status
$93	147	Load/Verify flag (0 = load, 1 = verify)
$9F	159	Error log
$AB	171	Length of tone written to tape
$AE/AF	174/175	End address for saving
$B2/B3	178/179	Start of tape buffer
$B7	183	Length of program name
$B8	184	Current file number
$B9	185	Secondary address parameter
$BA	186	Device number (1 = tape)
$BB/BC	187/188	Start address of program name

Loading Tape Data Anywhere in RAM

Program 14-6 first loads the header, then loads the remaining program independently to start at any new address you choose.

Program 14-6. Anywhere

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 OPEN 1                                        :rem 196
20 INPUT"START ADDRESS";S                         :rem 88
30 L=PEEK(831)-PEEK(829) +256*(PEEK(832)-PEEK(830)
   )                                             :rem 216
40 E=S+L                                         :rem 176
```

```
50 POKE 830,S/256:POKE 829,S-INT(S/256)*256:rem 84
60 POKE 832,E/256:POKE 831,E-INT(E/256)*256:rem 38
70 POKE 781,3                              :rem 149
80 SYS 62972                               :rem 113
```

Programmers using ML monitors will want an ML routine to load a block into RAM. This one is reliable.

```
LDA #0
STA $93     ; specify LOAD
LDA #$16
STA $C2     ; example start address
LDA #$00
STA $C1     ; is 1600, and
LDA #$1E
STA $AF     ; example end address
LDA #$00
STA $AE     ; is 1E00.
JSR $F63A   ; or F63D or F8C0—slight differences.
BRK
```

Writing Tape Data Anywhere from RAM

This is best done with ML, since BASIC might be overwritten. Program 14-4, given earlier, saves the ROM area as a tape file and writes separate buffer and program data; use lines 110 through 1020 of that program, and add line 100 FOR J=320 TO 356: READ D: POKE J,D: NEXT for a general-purpose tape write program. SYS 320 writes to tape. Lines 110–130 specify five parameters which control the length of the tone (normally 105 before headers and 20 before the program data) as well as the start and end addresses. Like all VIC routines, the end address itself is not actually written to tape, since the program stops when it gets there. This routine uses F8EA to write and sets several parameters.

Copy Protection for Tape

Security is an interesting aspect of tape programs. Before taking it too seriously, it's worth remembering that tapes may be copyable by audio means; you should also keep in mind the fact that a number of commercial tape software houses believe that determined copiers will copy anyway and so don't put in protection. However, there are opposing views.

This section will survey some methods of complicating copying without arguing the pros and cons.

Using the Header

Because SAVE erases most of the header, a program which relies on information stored after 16 bytes of name is less easy to copy. For example, if your BASIC program is "FRENCH LESSONS", save it as "FRENCH LESSONS [2 spaces]" + CHR$(96). This puts an extra ML instruction after the name. SYS 849 from within BASIC returns, but if 96 is missing, the program crashes. However, it is relatively easy to allow for this by simply POKEing 849 with 96.

This is only a very simple example. The entire header can be filled with ML routines, which could modify BASIC, load new programs or data, or whatever.

Also at the simple level, the SYS call can be concealed in a line erased by RAM followed by deletes. It can also be disguised—for example, as SYS 84923—by inserting a zero byte after the 9 which won't list. The program's name can include screen clear or white characters, or it could even be something like ?LOAD ERROR. All that's needed is SAVE "NAME" + CHR$(13) + "ERROR" + CHR$(5), or other analogous strings.

Using the Screen Positions

ML programs are sometimes designed to load into the entire area from $1000 to $1FFF, normally into one of the two screen positions set by the VIC. An ML jump to $1000 and then to the other possible screen position at $1E00 to initialize, for instance, makes the program impossible to stop in the usual way, as some of it will be corrupted. This type of program can be developed either by moving the screen to a nonstandard position like $1200 or by writing the header separately from the program.

BASIC can use this. With the unexpanded VIC, POKE 648,28: SYS 64818 to put the screen at $1C00. Then put some ML at $1E00–$1FFF which lowers the end-of-BASIC pointer ($2D and $2E) to $1C00 and runs BASIC (perhaps with JMP ($C000)). Add a SYS call to this screen subroutine and alter the end-of-BASIC pointer by POKEing 45 and 46 to point to the end of the screen subroutine. Finally, save the results.

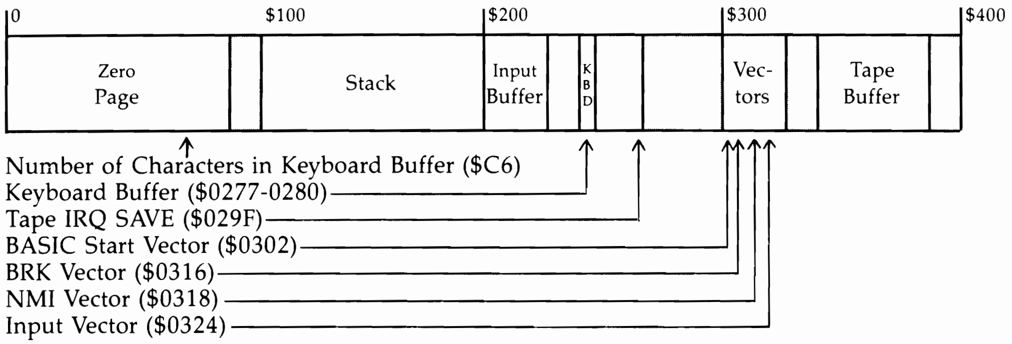
Using Headerless Programs

In BASIC or ML, you can load program data without a header. Such data can't be picked up by a normal LOAD and isn't copyable by a simple LOAD and SAVE. Of course, it must be written as a single chunk with the help of a tape write routine.

Programs Which Automatically Run When Loaded

Several techniques can be used to make programs run automatically, but all require some ML expertise on the part of the programmer. Figure 14-3 shows free RAM and key locations that are important for tape protection; three techniques can be used.

Figure 14-3. Key Locations for Tape Protection



```
Free RAM:  4 bytes $FB–$FE
          95 bytes $02A1–$02FF
           8 bytes $0334–$033B
           4 bytes $03FC–$03FF
```

RAM (with Care): Tape Buffer, 192 bytes, $033C–$03FB
Lower part of stack from $0100 or $013F if cautious about tape reading.

BASIC warm start vector in ($0302). Normally $C483, this can be directed either into the header or into the LOADed program, perhaps spanning $02A1–$0303. Then ML LOAD and RUN will automatically run the BASIC program that comes afterward.

Input vector in ($0324). Again, a loader might span $02A1–$0325, so the altered input vector might jump to $02A1.

Tape interrupt vector at ($029F). When the first program finishes loading, this vector is replaced as the IRQ. A program from $029F to $0300, for example, could start with two bytes which change the vector to point to $02A1, with ML at $02A1 which first sets the IRQ to normal.

A BASIC Autoloader

Program 14-7 will run the BASIC program immediately following it on tape. It's very simple, in order to keep it short and easily understood, and it lacks some features of more sophisticated autoload routines. For example, it doesn't disable the RUN/STOP or RESTORE keys or change the screen or character colors. Nonetheless, it works well.

When it's loaded, the first two bytes are put in the tape interrupt storage location; these alter the interrupt to point to $02A1. The rest of the program resets the IRQ to normal, then puts $83 into the keyboard buffer, which has the same effect as SHIFT–RUN/STOP. Finally, it jumps to the start of BASIC. Provided this is followed by a BASIC program, LOAD effectively becomes LOAD:RUN. The BASIC program can test for the presence of ML at $02A1 if some security is desired.

Program 14-7. BASIC Autoloader

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
20 FOR J=320 TO 356:READ D:POKE J,D: NEXT :rem 220
30 INPUT "NAME";N$                           :rem 5
40 N$=N$+"{18 SPACES}"                     :rem 244
50 POKE 828,3                              :rem 149
60 POKE 829,159:POKE 830,2                 :rem 101
70 POKE 831,200:POKE 832,2                  :rem 84
80 FOR J=1 TO 16 :POKE 832+J,ASC(MID$(N$,J)):NEXT
                                           :rem 144
90 SYS 320                                 :rem 253
110 POKE 321,20                            :rem 229
135 FORJ=828TO851:READD:POKEJ,D:NEXT        :rem 32
140 SYS 320                                 :rem 41
1000 DATA 169,105,133,171,169,3,133,194,169,60,133
     ,193,169,3,133,175                     :rem 30
1010 DATA 169,252,133,174,169,1,133,184,133,186,16
     9,0,133,183,169,255                    :rem 86
1020 DATA 133,185,76,234,248               :rem 102
1050 DATA161,2,169,191,141,20,3,169,234,141,21,3
                                            :rem 89
1060 DATA169,131,141,119,2,169,1,133,198,108,0,192
                                           :rem 204
```

Many tapes using this sort of copy protection load into the entire area of RAM from, for example, $0300 up. This makes LOADing times rather long, as the area from $0400 to $0FFF is usually wasted.

The general method should be clear: A forced LOAD header loads ML into parts of RAM which are used by the system. When the initial LOAD is over, the changes take effect, driving ML routines to load and run more material.

There is considerable scope for ingenuity in using encoding routines, nonstandard 6502 instructions, programs without headers, and overlays. If the RUN/STOP and RESTORE keys are disabled, a reset switch leaves most memory intact but erases all of RAM from $0000 to $0400 except the stack ($0100–$01FF). Thus, if key parts of a program are left in this area, the result can be all but impenetrable.

Chapter 15

Using the Commodore Disk Drive

Using the Commodore Disk Drive

Disk storage is more expensive than tape, but it is also more versatile. It can be used to store a selection of programs for rapid loading, but it also gives you access to large amounts of data.

This chapter begins with a discussion of straightforward disk commands and progresses through more advanced material. By the end of the chapter you'll be able to handle most disk programming tasks.

The chapter is divided into eight parts:

Introduction to disk storage. How it differs from using tape.

Basic disk commands. LOAD and SAVE, formatting a disk, reading its directory, channel 15, and more.

Handling disk files. Storing data in sequential and relative files, and tricks to use with program files.

Disk commands. Includes notes on each command, as well as a checklist of potential programming difficulties. Also covers messages from the disk drive.

Utilities. An annotated index of Commodore's *TEST/DEMO* diskette.

Hardware notes. Discusses disk drives and diskettes.

Disk storage of data. Explains how data is organized on diskette and how to read and/or change it using only BASIC. Special versions of the directory are described. The direct access commands are listed and explained. This section is essential for a full grasp of disk use.

Machine language programming with disks. Deals with converting BASIC into ML for extra speed.

Introduction to Disk Storage

The VIC-20's serial port (next to the video port) is a design unique to Commodore. It accommodates single-disk 1540 and 1541 disk drives. The earlier 1540 models were designed specifically for the VIC-20; the main difference between it and the 1541 is a single ROM chip in the 1541 that makes it compatible with both the VIC and with later Commodore machines. VIC-20 users are often told to type OPEN 15,8,15,"UI-":CLOSE 15 and RETURN, when using the 1541, but many users do not find it necessary.

VIC's disk units store data on 5¼-inch diskettes, and a demonstration diskette should be packed with each disk drive. The usual advice is to switch on the disk drive first, then the VIC, and then any printer, but this usually doesn't matter.

A diskette is inserted label up, with the read/write slot nearest the disk drive. The drive door, when closed, clamps the disk firmly and permits reading and writing to take place.

Disks are faster than tape, but the VIC's system (with data transferred one bit at a time) isn't fast by today's standards. Generally, you should allow about 10 seconds per 4000 bytes plus about 5 seconds overhead—roughly 25 seconds for an 8K program.

The non-CBM tape operating systems described in the previous chapter are just as fast; however, disks allow for random access. The disk lets you choose from a whole range of programs or files on a single disk, giving a versatility unavailable with VIC tape systems.

So-called "black boxes" are available to allow several VICs to access the same disk drive. This saves money where a group of people must use the same programs (in some teaching situations, for example) and the serial connector is reliable over distances up to about 12 meters (about 40 feet).

The VIC's disk drive is autonomous. In other words, it is largely independent of the VIC. In fact, it has as much ROM as the VIC itself. When the drive receives a command from the VIC, that command is stored in the disk drive's RAM and carried out only when the disk drive decides to do so. Similarly, results are typically stored in a buffer, waiting for the VIC to read them. This explains how it is possible for the VIC to print READY, even when the disk drive is still obviously working. It also permits disk functions to be changed by switching ROMs within the drive.

One side effect of this arrangement is that errors can occur either in the VIC or in the disk drive. For example, if there's no diskette present, the disk drive can't read data and an error condition is present in the disk drive. Commodore has a special channel to allow transfer of information from and to the disk, as you'll see.

The keyboard is affected by disk operations. For example, while data is being read from disk, the interrupt is mostly off. This means the keyboard cannot be processed normally, and keys pressed when the drive is active may not show up when it stops. If you're using a disk system to store data, bear this in mind. Clear the keyboard queue (POKE 198,0) before INPUT or GET to insure that incomplete data isn't written to disk.

Disk commands are more explicit than tape commands. Unless otherwise instructed, the VIC assumes that LOAD (or whatever) applies to tape. Thus, disk commands always include the device number, which is normally 8. Also, because disks can store many programs, the bare command LOAD is disallowed. Instead, quote marks and names are used. For example, to simply read the disk directory you must type in LOAD "$",8 then LIST. The VIC-20 Wedge (on the demo disk) offers limited help with this.

Serious disk drive users, who are using disks to store valuable data and writing their own programs too, should take note of a few points to keep from losing data. First, duplicating disks for security purposes isn't easy with only one disk drive. Second, there are potential problems when the process of writing to a disk is interrupted (for example, by a syntax error) because incomplete information is left on the disk and may corrupt other files. Subsequent parts of this chapter will discuss these areas in more detail.

Basic Disk Commands

This section will take you through the steps needed to store a program on a new, blank disk. You will then see how channel 15 allows communication between your VIC and the disk drive.

Formatting a Disk

Switch on the disk drive and VIC, insert a new disk in the drive, and close the drive door. Then you're ready to format the diskette. Formatting gives the diskette a name and a two-character identifier; it also writes data on the disk to identify it as a VIC disk.

Every time you format a disk, all programs and any data that it contains will be wiped out. Don't format a disk more than once unless you no longer need its contents and prefer an empty disk.

This is the formatting command:

```
OPEN 15,8,15,"NEW:NAME,ID":CLOSE 15
```

To format a disk, type in this command and press RETURN. The red light on the disk remains on for about a minute and a half; the drive should first move to the outer track (with some noise), then click gently as it writes to the disk. After 35 clicks the drive will stop and the red light will go off.

The disk's name can have up to 16 characters (for example, DISK TESTS 1) and the identifier up to 2 characters (for example, 00 through 99). Avoid using the symbols ? # * , : " or @ in names sent to the disk, since they may be interpreted as separators or special operators.

The identifier is written to the disk nearly 700 times. It helps to check that data is in its expected position and that the disk hasn't been inadvertently changed. It's thus advisable to give your disks individual IDs; otherwise, swapping disks to load from one and save to the other may scramble data if the IDs happen to match.

Inspecting a Disk's Directory

Any disk's directory or catalog is recoverable with the following command:

```
LOAD "$",8
```

Type in this command, press RETURN, and then LIST the directory.

A newly formatted diskette's directory has its name and ID in reverse video, followed by 2A, which shows the type of Commodore disk format. The message 664 BLOCKS FREE shows that 664 blocks of 256 bytes each are available for storage (but not quite all are usable). The directory is held as BASIC, as you may have inferred from LIST, and that explains the leading zero at the start of the directory. It is a dummy line number and can be ignored.

Note that inspecting the disk with LOAD"$",8 will erase any program you have in memory. Conversely, without NEW, a program typed in after reading the directory may contain odd lines left over from the directory. Subsequent sections give the full syntax of LOAD "$", allowing parts of the directory to be listed, and present BASIC programs that sort the directory and do other tricks.

Saving a Program

To see how to save a program to disk, first type NEW, press RETURN, and then type in any short program. Then type in SAVE "PROGRAM",8 and press RETURN to save the program to disk with the name (up to 16 alphanumeric characters) you gave it. Don't include ? # * , : or @ in the name. A null name (SAVE "",8) is rejected with ?MISSING FILE NAME ERROR.

If you wish, you can VERIFY, with either VERIFY "PROGRAM",8 or VERIFY "*",8. In either case you should see the following display as the program is compared with the version in memory.

```
SEARCHING FOR PROGRAM
VERIFYING
OK
```

Disks are generally reliable enough to make this unnecessary. The version with * uses Commodore's pattern-matching technique, explained under "Disk Commands." The same idea allows LOAD "*",8 to load the first program it finds, when the drive is turned on.

When using disks, SAVE won't work if a program with the same name already exists on a given disk. This is a security measure. An error is generated by the disk drive, and the red light flashes, but no error message is displayed on the screen. You'll soon see how to read the disk drive's message.

SAVE's syntax has an optional form causing SAVE with replace. It allows a program to overwrite another program with the same name. The command is SAVE "@:PROGRAM",8 where the added @: is interpreted by the disk as a command to overwrite. So, if you modify your program and then enter SAVE "@:PROGRAM",8, you'll find the newer version present on LOADing later. It's only fair to note that disk errors of the kind caused by unclosed files (corrupted programs and/or data) have been associated with this command, so if you're the cautious sort it's better to scratch the old file before saving.

After saving a program, LOAD and LIST the disk directory. The diskette's name and ID remain the same, but a program (PRG on the line after the name shows it's a program) is present. If it's a short program it will probably occupy only one block, leaving 663 blocks free.

Loading a Program

To load a program, type in LOAD "PROGRAM",8 and press RETURN. The disk drive will run for a few seconds, and the READY prompt will appear. Then LIST or RUN your program. LOAD "PR*",8 or LOAD "*",8 will have the same effect.

LOAD "*filename*",8,1 is necessary for a nonrelocatable load. Machine language, graphics definitions, VIC chip registers, and any data which needs to be replaced at the point from which it was saved uses this syntax.

There's a slight complication with LOAD and VERIFY in BASIC. VERIFY "PROGRAM",8 after LOAD "PROGRAM",8 reads and compares programs byte-for-byte; however, if BASIC was loaded into a different start area because its memory configuration has changed, the link pointers will be different and you'll get a spurious ?VERIFY ERROR message. Try VIC-20 WEDGE with an unexpanded VIC. VERIFY can be used from within a program; with BASIC this could be used to check that the right memory is being used.

You can also use LOAD in program mode. LOAD from within a program produces the same chaining effect that you get with tape; however, it is much faster. The new program, presumed to be BASIC, runs from the start. It retains all the old variables if the new program is no longer than the old one and if strings and functions held within BASIC are redefined. See CHAIN and OLD in Chapter 6 for further discussion.

ML and memory dumps can also be loaded successfully. Use 10 X=X+1:IF X=1 THEN LOAD "GRAPHICS",8,1:REM ONLY LOADS FIRST TIME.

Scratching a Program

"Scratch" is a strange computerese word meaning to remove or erase a program. With tape it's simple to rewind and obliterate a program by recording over it. Disks need a specific command, however, because the disk drive can't know which program to scratch unless it's told.

SCRATCH has this syntax:

OPEN 15,8,15,"SCRATCH:*filename*":CLOSE 15

Pattern-matching abbreviations are also usable, so OPEN 15,8,15,"SCRATCH:N*" :CLOSE 15 scratches anything beginning with N, while OPEN 15,8,15,"SCRATCH:*" :CLOSE 15 scratches everything and leaves the diskette empty.

The number of files scratched is reported in channel 15.

Copying Programs from One Disk to Another

Both BASIC and machine language programs can be transferred from disk to disk. However, BASIC programs are easier to transfer because the system keeps track of where they start and end.

First, though, it is helpful to look at the disk operation called *initialization*. With CBM disks this means forcing the drive to read the current diskette's directory information into its own memory. This process is often automatic (for example, when a directory is loaded from disk), but to be on the safe side you can use this command to guarantee that the disk to be copied to is correctly set up. INITIALIZE has this syntax:

OPEN 15,8,15,"INITIALIZE":CLOSE 15

or

OPEN 15,8,15,"I":CLOSE 15

or

PRINT#15,"I" (after OPEN 15,8,15 has been performed)

To actually copy a BASIC program, first acquire two disks. Call them "source" and "destination." Then follow these steps:

1. LOAD "*filename*",8 from the source diskette, being sure that your VIC has enough RAM to hold it.
2. Remove the source disk, replace it with the destination disk, and close the drive door.
3. Enter OPEN 15,8,15,"I" to initialize the destination disk.
4. SAVE "*filename*",8 to save the program onto the destination diskette.
5. Replace the source diskette, enter PRINT#15,"I" to initialize it, and return to step 1. Repeat the process until you've moved as many programs as you want.

Copying Machine Language and Memory Dumps

To copy machine language or memory dumps, you'll need to know the start and end address. Finding the end address is simple: Enter LOAD "*filename*",8,1 then PRINT

PEEK (45),PEEK (46). The end pointers are thus set, but the beginning is lost. To locate the starting address, you can read the start address as a program file.

Once you've located those addresses, the process is similar to that for BASIC. Have source and destination diskettes ready. Then follow these steps:

1. LOAD "*filename*",8,1 from the source diskette.
2. Exchange diskettes. Type NEW.
3. Enter OPEN 15,8,15,"I" to initialize the destination diskette.
4. POKE 43 and 44 with the low byte and high byte of the starting address, and POKE 45 and 46 with the low byte and high byte of the end address.
5. SAVE "*filename*",8
6. Exchange diskettes, enter PRINT#15,"I", and repeat from step 1.

Communicating with the Disk Drive: Using Channel 15

Channel 15 is variously known as the error channel, the command channel, or the information channel. The number 15 refers to its secondary address, the third parameter of the OPEN statement. Disk drives use this third parameter to identify the channel number, and generally it makes sense to use the same number for the file where possible. To better understand this, enter and run Program 15-1, a one-line BASIC program:

Program 15-1. Using Channel 15

```
Ø OPEN 15,8,15:INPUT#15,E,E$,T,S:PRINT E;E$;T;S:CL
  OSE15:END
```

If the disk drive has no current error stored in it, the result will be 0 OK 0 0 where the first zero is the error number, OK is the message from disk, and the track and sector of the error (both zero) mean there's no problem. This is a long-winded way to discover the disk status. It can be tedious to enter and run it just to discover the reason for a disk error or problem. Note that direct mode can't be used; the line must be entered as part of a program. Therefore, when developing disk programs, it makes sense to include this as, say, line 40000 so that RUN 40000 is ready and waiting if needed.

Note that reading the channel clears it, so a subsequent read will say OK even if there's a major problem (like an open disk drive door). The message remains until either the channel is read or disk activity forces in another message.

You'll see later the circumstances in which the flashing "error" light, which is apt to alarm newcomers, can be ignored. First, though, deliberately generate some errors and watch the effect of running line 10 above:

1. Enter LOAD "%",8. This program doesn't exist on disk. RUN gives:
   62 FILE NOT FOUND   0   0
2. Enter LOAD "1:HELLO",8. It tries to load a program from a nonexistent drive. Your drive is drive 0; since it's a single drive, there's no drive 1. The message is:
   74 DRIVE NOT READY   0   0

3. Enter SAVE "PROGRAM",8. (I'm assuming PROGRAM is still present on the disk.) RUN gives this:
   63 FILE EXISTS 0 0
4. Enter OPEN 15,8,15,"S:PROGRAM":CLOSE 15. This scratches PROGRAM from the disk. (The initial is sufficient.) Now RUN gives:
   1FILES SCRATCHED 1 0

   which, translated, means that message 1 (which always deals with scratched files) reports that just one file was scratched by the command. More than one file may be scratched if pattern matching (with "S:*") is used.
5. Turn the disk drive off. Open the disk door if there's a disk present; this insures that no magnetic glitch can occur on the disk. Turn on the disk drive and run immediately. Your message is something like this:
   73 CBM DOS V2.6 1541 0 0

   which tells you what type of ROM your disk unit has.

If you want to experiment more, try VIC-20 WEDGE from the demo diskette, which modifies BASIC so that just pressing @ prints the message.

Sending Messages to the Disk Drive

You've seen how to read channel 15, but how are messages sent to the disk? The syntax has two forms, both based on the syntax of OPEN:

OPEN 15,8,15,"command"

or

PRINT#15,"command" (assuming OPEN 15,8,15 has been carried out)

Formatting a disk and scratching a file are two examples we've seen so far. A subsequent section includes a comprehensive list of eight disk commands that use this channel.

Handling Disk Files with BASIC

Files on disk are more complicated, and thus more difficult to understand, than tape files. If you're a newcomer to disks, you may find the concept of a file hard to grasp. However, after working through the examples which follow, it should become clear.

There are two essential aspects of any computer filing system. One is that an external storage device (like a disk drive) must be able to store and retrieve data in a reliable way; the other is that the computer must have commands available to handle the output and input of that data. To illustrate the second condition, consider the fact that the VIC's disk drives can be programmed to store data almost anywhere on the disk surface. Although this can be a very useful feature, it does not provide a file in the true sense because specially written commands have to be used to process the data.

Disk files, unlike tape files, aren't always exclusively read or write files. The versatility of disks enables files to be open for writing and reading at the same time.

Another example of disk versatility is that several disk files can be open at once. For example, a sequential file—identical to a tape file—can be read, updated, and then written to a second sequential file. This is not possible with the VIC's tape unit. The tape system can use only a single track of tape, whereas a diskette works more like a multitrack system.

Types of File Organization

The VIC's disk system supports four types of files, shown on the directory as PRG, SEQ, REL, and USR (program files, sequential files, relative files, and user files). A user file allows a programmer to build his own type of file by writing data directly to the disk and the directory, but all the work of arranging the data on disk and reading it back must be done by the programmer. The subsequent section on disk storage explains how this is done; meanwhile, USR can be ignored, since it is not a true file system.

Program Files

These are simply programs or memory dumps which can be loaded and run (if they're programs) in the usual way. However, the disk system also allows them to be read from and written to, and that makes several nice programming techniques possible.

Sequential Files

Next to program files, sequential files are the easiest to understand. Data is written to them from a buffer, in sequence, without restriction on the type of data or its length. Thus, the file can be of any length, regardless of the computer's RAM. Because sequential file data isn't ordered, it is usually read back in sequence starting at the beginning. As a result, long sequential files can be slow to handle.

In practice, VIC sequential files—whether on tape or disk—aren't usually quite so free from structure. Why? Because it's easiest to use PRINT# to write data to a file, and INPUT# to read it back, and both of those commands have certain restrictions on length and type of character that they can handle.

Relative Files

Relative files do not have to be read from the beginning. Any record in the file can be read by number; thus, "random access" is a name sometimes given to such files. With the VIC, this is made possible by defining a record length when the file is initially opened, and diskette space is assigned as it's needed. For example, if record number 200 is to be written to a new relative file, the disk's operating system allocates space on the diskette for 200 records of the desired length before writing the data of record number 200. Relative filing is more ordered than sequential filing; later you'll see exactly how that is accomplished. For the moment, note that the records are the same length. This wastes disk space if some records are far longer than others. Obviously, a shorter maximum record length allows more records to be filed.

Note also that accessing records by number may not be what you really want. For instance, you may find yourself using extra files, or arrays, to convert JONES into number 93. Nevertheless, this is the most advanced form of filing offered by most microcomputers.

Direct access files may also be used. Commodore's manuals refer to the system of storing data at certain sectors on the diskette as "random access," which is explained in the section on data storage. More usually, "direct access" filing refers to a system allowing access to records by a single key. This is a fascinating system of file organization, easily implemented on the VIC.

To take an actual example: You want to be able to read from disk, as fast as possible, information on any one person out of a group of 400 by entering the person's name. VIC's relative file system requires a number between 1 and 400, and the idea of direct access is to convert the name into a number within that range. This could be done by converting some of the name's characters into ASCII, then generating a key from 0 to 1 and using RND(−key)*400+1 to generate a repeatable value in the required range. A good algorithm will, of course, evenly spread the coded values of the keys. With this kind of organization, records are held in the file in a jumbled sequence but can be recovered by applying the coding algorithm to the key.

Direct access has several drawbacks, however. First, there's no easy way to print a sequential list of the records. In other words, it's difficult to check what's on file. Second, many keys will inevitably generate the same record number, so it's necessary when writing to the file to check that the record number isn't used (if it is, try the next one). It's also necessary, when reading, to read until the correct record is found. For that reason, the file has to be longer than the number of records by at least 30 percent. In this case, about 35 percent of the records are synonyms, but this drops to 25 percent if all keys are tested first and all synonyms are stored in the file in a second pass. If the most frequently used records are entered first, efficiency improves again.

Inverted files are used in data base programming, where there are huge amounts of data in a main file, and where you want a list of items conforming to several stringent criteria.

Instead of reading the entire file, a large number of smaller files are established, with each holding keys to a subset of the original data. As a result, fewer files have to be read, but only at the expense of extra file space being taken up and extra work being required to add new records to a number of files. For example, 26 subsidiary files for initials A through Z plus a full-length relative file works well in some applications.

Writing and Reading Disk Files

OPEN. You can OPEN a disk file using this syntax:

OPEN file number, device number, disk channel, command string

For example, OPEN 2,8,2,"0:ORDINARY FILE,S,W" opens file #2 to disk drive #8, and uses channel #2 in the disk drive. (This is relevant with random access storage and with relative files.) The command string begins with 0: which is a construction from Commodore drives which have two disk drives, instead of only one. The other drive used the prefix 1:, but that generates 74 DRIVE NOT READY with VIC disk drives. This chapter ignores 0:, but readers with access to PET/CBM machines should bear it in mind.

The other part of the command string uses commas as separators and causes a sequential file, called ORDINARY FILE, to be set up. PRINT#2 will now write to this file, and CLOSE 2 safely completes all the housekeeping. You'll see further examples shortly in the demonstration programs on file handling.

Note that the file number cannot be zero. Ordinarily, use any number from 1 through 127. File numbers 128 through 255 should usually be avoided, because PRINT# to them sends linefeed (CHR$(10)) with carriage return. This is useful with some printers, but not generally helpful with disk files.

The device number is 8 unless changed by hardware or software. It's possible to connect two drives at once, one with device number 8 and the other with device number 9, and open several files to each (OPEN 3,9,3,"ORDINARY FILE,S,R"), allowing reading from 9 and writing back to 8.

The channel number should generally avoid 0, 1, and 15: 0 and 1 are related to the directory, and 15 is the command channel.

The command string syntax varies with the type of file. See the demonstration programs.

PRINT# is one of three BASIC commands (the others are INPUT# and GET#) that let you send output to a file and read it back, either as a batch of characters (INPUT#) or as individual characters (GET#). PRINT# outputs string and number expressions to the file in just the same way that output is sent to the screen. The organization of relative files is identical to that of sequential files, as far as the stored data is concerned. PRINT# treats a colon or end-of-BASIC line as requiring a carriage return character. The semicolon causes PRINT# to print no extra characters. The comma outputs 11 spaces (in effect tabulating half a screen across). Numbers appear in the file with a leading minus or space, and with a trailing space, too.

The effect of OPEN 2,8,2,"TEST,S,W" followed by PRINT#2,"HELLO";12345; "HELLO","HI" is shown in Figure 15-1.

Figure 15-1. Using PRINT#

H	E	L	L	O		1	2	3	4	5		H	E	L	L	O												H	I	Rtn	

PRINT# can output individual characters for GET# to read back later. In this case, there are no restrictions on character types. PRINT#2,X$; outputs the character stored in X$ (the semicolon prevents unwanted RETURNs), and GET#2,X$ reads the character back. PRINT# should thus be fairly straightforward, although most programmers will find themselves rearranging colons and semicolons to remove small bugs in their programs.

If you're reading data with INPUT#, remember not to write strings longer than 88 characters. INPUT# generates ?STRING TOO LONG if this happens. If long strings appear to be unavoidable, it's always possible (though slower) to evade this problem by replacing INPUT# with something like X$="": FOR J=1 TO 100: GET#1,Y$: X$=X$+Y$: NEXT.

Remember to include RETURN when estimating the lengths of records. Relative files in particular need a RETURN character if data is read back by INPUT#, and this adds 1 to the maximum record length.

INPUT#. Using INPUT# is the most convenient way to fetch information from files. The point to understand is that PRINT# and INPUT# are largely mirror images of each other. PRINT#1,X$:PRINT#1,Y writes a string, then a numeral, to a file; INPUT#1,X$,Y will interpret this correctly, reconstructing X$ and Y. If the variable types match, there should be few problems.

There are several small complications, all of which have been mentioned already, but they are worth going over again.

INPUT# cannot input a string more than 88 characters long.

INPUT# looks for a separator, normally a RETURN or a comma. Thus, PRINT#1,X$;Y cannot be read back properly by INPUT#, because the semicolon causes the two variables to be output with no break. It's easiest to separate the variables by a RETURN (CHR$(13)), but PRINT#1,X$",''Y works just as well.

INPUT# cannot input a null string. PRINT#1,X$: PRINT#1,Y$ then INPUT#1,X$,Y$ ordinarily works, but if Y$ is nothing, PRINT# puts two consecutive RETURNs on file, and INPUT# behaves as though RETURN were pressed on INPUT at the keyboard and goes on to the next item.

GET# reads individual characters from a file, with no exceptions. It will fetch null characters written as CHR$(0), quote marks (ASCII 34), RETURNs (ASCII 13), plus any punctuation and screen-editing characters. If you are interested in the entire contents of a file, use this command; if not, the intelligence of INPUT#, which assigns all your variables for you, makes a better command.

CLOSE. Closing a file is simple:

CLOSE filenumber

CLOSE operates on one file only; you need CLOSE 2: CLOSE 15 if files 2 and 15 are open. Unclosed files can cause problems. See the section "When to Ignore the Red Warning Light" for a full discussion.

ST (Status) and Disk Errors and Messages

ST has several applications in disk file handling. When INPUT# reads to the end of a file, ST is set to 64. ST can be tested for this condition if the file length is uncertain. After 64, ST becomes 66, which means the device isn't responding.

Two other possibilities are ST=−128 (usually accompanied by ?DEVICE NOT PRESENT) and ST=1 (if writing is slow). The other four bit-settings of ST don't apply to disk. ST isn't usually important, because an end-of-file marker makes ST=64 superfluous and the other errors are generally obvious. However, a command like IF ST>0 THEN GOTO EXIT provides a rough-and-ready exit mechanism when testing files. If you do this, remember that ST is reset after every input or output, so put the test immediately after the relevant command.

Disk messages nearly always indicate that a program can't run. The exceptions are message 1, the number of files scratched, and message 50, RECORD NOT PRESENT, which always occurs when a relative file is set up. The error may not be serious—for example, a syntax error in a command string—but it's good practice to follow each disk command with a subroutine call to read channel 15 and exit if the message number is 20 or more. The subroutine should print the message number and its message, and close all open files, as the following example shows:

```
10000 INPUT#15,E,E$,T,S: IF E<20 THEN RETURN
10010 CLOSE 2: CLOSE 15: PRINT E;E$;T;S
```

Note that this subroutine slows processing (especially after GET# statements) and can be ignored in ordinary, noncritical programming.

When to Ignore the Red Warning Light

Newcomers to Commodore disks are often concerned with the red warning light. Does it mean that something horrible has happened to the disk? Usually not.

The red light combines several functions, mostly informative rather than warning. A steady light means that a file is open. Try for instance OPEN 2,8,2,"TEST,W" in direct mode. The drive will start, and a write file will be opened to the disk. When the disk stops (the motor runs on for a few seconds to reduce delays when there are repeated disk accesses), the light remains on. Enter PRINT#2,"HELLO" then CLOSE 2. As with a tape file, the data is stored in a buffer; it's only written when the file is closed. A directory of the diskette shows FILE with type SEQ, occupying one sector only because there's such a small amount of data.

Any read/write activity causes the light to turn on mainly as a warning not to interrupt the process by opening the drive door. However, this is important only if there's a file writing to the disk.

A flashing light indicates an error message (scratching files generates message 1, but in that case the light doesn't flash). In fact, the number of flashes varies with the type of error, though not in a very useful way. The message can be read (by RUN 1000 with 1000 INPUT#15,E,E$,T,S: PRINT E;E$;T;S: END); once it is read, the light goes off and the message is cleared.

You can ignore the flashing red light if you are reading from disk. Suppose you've typed LOAD "PROGARM",8 in error; the red light flashes, and the message is something like ?FILE NOT FOUND. Type the correct version LOAD "PROGRAM",8 and loading will proceed normally with no problems. The same sort of thing obtains in a program; 10 OPEN 2,8,2,"FIEL,S,R" will generate an error, but if the line is edited and the program rerun, no harm will result.

Take the red light seriously if you are writing to disk and a write file is still open. An unclosed file can cause problems with storage to disk, because the normal system of chaining between sectors is disturbed; other programs and files can become corrupted. This isn't likely to be a major problem. However, if you are using a file system for a serious purpose, you should be aware of this possibility, since there will almost inevitably be program crashes during testing. When programs are finally completed, it is good practice to transfer them to new disks to avoid any chance of error. The steps to take, and danger signs to watch for, are listed in the next section's notes.

Handling Program Files

Program files are marked PRG in the directory. They are used for storing BASIC programs in tokenized form and ML or graphics as simple consecutive bytes. There's no way of telling from the directory whether PRG is BASIC or not; if LOAD "NAME",8 and RUN works, then it is BASIC, at least in part. ML programs usually need a SYS call to run.

PRG files can be opened for read or write. If such a file is read, the first two bytes are invariably the LOAD address, and the rest is the data. LOAD "NAME",8,1 always loads into this LOAD address, but LOAD "NAME",8 allows relocation (and also relinks the program, assuming it to be BASIC). There's no way to force a program file to load where you want with LOAD "NAME",8.

OPEN 2,8,2,"NAME,PRG,WRITE" opens a program file for write, while OPEN 2,8,2,"NAME,PRG,READ" opens the same file for read. There are, of course, variations on this. For instance, the file numbers and channel numbers needn't be 2; the device number may not be 8; and the command string can be made up of string expressions. In addition, the command string can be abbreviated such as ,P,W for ,PRG,WRITE.

Program 15-2 is a short program that reads any program byte by byte, printing out the results in ASCII.

Program 15-2. Reading Programs Byte by Byte

```
Ø OPEN 15,8,15,"I":REM INITIALIZE DISKETTE
1 OPEN 2,8,2,"NAME,P,R":REM OPEN PROGRAM FILE FOR READ
2 GET#2,X$:REM GET A CHARACTER FROM FILE
3 IF ST>Ø THEN CLOSE 2:END:REM END WHEN FILE ENDS

4 PRINT ASC (X$+CHR$(Ø));:REM PRINT ASCII VALUE

5 GOTO 2:REM CONTINUE ONE CHARACTER AT A TIME
```

Line 1 must include the name of the program to be examined. Alternate forms of the command string such as OPEN 2,8,2,"NAME,PROGRAM,READ" or OPEN 2,8,2,N$+",P,R" are perfectly acceptable.

Run this with a BASIC program as its PRG file, and you'll get BASIC in its tokenized form. For instance, if you save a one-line program (10 PRINT"HELLO") and then look at it using this program, you'll get something like Figure 15-2.

Figure 15-2. Tokenized BASIC

LOAD Address	Link Address	Line Number	Tokenized Line PRINT "HELLO"	End of Program
1 16	14 16	10 0	153 34 72 69 76 76 79 34 0	0 0

Uses for Program File Processing

Analyzing BASIC. Provided allowance is made for link addresses, line numbers, and the tokenized form of keywords, BASIC programs can be read, perhaps to see if they're identical. Hidden code can be searched for. Appending, deleting, and similar manipulation are possible. The link address need not be correct, since LOAD will put it right.

It's possible to write BASIC direct to a PRG file, by opening a program file for write, printing any two bytes as the start address (they'll be overridden when the program loads), and printing a further series of CHR$(n) commands to make up the program. This can be useful in some antilisting techniques, and BASIC lines longer than 88 characters can be written in this way too.

Finding ML or memory dump LOAD addresses. This can't be done in direct mode. Instead, use the following routine:

Program 15-3. Finding ML or Memory Dump Load Addresses

```
10 INPUT "PROGRAM NAME";N$
20 OPEN 1,8,2,N$+",P,R"
30 GET#1,X$,Y$
40 PRINT ASC(X$+CHR$(0))+256*ASC(Y$+CHR$(0))
50 CLOSE 1
```

Analyzing machine language programs. You've seen how to read the two LOAD address bytes. If you wish to load ML into a different area, you can change the LOAD address by rewriting the two leading bytes using the routine in Program 15-4.

Program 15-4. Changing the LOAD Address

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 OPEN 2,8,2,"ML FILE1,P,R"                :rem 219
20 OPEN 3,8,3,"ML FILE2,P,W"                :rem 228
30 GET#2,X$,X$                              :rem 232
40 PRINT#3,CHR$(1)CHR$(4);                  :rem 183
50 GET#2,X$:IF X$="" THEN X$=CHR$(0)        :rem 114
60 S=ST:PRINT#3,X$;                         :rem 157
70 IF S=0 GOTO 50                            :rem 84
80 CLOSE 2:CLOSE 3                          :rem 243
```

Lines 50 through 70 transfer the entire file, except for the first two bytes. The old LOAD address is thrown away; the new is set at $0401. Note in line 50 how a character which GET# regards as null must be converted into CHR$(0); otherwise, zero bytes will be lost. Line 60 preserves ST, which is reset by the PRINT# command, for the end-of-file test in line 70.

If you change disks, you may need to initialize the new disk (or add line 0 OPEN 15,8,15,"I": CLOSE 15 to the program).

PET/CBM programs load at $0401, so VIC programs can be made to load into these machines with this program. Occasionally, you may even want to use the program to restore LOAD addresses to programs which have lost them through incorrect copying.

Writing machine code or graphics definitions directly onto disk. As with BASIC, there's no problem in opening a program file, writing a two-byte LOAD address, and following this with bytes. For example, where RAM is already occupied by machine language or BASIC, or in tricky areas like zero page or the screen, this technique allows any area of RAM to be saved to disk. An autorun routine, analogous to those used for tape, provides an illustration.

Autorunning program. This is trickier with disk than with tape. If the start of BASIC is fixed, extra ML can be added to the start of the program to cause it to run automatically after loading. Alternatively, and for greater versatility, you can use a loader which calls the program by name, and so allows for variations in starting address.

Program 15-5 autoruns ML programs, which therefore need no SYS call. The forced LOAD address is $02A1. The autorun feature is caused by changing the vector at $0302/0303 to $02A1, where the ML program is loaded by name (in effect, with LOAD "ML",8,1) and then jumped to. In order to use this program, you'll need to choose a new name for the loader, such as RUN. Then, LOAD "RUN",8,1 will automatically load and run the ML.

Program 15-5. ML Autorun

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 FOR J=673 TO 698: READ X: POKE J,X: NEXT:rem 23
20 PRINT "NAME OF NEW LOADER?": INPUT L$     :rem 3
30 PRINT "PROGRAM TO BE RUN?": INPUT P$   :rem 232
40 L=LEN(P$): POKE 683,L                   :rem 19
50 FOR J=1 TO L: POKE 700+J,ASC(MID$(P$,J))
                                          :rem 245
60 NEXT                                   :rem 165
70 OPEN 2,8,2,P$+",P,R"                   :rem 150
80 GET#2,X$,Y$: CLOSE 2                   :rem 208
90 POKE 699,ASC(X$+CHR$(0))               :rem 189
100 POKE 700,ASC(Y$+CHR$(0))              :rem 213
110 OPEN 3,8,3,L$+",P,W"                  :rem 196
120 PRINT#3,CHR$(161)CHR$(2);              :rem 75
130 POKE 770,161: POKE 771,2              :rem 139
140 FOR J=673 TO 771                      :rem 229
150 P=PEEK(J): X$=CHR$(P)                 :rem 120
160 PRINT#3,X$;: NEXT                     :rem 214
170 CLOSE 3                                :rem 65
180 POKE 770,131: POKE 771,196            :rem 251
1000 DATA 169,1,162,8,160,96,32,186,255,169:rem 79
1010 DATA 0,162,189,160,2,32,189,255,169,0 :rem 15
1020 DATA 133,10,32,213,255,76            :rem 181
```

Notes on the program. RAM from locations 673 through 771 is written as an ML file, preceded by two bytes to force the LOAD address to $02A1. The ML is POKEd in by line 10. It's fairly straightforward to add new features, for example to make the background and current color both white, suppressing the next LOADING message. Lines 40–60 put the name of your ML program in RAM, so the same ML program will always be loaded (assuming it's on the same disk). Lines 70 and 80 read the ML program's LOAD address; this assumes the program's SYS address is the same as its LOAD address. If this isn't true, just substitute the correct LOAD details; for example, if SYS 64802 runs your program, then change lines 90 and 100 to POKE 699,34 and POKE 700,253.

The loader is created in lines 110–170. Line 120 sets the start address to $02A1. Line 130 changes the warm start vector to $02A1, and lines 140–170 print the whole of the RAM area as a program file.

PRG files occupy 254-byte sectors on the disk; the first two bytes are the LOAD address. Thus, 252 bytes go into the first sector, while 254 bytes go into the remaining sectors. An 8K program (8192 bytes) therefore occupies approximately 32¼ sectors, which appear as 33 sectors on the directory. The number of sectors taken up by any program or memory dump can be similarly calculated.

Handling Sequential Files

Sequential files are marked SEQ in the directory. They are easy to use and can store large quantities of data. The records are free from length restrictions, subject to the 88-character limit if INPUT# is used for reading, so there's no space overhead apart from separators like RETURN characters. In sequential files, records are likely to be stored in similar sets—for instance, name followed by four address lines and a phone number—so there are no problems in interpreting data when it is read back from the file.

Sequential files, once written, aren't readily changed, but new records can easily be added onto the end. The disk operating system (DOS) has a built-in append command. Files can be updated only by reading, correcting records as they are read, then writing back the edited version (with old records removed and new ones inserted) as a new file with a different name. This process, which is impossible on VIC tape, is quite easy with disks.

The DOS has another command, COPY, which copies a sequential file onto the same disk and optionally concatenates another file on the end. This is more useful with CBM's double-disk units, but still has a few uses with the VIC.

Use OPEN 2,8,2,"FILENAME,SEQ,WRITE" to open a sequential file for write operations. OPEN 2,8,2,"FILENAME,SEQ,READ" opens the same file for read; OPEN 2,8,2,"FILENAME,SEQ,APPEND" opens an existing file for append.

The file and channel numbers needn't be 2, and the device number may not be 8; there are alternative, similar forms. Sequential files are assumed by default, so if SEQ or the shorter S is omitted from any of these commands, it makes no difference. "Read" is a further default, so OPEN 2,8,2,"FILENAME" assumes a sequential file will be read, and reports an error if the file isn't found.

A sequential file can be opened for write only once. Thereafter, data can be appended, but an attempt to open it again for write using the same file number will cause a FILE EXISTS error message within the disk drive. However, using a different file number erases the file and starts over.

COPY has the following syntax:

OPEN 15,8,15,"COPY:*new name=old name*"

or

PRINT#15,"COPY:*new name=old name*" (after OPEN 15,8,15 has been carried out)

This command writes another copy of the file, under a different name (or you'll get FILE EXISTS), to the same disk.

OPEN 15,8,15,"COPY:*new name=first file,second file*" combines two (or more, if you wish) named files into another; again, the combined file must have a new name.

Program 15-6 is a simple example of a program that reads a sequential file and displays its contents onscreen. Note that ST in line 60 tests for the end-of-file condition. If that line is omitted, nothing very terrible happens; however, line 40 will then repeatedly fetch a meaningless character.

Program 15-6. Reading and Displaying a Sequential File

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 PRINT"NAME OF SEQ FILE TO BE DISPLAYED":rem 186
20 INPUT N$                                :rem 100
30 OPEN 1,8,2,N$+",S,R":REM OPEN SEQ FILE FOR READ
                                           :rem 238
40 GET#1,X$:REM FETCH A SINGLE CHARACTER FROM FILE
                                           :rem 172
50 PRINT X$;:REM PRINT A CHARACTER          :rem 34
60 IF ST>0 THEN CLOSE 1:END:REM STOP AT END OF FIL
   E                                       :rem 118
70 GOTO 40                                   :rem 4
```

You may find that the disk warning light flashes if you misspell the file's name or try to read a program rather than a sequential file. However, incorporating a test for these messages is straightforward. First, add the following line:

5 OPEN 15,8,15

It is good practice to close file 15 last; if it is closed during a program, other disk files will close too. Then add this subroutine:

10000 INPUT#15,E,E$,T,S: IF E<20 THEN RETURN
10010 PRINT "*DISK WARNING": PRINT E;E$;T;S: CLOSE 15**

Whenever the disk is accessed, a call to this subroutine will test that all's well. Add 35 GOSUB 10000 to check that the file was opened properly; add 45 GOSUB 10000 to check each read from the file. This slows processing, of course.

The program is now almost ready, but one further subtlety is possible. ST is changed by the new line 45 to reflect the status of input from the command file rather than the sequential file, so you can add another line (42 A=ST) and change ST in line 60 to A; the program then tests for all error conditions.

If the warning light flashes, it can be ignored with this program since no files are being written.

If you prefer to see every file character (for example, RETURN showing as 13), replace X$ in line 50 by ASC(X$+CHR$(0)).

Writing Sequential Files

Writing a sequential file is straightforward. It's similar to reading, except that PRINT# is used and the file must be opened with the W parameter. You can see how this works by typing in OPEN 2,8,2,"SEQ TEST,W" in direct mode. On RETURN, assuming the file doesn't already exist, the disk shows activity; when it stops, the red light remains on because a file is open. Enter PRINT#1,"HELLO" and note the absence of activity. CLOSE 2 writes this data to disk and closes the file. The previous program will read back the five letters of HELLO plus a final RETURN character.

Repeating the same command causes a FILE EXISTS disk error. If the file isn't important, OPEN 2,8,2,"@:SEQ TEST,W" will open a new file with the same name.

Appending Sequential Files

Appending means adding new data onto an already existing file. To append to a sequential file, use OPEN 2,8,2,"SEQ TEST,A" which reopens the file, leaving the red light on. PRINT#1, "GOODBYE":CLOSE 1 writes an extra record; again this can be checked by reading.

Sample Uses for Sequential File Processing

Storing records. If a file is to be written once only, open it for write, use INPUT from the keyboard, then PRINT# to write to the file and CLOSE the file. Where a file is to have records added from time to time, but none removed or altered, it's easiest to set up the file first, then open it for append whenever it's needed, INPUT the new data, and PRINT# to the file.

Where a file is to be edited, however, use two files—perhaps "NAME"+STR$(N) followed by "NAME"+STR$(N+1). With this scheme, there will always be a file called something like "NAMES/PHONES 33" on disk. The file-editing program will ask for the update number (33 in this case) and open the earlier version for read and the later version for write. Alternatively, you may prefer to rename the existing file OLD and write to NEW.

The file OPEN commands have the following form:

```
OPEN 2,8,2,"OLD":OPEN 3,8,3,"NEW,W"
```

INPUT#2 takes data from OLD, while PRINT#3 writes it to NEW.

For security, add a channel-reading subroutine which closes files if an error is detected.

Dealing with BASIC. There's a close connection between SEQ and PRG files. Commands to append, concatenate, and copy all work with program files, although the results don't always appear similar because BASIC uses three zero bytes as terminators. Thus, appending like this can only work if two of these bytes are thrown away. BASIC can be written as a sequential file by opening a write file (for instance, file #1) and using CMD 1: LIST and CLOSE 1 to print the program to the file. BASIC stored in this way is not tokenized and is generally longer than its normal equivalent. But this storage method allows for fairly easy program analysis. Cross-reference tables of variables by line numbers are a typical application.

As you'll see, the directory track can be read as a file, and this gives a lot of information about the way files are stored. Using SAVE "PROGRAM,S,W",8 it's even possible to save programs as SEQ files.

Copying files. SEQ files can be copied for security either with a CBM 4040 disk drive or by reading the data, storing it in RAM, and writing it back onto a new disk. If the file is long, of course, this method is impossible; in such a case the best compromise is to write to a tape file, which obviously has no space restrictions, then read back and write to the new disk.

SEQ files occupy 254-byte sectors. Add together the lengths of all the strings of data, including RETURNs, divide by 254, and round up to estimate the storage requirement of any SEQ file.

Handling Relative Files

Because of their highly structured format, relative files (REL in the directory) allow both reading and writing in the same open file. Every record is assigned a set length, which cannot be exceeded. Shorter records are automatically padded with null characters. Thus, when updating a record, it is important to write back the entire record, or the final part will be erased. For example, WILLIOMS must be printed back as WILLIAMS, not as an A at the sixth position.

Relative files are referred to by number. The DOS uses a "P" parameter to transfer the record number to disk.

Whenever a record is written beyond the present end of file, message 50 RECORD NOT PRESENT is generated. The first time around, this can be ignored. You've seen already that it's a good idea to format the entire file right at the start, assuming the number of records needed in the complete file is known. This sets up each record as CHR$(255), so reading back an empty file lists each record as a π symbol.

A relative file's data is stored like a sequential file (with ASCII characters separated by RETURNs), but has an extra file of pointers. A maximum of three disk reads is needed to read a record with this system, which is therefore often slower than a sequential file, which never uses pointers. Of course, for random access, relative files are faster than any but the shortest sequential files. Records are stored in a disk buffer, so reading or writing adjacent-numbered records often requires no disk access time.

Use OPEN 2,8,2,"*filename*,L," + CHR$(L) to open a relative file. As usual, the file number and channel may take a range of values, and the device number may not be 8. L is compulsory when the file is set up for the first time; it is followed by the record length, which must allow for a RETURN character. For example, use CHR$(21) if the longest record has length 20.

The record length parameter is stored on the diskette. If you attempt to reopen the file with a different record length, error 50 RECORD NOT PRESENT shows. The maximum record length is 254. Anything beyond this gives error 51 OVERFLOW IN RECORD.

Once the file has been opened, the L and parameter are optional and a simple OPEN statement with the name is sufficient. It makes sense to use the full version, though, in case you forget the record length.

Obviously, the number of the record which is about to be read or written must be sent to disk. The syntax is a bit fussy: PRINT#15,"P"+CHR$(channel) + CHR$(low byte) + CHR$(high byte) + CHR$(position) assuming OPEN 15,8,15. The channel parameter is identical to the channel used in OPEN, 2 in the example. The low- and high-byte idea is familiar; thus record number 200 needs PRINT#15,"P"+CHR$(2)+CHR$(200)+CHR$(0)+CHR$(1).

The final parameter is a pointer, with 1 representing the start of the record. It allows writing or reading to take place a set distance within a record. Obviously, it shouldn't exceed the record length. It is usually 1. Don't omit it.

The pointer allows records to be subdivided, so that a 200-byte record might have several fields (for instance, starting at 1, 40, and 100). In practice, each field can be written within its record sequentially, without bothering with this, because

PRINT#, INPUT#, and GET# each advance the pointer as they write or read into the file buffer. In any case, writing to such a record requires that everything up to the final record be written. If only the field starting at 40 were written, the field starting at 100 would be erased.

Program 15-7 is an example of relative file handling. It asks for record numbers, then for the data to be input, and writes the record to disk. It does not include a message channel.

Program 15-7. Handling Relative Files

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 OPEN 15,8,15                                :rem 239
20 PRINT "REL. FILENAME": INPUT N$             :rem 193
30 PRINT "RECORD LENGTH": INPUT L              :rem 203
40 OPEN 1,8,2,N$+",L,"+CHR$(L+1)                :rem 95
50 INPUT "RECORD#";R                           :rem 168
60 IF R=99999 THEN CLOSE 1: END                :rem 136
70 RH%=R/256: RL=R-RH%*256                     :rem 155
80 PRINT "RECORD": INPUT R$                     :rem 56
90 R$ = LEFT$(R$,L)                            :rem 170
100 PRINT#15,"P"+CHR$(2)+CHR$(RL)+CHR$(RH%)+CHR$(1
    )                                           :rem 27
110 PRINT#1,R$                                  :rem 21
120 GOTO 50                                     :rem 49
```

Line 40 opens a relative file, named by the user, and assigns a record length 1 greater than the input value (to allow for a RETURN at the end). Line 60 allows record number 99999 to act as an indicator that no more data is to be entered. Lines 80 and 90 take in the material to be written to disk, and check that it isn't too long. Line 100 sets the record number parameters from the channel number and record number. Line 110 finally puts the record onto disk. This is the simplest case, where the record starts at the beginning of its allotted space.

Channel 15 can be read as usual. A subroutine call in a new line 45 will check the OPEN, and lines 105 and 115 can also be added to test the command and the print.

Remember that message 50 signals that the file is being extended, and therefore it should be expected when the file is being set up.

Reading the file. The easiest way to read the file you've just created is to modify the write program, delete lines 80 and 90, and alter 110 to INPUT#1,R$: PRINT R$. Line 30 can be removed, and 40 OPEN 1,8,2,N$ is sufficient.

The same file is usable for either reading or writing. Typically a program will have a menu allowing either mode to be selected.

Copying. Use OPEN 15,8,15,"C:*new name*=*old name*" to copy this type of file onto the same disk with a new name. This provides some security. Apart from direct disk copying, or putting data into RAM (where there may not be enough available room), copying records to tape by reading them in sequence, then writing them back to a different diskette, is the easiest method.

Storage. The total file length is the L parameter multiplied by the largest record number plus one (since record zero exists). 1000 records of length 21 occupy 21000 bytes; these are stored in 254 byte sectors, so the data occupies 83 sectors. (VIC disks have 664 free sectors.) Additionally, the side sectors containing the pointers occupy from 1 to 6 sectors, depending on the file length. The actual number is the file length in sectors divided by 120; so our example file needs one side sector only, and the entire file takes 84 sectors.

A relative file which fills the entire diskette takes about five minutes to set up when first written.

Note that some versions of CBM DOS aren't reliable in validating disks with relative files; however, 1540/1541 DOS does not have this fault.

Summary of Disk Commands, Disk Messages, and Error Handling

The following list summarizes the VIC disk drive's file and program commands. All the syntax examples are illustrations only; modifying them to your own requirements where necessary isn't much work.

Append. Used mainly for sequential files, append opens an already existing sequential file for write. New records are added to the end of the existing file. The syntax is OPEN 2,8,2,"*filename*,A" then PRINT#2 and CLOSE 2.

Copy and Concatenate. Used mainly for sequential files, these commands create a new file with a new name, consisting of a copy of just one file or of several concatenated files. OPEN 15,8,15,"C:NEW=FIRST,SECOND,THIRD" combines three files in sequence in the new sequential file called NEW.

Directory. This command lists any diskette's contents. Its form is LOAD "$",8 then LIST. Getting a directory in this way will destroy any BASIC program in memory, so during program development, *never* read the directory unless you've saved the current version. VIC's wedge program on the demo disk reads the directory directly into the screen and can therefore be used during program development.

Initialize. Usually automatic, this command reads the present diskette's storage details into disk RAM. If diskettes are changed, it's always safest to initialize the new diskette with OPEN 15,8,15,"I":CLOSE 15. The command is necessary in some programs reading directly from the diskette and provides a means to get the drive working again in occasional anomalous situations. The VIC's RAM is unaffected.

New. New, used to format all new disks, has been explained earlier. The syntax is OPEN 15,8,15,"N:*name,id*" for a brand-new diskette.

OPEN, CLOSE. OPEN and CLOSE were discussed in the sections on program, sequential, and relative files, and in the discussion of channel 15. Typical BASIC is as follows:

OPEN 15,8,15,"COMMAND STRING" to the command channel
OPEN 2,8,2,"SEQFILE,S,W" to open SEQFILE for write
OPEN 3,8,3,"@:PROGRAM,P,W" to scratch and reopen PROGRAM for write
OPEN 4,8,4,"RELFILE,L,"+CHR$(101) to open RELFILE
CLOSE 2: CLOSE 15 to close two files.

Record#. For relative files only, this sets the record number and position within the record from which write or read will take place. Typical syntax is PRINT#15,"P"+CHR$(4)+CHR$(LO)+CHR$(HI)+CHR$(1) where OPEN 15,8,15 is assumed and the relative file uses channel 4.

Rename. Rename changes the name of any type of file. It has the syntax OPEN 15,8,15,"R:NEW=OLD":CLOSE 15 where the file previously called OLD is now called NEW. Only the name is changed; a duplicate file is not created.

Scratch. Scratch deletes any type of file by name, using pattern matching. Typically it uses the form OPEN 15,8,15,"S:*filename*":CLOSE 15.

Validate. This command checks disk integrity. It tests and collects together all the disk sectors' chaining. This is safe with any type of closed file but will erase unclosed files; its syntax is OPEN 15,8,15,"V":CLOSE 15.

Use this command whenever disk writes have been interrupted, for example, by a syntax error in BASIC. However, if you have an incomplete file you wish to save, first follow the instructions in the next section. If your disks are used only to load programs, this command is unimportant; it is significant only when you're developing file-handling programs and is unlikely to be needed with straightforward LOADs and SAVEs.

Pattern Matching

Disk commands involving loading or opening for READ generally can be abbreviated using * and ? as pattern-matching symbols. For example, LOAD "A*",8 loads the first PRG type file in the directory which begins with A. However, LOAD"*",8 and VERIFY "*",8 assume the last loaded program applies, unless no program has been loaded, in which case the first program loads. Similarly, OPEN 2,8,2,"S:X*" :CLOSE 2 scratches all files beginning with X.

The ? symbol allows wild card matching, but the exact positions have to match; LOAD "????BON*",8 loads TROMBONE, but not BONZO.

Because of the possibility of sending spurious disk commands, you should not include symbols like * ? # :, in filenames.

Disk Messages

Table 15-1 summarizes the messages generated within the disk drive.

Problems with Disk Drives

Unresponsive drive. Sometimes you'll get ?FILE NOT FOUND, even with LOAD "$",8. Try again. If that doesn't work, open the disk door and close it, then try again.

You may also get ?DEVICE NOT PRESENT when the disk is switched on and ready. Try initializing, or if the red light is lit, OPEN 15,8,15: CLOSE 15. Sometimes a printer causes this hangup. Try turning your printer off.

File problems. Unclosed files are signaled with an asterisk (for example, *SEQ). However, some aborted files don't have this. You may have a situation where a program occupies two sectors, even though its file is reported as occupying only one, and there are only 618 blocks free instead of 661 as expected. In each case it's ultimately best to validate, but you could either leave the disk alone, using it only for READ, or recover some of the file data using OPEN 2,8,2,"FILENAME,N" which enables unclosed files of all types to be read as far as possible.

To avoid this sort of problem, simply make a point of closing write files if a program crashes before they are closed properly. Enter CLOSE 2 in direct mode, for

Table 15-1. Summary of Disk Drive Messages

Message Type	Information (not an error)	Programming Mistake or Simple Mechanical Error	Hard Error
	0 Everything OK 1 Files scratched (gives number) 2–19 Undocumented. Not important.		
Input/ Output Errors at Disk Level			20 Sector header not found 21 Sync mark not found 22 Sector not found 23 Checksum error in byte 24 Byte read error 25 Readback compare error 26 Write-protected diskette 27 Checksum error in header 28 Next sync mark not found
Initialization		29 Disk ID/BAM mismatch	
Syntax Errors		30 Syntax error 31 Unrecognized command 32 Overlength command 33 Wrongly used ? or * in name 34 File name omitted 39 Unrec. command to channel 15	
Relative and Seq. Files	50 Expand rel. file size	50 Relative file parameter error 51 Relative record too long 52 Relative file too big for disk	
File Errors		60 Attempt to read a write file 61 File not open 62 File doesn't exist 63 File does exist 64 File type mismatch	
Track and Sector Errors		65 Block-Allocate error: gives next available track & sector 66 Track or sector out of range 67 System track or sector error	
DOS Errors		70 Channel to disk unavailable 71 Error in directory 72 Disk (or directory) full 73 DOS type message 74 Drive not ready	

example, and include a CLOSE statement for channel 15.

OPEN 15,8,15:CLOSE 15 will generally close all disk files successfully.

Program problems. Sometimes the final sector of programs becomes corrupted; on LOAD the program loads, but READY never appears. The best solution is to press RUN/STOP–RESTORE, then POKE zeros into memory after the end of the program.

Commodore Utility Programs

Commodore's demo disks contain a number of programs. Those listed below are typical of what you will find.

CHECK DISK

Tests a diskette by writing to and reading from every sector.

COPY/ALL

A BASIC program, written by Jim Butterfield, to copy an entire disk from one drive to another. Two drives are necessary. One drive can be reassigned device number 9 with DISK ADDR CHANGE.

DIR

Reads the directory of device 8 from BASIC. No advantage over ordinary directories.

DISK ADDR CHANGE

Writes a new device number through the command channel, usually 9, to permit interdrive copying.

DISPLAY T&S

Displays any track and sector on the diskette. Very useful for examining the disk's entire storage system or (in extreme cases) for reading programs or files directly.

DOS 5.1

For the Commodore 64.

PERFORMANCE TEST

Formats a disk, writes, and reads, but doesn't exhaustively test either diskette or drive.

PRINTER TEST

For CBM printers only.

VIC-20 WEDGE

LOAD and RUN this to make direct-mode disk commands simpler. The ML program stores itself at the top of BASIC RAM; the wedge will not coexist with some other utilities.

It adds three direct-mode commands. One of them (@) reads and prints the disk status; another (@$) reads and displays the directory without affecting the program in memory. A third command (*/filename*) loads a program from disk.

VIEW BAM

Prints a diagram of the Block Availability Map.

Hardware Notes

1540/1541 Disk Drive Units

These drives contain a transformer to supply power, a printed circuit board containing the ROM, RAM, and interface chips which hold the disk operating system, and a drive unit, which is positioned away from the heat-generating components. Some models have metal shielding over the printed circuit board to reduce radio frequency emission. Both the shielding and the top half of the outer casing are easily removed (for example, to exchange a 1540 ROM for a 1541 ROM or to change the device number from 8). The design is similar to earlier Commodore disk drives, the 2031 single disk and 4040 double disk.

The device number can be set to any number from 8 to 11. At least four drives can be daisychained together, so in principle, a four-drive system would be feasible. Given the right hardware, a single VIC can also share a disk drive with other VICs.

The read/write head faces up, so the underside of the diskette is the active side. Closing the door brings a pressure pad down on the head, keeping it in close contact with the diskette. During read/write operations, the diskette is rotated by the spindle motor at about 300 revolutions per minute, and centrifugal force gives the diskette some rigidity. The head itself is mounted on rails and can move, along with the pressure pad, a maximum of about one inch. Movement is handled by a stepper motor. Each step moves the head about 1/30 inch.

These drives use 35 tracks. The actual magnetized zones are about 1/60 inch wide; the clutch mechanism which grips the diskette has to position it within that tolerance.

Head alignment problems sometimes occur, in which diskettes work on one disk drive but not on another, because the heads aren't quite in the same place relative to the disk center. Special alignment diskettes, having very slightly elliptical tracks, allow a disk drive head to be accurately repositioned. Realigning disk drive heads is specialized work.

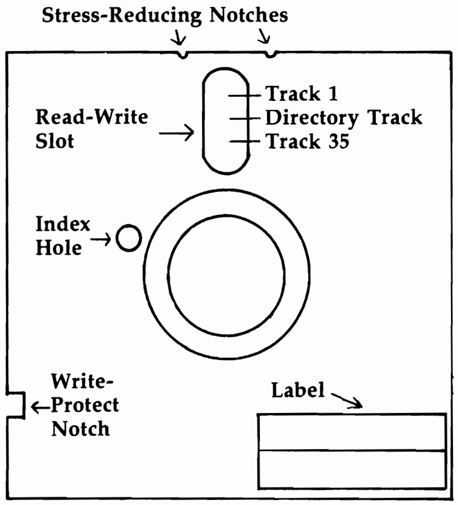
Diskettes

VIC disk drives use 5¼-inch diskettes ("floppy disks"). Any good-quality, single-sided, single-density diskettes are fine. Soft-sectored diskettes are generally used, but hard-sectored disks will also work well, as their index hole isn't used by the drives.

Write protection is readily implemented with 1540/1541 drives. An adhesive tab over the notch prevents writing to the disk. Attempting to write to such a disk returns 26 WRITE PROTECT ON.

Diskettes are inserted label up, read-write slot foremost. Diskette labels are deliberately positioned away from the slot, to reduce the chance of fingerprint damage and to allow the label to be read when the diskette is in its dust cover. Writing

Figure 15-3. A Typical Diskette



on the label with a sharp implement—for instance, a ballpoint pen—may damage the diskette surface below. Always write on the label before putting it on the disk.

It is good practice to open the drive door when drives are turned on or off. There's some small chance of magnetic "glitch" damage to a diskette that's left in a drive, with the door closed, when power is turned on.

It's easy to modify diskettes so both sides are usable. The index hole isn't a factor, so all that's needed is to cut a notch in the diskette opposite the write-protect notch. The diskette then works on either side. However, that may not be desirable. The standard argument against this practice is that small particles of dust, smoke, and other debris, which become trapped by the self-cleaning wiper which lines the diskette, may be dislodged when the direction of rotation is reversed. In addition, some single-sided diskettes have defects on the back side. Nonetheless, quite a number of people do this successfully.

Diskette life is typically quoted as several million passes per track. At 300 rpm this represents about a week's *continual* running.

Track and Sector Storage System

All 1540/1541 units use 35 tracks, defined by the head positions. Track 18 is exactly midway between the edge and center of the disk, and it stores all the directory information, thus minimizing delays due to head movement. When a disk is formatted, the head moves to the outer track (track 1) end stop, then counts in one track at a time to 35. The same head movement to track 1 (making a rapid clicking sound) happens whenever there is a read error. This occurs because the head counts in until it arrives at its correct track, then tries reading again in case its position was wrong before.

A track is not a solid block of data. Instead, it is broken into 256-byte blocks called sectors. Any program or file is stored in sectors, and the first two bytes are always pointers to the next sector.

Sector storage tolerates some, but not much, variation in disk rotation speed. If the disk spins too fast, sectors will overlap and data will be lost. Typically, there's at least a one-second delay between starting the disk motor and writing or reading data. For this reason, the motor is left on for some time after an access, so if another access follows shortly, no time is lost waiting for the speed to build up.

Commodore's system uses more sectors on the outer tracks than the inner. This takes advantage of the fact that the outer circumference is greater than the inner, in the same way that other recording media usually give better resolution at the edge than near the middle. However, because the angular speed is constant, outer tracks must be written and read more rapidly than inner tracks, so hard sectoring is impossible.

Sectors are not written in sequence around the disk. If an entire track is filled with data from a single file or program, it's more efficient to chain sectors which are far apart on the disk, so that only half a revolution (rather than a whole revolution) is lost between reads or writes. A typical sequence on the outer tracks is 8, 18, 6, 16, 4, 14, 2, 12, 0, 10, 20, 9, 19, 7, 17, 5, 15, 3, 13, 1, and finally 11.

Sectors are stored with a short header, followed by data. Each part begins with a so-called "sync field" and ends with a checksum. The header contains 08, a two-byte ID, and the track and sector number. The data is preceded by 07. Messages 20–29 from the disk may indicate that some aspect of this elaborate error-checking system has failed. For example, if a magnet is held near the edge of a diskette, the outer sectors become unreadable. This technique can be used to protect disks from being copied.

The conversion of bytes into magnetic patterns on disk, and vice versa, is an analog hardware function, relying on crossover detectors, amplifiers, and pulse shapers.

Changing the Disk Device Number from 8

Device number 8 is set by hardware, and many programs using disk assume drive 8. Therefore, it is generally better to use software to alter the device number, even though the process has to be repeated whenever the drive is turned off. The exceptional case, where hardware change is desirable, occurs with a fairly permanent setup with two drives. In such a case, the change can be made permanent, or the disk unit can be fitted with a switch to select its device number.

Software conversion is easily done using CHANGE DISK ADDR on the demo disk. This program, which works with any Commodore disk, writes the new device number into two disk RAM locations. Commodore disks vary a great deal internally, so the program also has to work out the type of disk drive. With VIC, use OPEN 15,8,15:PRINT#15,"M-W"CHR$(119)CHR$(0)CHR$(2)CHR$(32+9)CHR$(64+9) :CLOSE 15 to convert from 8 to 9; the analogous statement will work for any other conversions within the range 8 to 14.

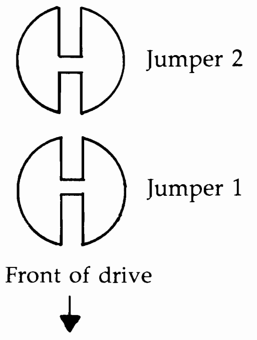
When two drives are used, they must be turned on separately. Typically one drive is turned on, then the VIC is turned on, then the live disk's number is changed and the second drive is turned on. In that way the system isn't confused. LOAD "$",8 and LOAD "$",9 load directories from the two drives.

"$",8 and LOAD "$",9 load directories from the two drives.

Hardware conversion involves cutting jumpers. These jumpers are not wires, but round spots of solder on the circuit board separated into halves, with a thin strand of solder connecting each half. You cut the jumpers by scraping away, or breaking, the connecting strand with a sharp knife.

The actual board layouts vary. The jumpers in the 1540 and early 1541 disk drives are located on the left side of the circuit board as you face the front of the disk drive. On the newer 1541 drives, the jumpers are in the center of the board. The early 1541 drives can be identified by their white cases, while the newer 1541 drives have brown cases. In both versions, jumper number 1 is nearest the front, and just behind it is jumper 2. Figure 15-4 shows the layout.

Figure 15-4. Changing Drive Numbers by Hardware Modification



Cutting only jumper 1 changes the device number to 9. Cutting only jumper 2 changes it to 10. Cutting both jumpers changes it to 11.

Disk ROM

Commodore disk drives have internal ROM from $C000 to $FFFF and RAM from $0 to $7FF. It's easy to disassemble disk ROM, because disk memory can be read with the following command:

PRINT#15,"M-R"CHR$(*low*) CHR$(*high*):GET#15,X$:X=ASC(X$+CHR$(0))

That assumes, of course, that OPEN 15,8,15 has been performed. The value X is equivalent to PEEKing the disk's memory. The low and high bytes of the location should be used in place of *low* and *high*. You can disassemble the ROM by replacing PEEK in a BASIC disassembler with this routine.

An alternative approach would be to look at the source code of the disk system, but Commodore is not likely to release this.

Disk ROM has the conventional 6502 features, including NMI, Reset, and IRQ vectors at the top of memory. It also has tables of error messages and tables of commands, some of which are undocumented.

Minimizing Errors

To minimize errors, the general rules are simple: Keep the disk drive free of dust, smoke, and other contaminants; store and treat the diskettes properly; keep copies of programs and data; and so on. It's worth having a standby system if your VIC is used for any serious purpose.

Hard errors are rare; one bad bit in 10^{11} is typical of quoted figures. Errors caused by unclosed files are far more likely. With some systems, programs to validate data may be used. Such systems can be written to minimize disk use, favoring RAM where possible to minimize the probability of a mistake.

Disk Storage of Data

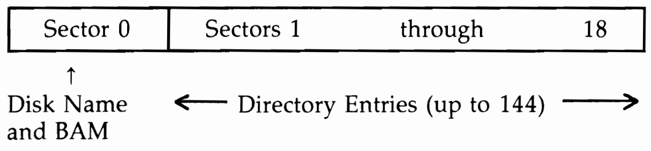
Commodore disks have 35 tracks. Of those tracks, 17 have 21 sectors each, 7 have 19 sectors each, 6 have 18 sectors each, and 5 have 17 sectors each. That gives a total of 683 sectors. Track 18 holds the directory information. Subtracting 19 for the directory gives 664 blocks free, as reported by the directory for an empty disk. 664 blocks of 254 bytes (excluding the track and sector pointers) gives 168,656 usable bytes. Relative files, as you've seen, require slightly more space; an entire diskette filled with a single relative file can occupy 658 blocks (167,132 bytes at most). Table 15-2 shows how the sectors are arranged on a disk.

Table 15-2. The Number of Sectors Per Track

Track Number	Sectors
1–17	0–20
18–24	0–18
25–30	0–17
31–35	0–16

The directory track, track 18, is diagrammed in Figure 15-5 and has 19 sectors. Sector 0 holds the disk name, as well as a bitmap of every sector on the disk, showing whether the sector is used or not. Sectors 1–18 store file type, filename, and pointers to the actual data. Each of these sectors can store eight filenames, giving a maximum of 144 directory entries.

Figure 15-5. Track 18, the Directory Track



Each time a file is written, the BAM (Block Availability Map) is updated, so the system knows which sectors are free for subsequent recording. VIEW BAM on the demo disk prints a diagram of this map. To see how this works, load DISPLAY T&S and inspect track 18, sector 0. Its layout is described in Table 15-3.

Table 15-3. Track 18, Sector 0 (BAM)

Byte Numbers	Track 18, Sector 0 (Directory Track)
0,1	Pointer to directory entries—track 18, sector 1
2,3	Disk format A
4–143	BAM (Block Availability Map): 35 sets of 4 bytes each
144–159	Diskette Name (16 characters maximum)
162–163	Diskette ID
165–166	2A (version of disk operating system)

(Omitted bytes are SHIFT-spaces, $A0, or spaces, $20. Remember, DISPLAY T&S prints values in hex.)

BAM

Each of the 35 tracks is represented by four bytes in the BAM, as shown in Table 15-4.

Table 15-4. BAM Organization

First Byte	Second Byte	Third Byte	Fourth Byte
Number of sectors free in this track [From 0 to 21]	Bits for sectors 7, 6, 5, 4, 3, 2, 1, 0 [0 = used, 1 = free]	Bits for sectors 15, 14, 13, 12, 11, 10, 9, 8 [0 = used, 1 = free]	Bits for sectors X, X, X, 20, 19, 18, 17, 16 [0 = used or unavailable, 1 = free]

For example, the first track may appear as

```
04: 15 FF FF 1F
```

which means that all 21 sectors of track 1 are free. 1F has bit pattern 0001 1111, meaning sectors 16 through 20 are unused. VIEW BAM picks through and displays these bit patterns. Note the way information is preferentially stored near the middle track to minimize head movement time.

Directory Entries

Directory entries are fairly straightforward. Use DISPLAY T&S on track 18 sector 1; you'll find it split into eight sets of 32 bytes each. Except for the first two bytes of the sector, which link to the next directory entry, the interpretation is shown in Table 15-5.

Table 15-5. Contents of a Directory Entry

BYTES	Contents of a Directory Entry
0–1	Track and sector pointer in first entry. Otherwise unused.
2	File Type. $ 0=Scratched/Not yet used $80=DELeted $81=SEQuential file $82=PRG, program file $83=USR, user file $84=RELative file $1–$4 signals an unclosed file. Such files are removed by COLLECT. $80 is a scratched unclosed file, a type to be avoided.
3–4	Track and sector pointer to first block of file
5–20	Filename + shifted spaces ($A0 characters)
21–22	Track and sector pointer to relative file's first side sector
23	Record size of relative file (i.e., parameter following L on opening file)
24–27	Unused
28–29	Replacement track and sector pointer for OPEN@
30–31	Low and high byte of no. of blocks in file, as shown on the directory

The first directory entry in track 18, sector 1 is as follows:

```
00:  12  04  82  11  Track 18, sector 4 next entry. File is PRG. It starts track 17
04:  00  48  4F  57  Sector 0 Name is: HOW
08:  20  54  4F  20                     TO
0C:  55  53  45  A0                     USE
10:  A0  A0  A0  A0  Name padded with SHIFT-space characters to length 16
14:  A0  00  00  00
18:  00  00  00  00
1C:  00  00  0D  00  Occupies 13 sectors
```

Relative files have slightly more detail than other file types, because of their index system. A track and sector pointer points to the first (of a possible six) side sector, which is linked like any other file and treated as a separate file by the operating system. The record length parameter is also stored here. If you've forgotten it, this is the place to look.

The side sectors have the structure shown in Table 15-6.

Up to 120 sectors can be stored in one of these blocks. The system calculates the effect of the record it is asked to read or write, by multiplying record length by record number, then calculates which sector the start of the record must appear in. In the worst case, a new side sector has to be loaded, a track and sector looked up in it, then finally the correct track and sector read. (If a record straddles two blocks, a fourth disk movement occurs.)

Table 15-6. Side Sectors in Relative Files

Bytes	Contents
0–1	Track and sector pointer to next side sector
2	Side sector number, 0–5
3	Record length of relative file
4–15	6 pairs of pointers to *every* side sector
16–255	120 pairs of pointers to consecutive sectors of data

Six side sectors can cover 720 blocks; this, of course, is enough for a file covering the whole diskette. However, in this case an extra channel (for a total of three) needs to be kept open within the disk: one for a side sector, one for a data sector, and a third for the data itself. A sequential file needs only two, one for the current sector and one for the correct data. VIC disk drives allow five channels, so two sequential files, or one relative file, or one of each type, can be open at the same time.

File and Program Storage

DISPLAY T&S allows any file or program to be examined byte by byte. First, the directory entry must be found in track 18. Bytes 3 and 4, immediately after the file type indicator, show the track and sector of the first block, and DISPLAY T&S has been modified from earlier versions to automatically read chained blocks, when required.

Program files (type $82) can be either BASIC or ML dumps. The first two bytes are the load address (for example 01 10 ($1001) for unexpanded VIC BASIC). BASIC includes tokens, link addresses, and line numbers in coded form; though it looks rather strange, the messages are legible. ML, however, generally needs disassembly since it appears as a collection of seemingly random characters.

Relative files are stored like sequential files, with the addition of side sectors, which are largely a list of track/sector combinations allotted to the relative file and noted in BAM as allocated.

This may appear complex at first. However, DISPLAY T&S will allow you to explore, and soon the system concepts will fall into place.

The Disk Directory

Both the entire disk directory track and the directory program can be read from BASIC, and this section explores those and other aspects of the directory. The information here will help you examine or modify disk programs, files, or directory entries by writing directly onto the disk.

What Is the Directory?

LOAD"$",8 doesn't load a conventional file. Instead, it processes the directory track, taking the diskette name, ID and DOS version from sector 1, and taking file type, filename, and file length from the directory entries in the sequence they are recorded. Because of this processing, diskettes with many files are slower than fairly empty diskettes. It is not possible to write to a file called $.

The number of blocks free is calculated from the individual directory entries. If file storage has gone awry, the computed figure may include files which don't appear in the directory; in such a case, validation is desirable. The blocks-free figure sometimes differs from the total calculable from the BAM entries.

Extending the Simple Directory

The $ directory has its own pattern-matching rules.

LOAD "$:VIC*",8 lists all programs and files beginning with VIC.
LOAD "$:??ML*",8 lists all programs and files with ML at third and fourth positions.
LOAD "$:*=S",8 lists only sequential files.
LOAD "$:MUS*=P",8 lists only programs beginning MUS.
LOAD "$:NAME" lists only NAME's entry.

Reading the Directory Within BASIC

The directory can be read from within a BASIC program without overwriting the program by using OPEN 1,8,0,"$". The zero channel is essential. GET #1 then fetches two bytes (the LOAD address), then four bytes (link pointers and line numbers) followed by a directory line and terminated by a null byte, and so on until a link pointer of 0 is found.

Program 15-8 shows how this works:

Program 15-8. Reading the Directory

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 OPEN 1,8,0,"$":REM OPEN DIRECTORY AS FILE
                                        :rem 209
20 GET#1,X$,X$:REM REJECT LOAD ADDRESS  :rem 231
30 GET #1,X$,X$,X$,X$:REM REJECT LINK POINTER & LI
   NE NUMBER                            :rem 120
40 IF ST THEN CLOSE 1:END:REM STOP AT END OF FILE
                                          :rem 6
50 GET #1,X$:IF X$="" THEN PRINT:GOTO30:REM NEW LI
   NE WHEN NULL                          :rem 54
60 IF X$=CHR$(34) THEN Q=NOTQ:REM SET QUOTES FLAG
                                        :rem 107
70 IF Q THEN PRINT X$;:REM PRINT FILENAME :rem 166
80 GOTO 50                                :rem 6
```

Sorted Directory

Program 15-9 prints a directory in the usual format, except that the names are sorted alphabetically. That makes it particularly useful if you have lots of programs. It can be modified for use with a printer and can process any number of disks, one after another.

Program 15-9. Sorted Directory

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
Ø DATA 32,115,Ø,133,97,169,128,133,98,32,115,Ø,24Ø
  ,7,9,128,133,98,32,115                       :rem 213
1 DATA Ø,165,47,133,99,165,48,133,1ØØ,16Ø,Ø,165,97
  ,2Ø9,99,2Ø8,7,2ØØ,165,98                      :rem 79
2 DATA 2Ø9,99,24Ø,2Ø,24,16Ø,2,177,99,1Ø1,99,72,2ØØ
  ,177,99,1Ø1,1ØØ,133                           :rem 71
3 DATA 1ØØ,1Ø4,133,99,144,221,16Ø,5,177,99,133,1Ø2
  ,2ØØ,177,99,133,1Ø1,2Ø8                        :rem 3
4 DATA 2,198,1Ø2,198,1Ø1,24,165,99,1Ø5,7,133,99,16
  5,1ØØ,1Ø5,Ø,133,1ØØ,165,1Ø1                  :rem 192
5 DATA 2Ø8,2,198,1Ø2,198,1Ø1,2Ø8,4,165,1Ø2,24Ø,18,
  133,1Ø5,162,Ø,134,1Ø3,134                     :rem 82
6 DATA 1Ø4,165,99,133,1Ø6,165,1ØØ,133,1Ø7,24Ø,224,
  24Ø,114,24,165,1Ø6,1Ø5                       :rem 198
7 DATA 3,133,1Ø6,165,1Ø7,1Ø5,Ø,133,1Ø7,23Ø,1Ø3,2Ø8
  ,2,23Ø,1Ø4,16Ø,2,177,1Ø6                      :rem 17
8 DATA 153,1Ø9,Ø,136,16,248,16Ø,5,177,1Ø6,153,1Ø9,
  Ø,136,192,2,2Ø8,246,17Ø                        :rem 7
9 DATA 56,229,1Ø9,144,2,166,1Ø9,16Ø,{2 SPACES}5,23
  2,2ØØ,2Ø2,2Ø8,8,165,112,197,1Ø9              :rem 169
1Ø DATA 144,1Ø,176,34,177,113,2Ø9,11Ø,24Ø,238,16,2
   6,16Ø,2,185,112,Ø,145                       :rem 142
11 DATA 1Ø6,136,16,248,16Ø,5,185,1Ø6,Ø,145,1Ø6,136
   ,192,2,2Ø8,246,169,Ø,133                     :rem 49
12 DATA 1Ø5,165,1Ø1,197,1Ø3,2Ø8,152,165,1Ø2,197,1Ø
   4,2Ø8,146,165,1Ø5,24Ø,138,96                  :rem 1
15 REM *** SORT DIRECTORY *** (SEE LINE 4ØØØØ FOR
   {SPACE}OUTPUT){2 SPACES}***                  :rem 227
2Ø POKE 56, PEEK(56)-1: CLR                     :rem 13Ø
3Ø T = PEEK(55) + 256*PEEK(56)                  :rem 167
1ØØ FOR J=T TO T+242: READ X: POKE J,X: NEXT
                                                :rem 1Ø7
1ØØØ PRINT "INSERT DISK; PRESS{5 SPACES}RETURN"
                                                 :rem 58
1ØØ2 GET X$: IF ASC(X$+CHR$(Ø))<>13 GOTO 1ØØ2
                                                 :rem 19
1ØØ4 OPEN 15,8,15,"IØ": OPEN 1,8,Ø,"$Ø"          :rem 93
1ØØ6 PRINT "OK"                                  :rem 5Ø
1ØØ8 N=2: GOSUB 1ØØØØ                            :rem 49
1Ø1Ø N=32: GOSUB 1ØØØØ: IF ST=Ø THEN D=D+1: GOTO 1
     Ø1Ø                                        :rem 191
1Ø12 CLOSE 1: DIM D$(D)                         :rem 124
1Ø14 T = PEEK(55) + 256*PEEK(56)                 :rem 1Ø
11ØØ OPEN 1,8,Ø,"$Ø"                            :rem 169
111Ø N=6: GOSUB 1ØØØØ                            :rem 47
112Ø FOR J=1 TO 25: GET#1,X$: D$(Ø)=D$(Ø)+X$: NEXT
                                                :rem 236
```

```
2000 N=3: GOSUB 10000: K=K+1: GET#1,N1$: GET#1,N2$
     : IF ST>0 GOTO 20000                         :rem 24
2010 D$(K) = STR$(ASC(N1$+CHR$(0)) + 256*ASC((N2$)
     +CHR$(0))) + " "                             :rem 21
2020 FOR J=1 TO 27: GET#1,X$                     :rem 133
2030 D$(K)=D$(K)+X$: NEXT                         :rem 42
2040 GOTO 2000                                   :rem 193
10000 FOR J=1 TO N: GET#1,X$: NEXT: RETURN :rem 42
20000 CLOSE 1: CLOSE 15                          :rem 175
30000 SYST:D                                     :rem 196
40000 OPEN 4,3: REM OR OPEN 4,4 TO DISPLAY TO PRIN
      TER                                        :rem 190
40010 FOR J=0 TO K-1: PRINT#4,"{10 SPACES}" D$(J):
       NEXT                                      :rem 233
40020 FOR J=1 TO 10: PRINT#4: NEXT                :rem 48
40030 CLOSE 4                                    :rem 161
40040 CLR: GOTO 1000                              :rem 13
```

Counting Blocks Free Within BASIC

Program 15-10 prints the number of blocks free, as reported by the directory.

Program 15-10. Number of Blocks Free

```
10 OPEN 1,8,0,"$:U=U"
20 FOR J=1 TO 35: GET #1,X$:NEXT
30 GET #1,Y$:CLOSE1
40 BF=256*ASC(Y$+CHR$(0))+ASC(X$+CHR$(0))
50 PRINTBF"BLOCKS FREE"
```

Reading BAM and the Directory Entries

The command OPEN 2,8,2,"$" (channel is nonzero) allows the BAM track and directory entries to be read directly. In other words, the whole of track 18 is read as though it were a file, and 254 characters (not including the track and sector numbers) from each block can be read with GET#. This is a convenient way to look at the directory's internal information.

Program 15-11. Reading BAM

```
10 OPEN15,8,15,"I"
20 OPEN 2,8,2,"$"
30 GET #2,X$,X$
40 FOR J=1 TO 35:GET#2,A$,B$,C$,D$
50 PRINT J;ASC(A$+CHR$(0));ASC(B$+CHR$(0));ASC(C$+
   CHR$(0));ASC(D$+CHR$(0))
60 NEXT:CLOSE 2:CLOSE15
```

Program 15-11 prints all 35 tracks of BAM information, arranged in sets of four, preceded by the track number. For example, 35 17 255 255 1 means that track 35 has 17 free sectors, and all bits 0–16 are on. The number of free blocks can be calculated from BAM; this number is usually the same as the directory's blocks-free figure.

Knowing that, you can write a directory to use information from the directory entries, for example, the first track and sector. Program 15-12 reads the directory track and reports the load address of every PRG type file; this is often helpful if you're trying to remember whether a program is BASIC or ML, or where a memory dump belongs in RAM.

Program 15-12. Reading the Directory Track

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
Ø DIM X$(3Ø)                                   :rem 58
1Ø OPEN 15,8,15,"I"                           :rem 168
2Ø OPEN 3,8,3,"#"                              :rem 29
3Ø OPEN 2,8,2,"$"                              :rem 29
4Ø FOR J=1 TO 254: GET#2,X$: NEXT             :rem 2Ø9
1ØØ FOR J=1 TO 8                               :rem 11
11Ø FOR K=1 TO 3Ø: GET#2,X$(K): NEXT          :rem 1ØØ
12Ø IF X$(1)<>CHR$(13Ø) GOTO 2ØØ               :rem 75
13Ø FOR K=4 TO 19: PRINT X$(K);: NEXT         :rem 215
14Ø PRINT#15,"U1:";3;Ø;ASC(X$(2)+CHR$(Ø));ASC(X$(3
    )+CHR$(Ø))                                :rem 211
15Ø GET#3,X$,X$,L$,H$                          :rem 8Ø
16Ø PRINT ASC(L$+CHR$(Ø)) + 256*ASC(H$+CHR$(Ø))"
    {LEFT}"                                   :rem 125
2ØØ IF J<8 THEN GET#2,X$,X$                   :rem 147
21Ø NEXT J                                     :rem 28
22Ø IF ST=Ø GOTO 1ØØ                            :rem 1
3ØØ CLOSE 2: CLOSE 3:CLOSE 15                  :rem 52
```

Line 40 skips the BAM sector, and line 100 loops through each sector in track 18. That is necessary because, although most entries have 32 bytes, the first in each block has only 30. Line 120 tests for PRG type. If this is found, its name is printed (line 130) and its track and sector pointers are used to read the block holding the start of the program. The command U1 is explained in the next section. Line 150 rejects the track and sector links but reads the low and high bytes of the start address.

Direct Access Commands

Direct access commands are roughly 20 commands that give the VIC some direct control of the disk drive. There are three types of direct access commands: those that read or write on individual diskette sectors, those that read disk drive memory or store programs in disk's RAM, and those that jump to and run programs within the disk drive memory (either in RAM or ROM). Most users need not bother with direct access, except on rare occasions, since normal disk commands can do almost as much and do it more easily. Moreover, there may be obscure bugs in these little-used commands.

The most common uses of these commands are in programs like DISPLAY T&S and VIEW BAM that rely on reading full 256-byte sectors. Disassembly of disk ROM uses a memory-read command. Generally, the write commands (apart from sector write) require some knowledge of the disk ROM; Commodore doesn't supply this, so the commands are largely unusable. In any case, disk RAM is limited.

It is risky to use individual sectors to store data (unless they are linked in a USR file) because validating the disk deallocates them in BAM and leaves them at risk of being overwritten.

The U Commands

These commands, summarized in Table 15-7, work via channel 15. For example, OPEN 15,8,15"UJ" resets the drive by turning off the light, setting the device number to 8, and generally behaving as though the disk were just turned on. U1 and U2 are versions of B-R and B-W; they operate correctly on entire sectors, including track and sector numbers of links at the start. Thus, you should generally use U1 and U2 and not B-R and B-W.

Table 15-7. U Commands

Command	Function
U1 or UA	Block Read
U2 or UB	Block Write
U3 or UC	Jump to $0500
U4 or UD	Jump to $0503
U5 or UE	Jump to $0506
U6 or UF	Jump to $0509
U7 or UG	Jump to $050C
U8 or UH	Jump to $050F
U9 or UI	Jump to ($FFFA)
UI−	Set 1541 for VIC
UI+	Set 1541 for 64
U: or UJ	Jump to ($FFFC)

Block Commands

Block read and block write (unlike all other commands) need an extra channel in which to store their data. OPEN 1,8,2,"#" opens a buffer, which BASIC refers to by its channel number (2) and file number (1). An alternative system is typically OPEN 1,8,2,"#3" where, if the channel isn't available, error 70 (NO CHANNEL) is returned. You can use this to experiment with channels.

For this discussion, assume OPEN 15,8,15 has been entered. Remember: If you are writing data, be sure to close these files so that the final buffer is written. The syntax for block read is PRINT#15,"U1"; channel;0;track;sector.

Program 15-13 is an example of how block read works. It follows a chain of sectors; try inputting track 18, sector 0 at the start. Note the use of *two* files, the command channel to load sectors in line 40, and the file to input characters in line 50. The program ends when a sector has a link set to track 0.

Program 15-13. Using Block Read

```
10 OPEN 1,8,2,"#":OPEN 15,8,15
20 INPUT"STARTING T&S";T,S
30 PRINT"TRACK"T", SECTOR"S
40 PRINT#15,"U1";2;0;T;S
50 GET#1,T$,S$:IF T$="" THEN CLOSE 1:CLOSE 15:END
60 T=ASC(T$):S=ASC(S$+CHR$(0)):GOTO 30
```

Program 15-14 demonstrates block write. It reads, alters, and writes back the first directory entry block, on track 18, sector 1. Note the use of block pointer, or B-P, in line 30, which is exactly analogous to the P parameter used with relative files.

Program 15-14. Using Block Write

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 OPEN 1,8,2,"#":OPEN 15,8,15                    :rem 225
20 PRINT#15,"U1";2;0;18;1:REM READ TRACK 18, SECTO
   R 1                                            :rem 155
30 PRINT#15,"B-P";2;2:REM ET POINTER TO POSITION 2
                                                  :rem 164
40 PRINT#1,CHR$(130+64);:REM SET PRG FILE + BIT 6
   {SPACE}ON                                       :rem 23
50 PRINT#15,"U2";2;0;18;1:REM WRITE TRACK 18,SECTO
   R 1                                             :rem 14
60 CLOSE 1:CLOSE 15:END                            :rem 52
```

This program assumes the directory has a PRG file first; by setting bit 6 to 1, the file is locked and cannot be scratched. It appears as file type PRG<. Making line 30 PRINT#15,"B-P";2;34 selects the second file in the directory, and so on, adding 32 to the second parameter for each subsequent file. If line 20 is omitted, garbage in the buffer gets written to the directory and corrupts it.

Another example is a diskette test program. DATA statements hold the highest sector numbers (from 20 to 16) for all 35 tracks; a loop (FOR T=1 TO 35:READ MS: FOR S=0 TO MS: write 255 character string and return: NEXT S: NEXT T) writes the same data to every sector. A similar loop reads each sector back to check.

Block Execute

Block execute, or B-E, has syntax OPEN 15,8,15,"B-E"; CHANNEL; 0; TRACK; SECTOR, exactly like the two previous commands. It loads the requested sector into disk memory, then jumps to the start of the same buffer, thus executing the ML program. RTS or the equivalent returns to BASIC. This could be used as the basis of a diskette copy protection device. Obviously, ML knowledge is necessary.

Memory Commands

Like the U commands, each of the following commands acts on disk memory rather than on sectors.

B-A (Block Allocate). Block allocate sets a bit in BAM low, to show that a sector is in use. Bit value 1 means it's free. Use the following form:

```
1000 PRINT#15,"B-A";0;T;S
1010 INPUT#15,E,E$,ET,ES
1020 IF E<>65 THEN END :REM T,S OK
1030 T=ET:S=ES:IF T=18 THEN T=19
1040 GOTO 1000
```

If block allocate fails (that is, if T and S in line 1000 are already used), error 65 NO BLOCK causes the program to calculate the next block, which is returned in channel 15. In this way, BAM can accurately reflect blocks written to disk by U2.

B-F (Block Free). The block free command sets a bit in BAM high, corresponding to one sector. The syntax is identical to that for B-A. Obviously, the input message isn't needed.

B-P (Block Pointer). Block pointer, as you've seen on U2, sets the point within a sector where read or write will start. Its syntax is PRINT#15,"B-P"; CHANNEL; POSITION 1—255. For example, PRINT#15,"B-P; 2; 32*F-31, where F is 1 through 8 with the directory entries in track 18, can be used to read (or write to) any of the eight file entries in any of the sectors.

M-E (Memory Execute). The memory execute command jumps to ML in disk, exactly like B-E, except that no sector is loaded and the starting address can be anywhere. Its syntax is PRINT #15,"M-E"; CHR$(low byte); CHR$(high byte). The ML can be ROM or written (with M-W) by the programmer.

M-R (Memory Read). This command sends an address to disk, and returns the value at that location along channel 15. Its syntax is PRINT#15,"M-R"; CHR$(low byte); CHR$(high byte): GET#15,M$.

To disassemble disk ROM, use a BASIC disassembler and add the following subroutine, replacing X=PEEK(P) in the disassembler.

```
10000 PRINT#15,"M-R";CHR$(P-256*INT(P/256));CHR$(P/256)
10010 GET#15,X$:X=ASC(X$+CHR$(0)):RETURN
```

M-W (Memory Write). Memory write puts data into disk RAM or interface chips. Each M-W command can write 35 bytes at most. The syntax is PRINT#15,"M-W"; CHR$(low byte) CHR$(high byte) CHR$(length) X$, where X$ is a string of not more than 35 CHR$ bytes and the other parameters are the starting address in RAM and the number of bytes.

Machine Language Programming with Disks

LOAD and SAVE

BLOCK LOAD and SAVE are discussed in Chapter 6. These work from within a program without disturbing its sequence of operations.

A loader, in the sense of LOAD then RUN, is illustrated by the autorunning program in the section on program files. It uses Kernal subroutines, as shown below:

```
LDA  #1       ;FILE NUMBER
LDX  #8       ;DEVICE NUMBER
LDY  #0       ;SECONDARY ADDRESS
JSR  $FFBA    ;SETLFS
LDA  #LENGTH  ;NAME LENGTH
LDX  #LOW     ;START OF NAME
```

```
LDY  #HIGH
JSR  $FFBD      ;SETNAM
LDA  #0
STA  $0A        ;LOAD/VERIFY FLAG 0
JSR  $FFD5      ;LOAD
JMP  START
```

This is typically standard. Note that a name is necessary with disks, even if it's only "*"; the autorunning program POKEs it in just after the ML.

SAVE is similar, except that JSR $FFD8 is SAVE, and the start and end addresses must be specified. The X and Y registers hold the low and high bytes of the final address+1. The accumulator holds the zero page address of the start address. In addition, the setup for the Kernal routine SETLFS is slightly different. The parameters for SETLFS are summarized in Table 15-8.

Table 15-8. SETLFS Summary

LOAD "NAME",8
A = 0
X = 8
Y = 0
	LOAD "NAME",8,1
	A = 1
	X = 8
	Y = 0

SAVE "NAME",8
A = 0
X = 8
Y = 1
	SAVE "NAME",8,1
	Secondary Address Irrelevant

File Handling

OPEN and CLOSE can be done in ML, though it's often easier to OPEN files in BASIC and save the hassle of setting up a name or command string in RAM.

As an example, consider the process of copying sequential or program files, in order to change a program's LOAD address. That can be done in BASIC with OPEN 1,8,2 "ORIGINAL,P,R" and OPEN 2,8,3,"NEW,P,W" followed by GET#1,X$:PRINT#2,X$; with any necessary alterations. However, the ML equivalent of GET#1 and PRINT#1 is as follows:

```
LOOP LDX  #1
     JSR  $FFC6  ;OPEN FILE 1 FOR INPUT
     JSR  $FFCF  ;INPUT A BYTE (LIKE GET#)
     PHA         ;STORE IT
     LDY  $90    ;LOAD ST
     LDX  #2
     JSR  $FFC9  ;OPEN FILE 2 FOR OUTPUT
     PLA         ;RECOVER BYTE
     JSR  $FFD2  ;OUTPUT IT (LIKE PRINT#)
```

```
CPY  #0
BEQ  LOOP   ;CONTINUE IF ST IS 0
LDA  #1
JSR  $FFC3  ;CLOSE 1
LDA  #2
JSR  $FFC3  ;CLOSE 2
JSR  $FFCC  ;BACK TO NORMAL
RTS         ;RETURN
```

The demonstration uses CHKIN and CHKOUT (from the Kernal) to signal file numbers, and CHRIN and CHROUT to get and print a character. CLOSE is easy to use, as the example shows. CLRCHN ($FFCC) returns things to normal operation.

Program 15-15 gives another, shorter example. It is POKEd from BASIC; try it if you're inexperienced in ML. It displays 256 bytes from an open file #1 on an unexpanded VIC-20. Try OPEN 1,8,2,"*,P,R": SYS 828 which will open the first file on disk (assumed to be a program) and display 256 bytes in black. Further SYS 828 commands read further; the end is marked by RETURN characters, appearing as m. Enter CLOSE 1 to finish. You can also use this technique to examine sequential files, with OPEN 1,8,2,"*filename*": SYS 828.

Program 15-15. ML File Reader

```
10 FOR J=828 TO 851: READ X: POKE J,X: NEXT
100 DATA 162,1,32,198,255,160,0,32,207,255
110 DATA 153,0,30,169,0,153,0,150,200,208
120 DATA 242,76,204,255
```

OPEN and CLOSE in ML

OPEN uses SETLFS to set the parameters typified by 1, 8, and 2 in OPEN 1,8,2"*filename*". Use the following:

LDA # File Number (e.g., #1)
LDX #8
LDY # Channel number (e.g., #2)

The Kernal SETNAM routine at $FFBD uses the name, or command string, pointers, and length exactly like LOAD or SAVE. The Kernal OPEN routine is at $FFC0.

The Kernal CLOSE routine is easier. The file number is stored in the accumulator, then JSR $FFC3 closes the file.

Channel 15 and ML

OPEN 15,8,15 is just a special case of OPEN. Messages from channel 15 consist of ASCII numerals and the message separated by commas and terminated by return. Thus, message 0 is this string:

```
48 48 44 32 79 75 44 48 48 44 48 48 13
 0  0  ,     O  K  ,  0  0  ,  0  0 RETURN
```

Thus, to check channel 15 from disk, open file 15, input two bytes, and check that each is 48. If not, the message can be printed by inputting further characters and outputting them, using $FFD2, in a loop until RETURN is received.

The following routine performs the equivalent of OPEN 15,8,15: INPUT#15,E,E$,T,S: PRINT E,E$,T,S: CLOSE 15:

```
                    ;OPEN 15,8,15
     LDA #$0F
     LDX #8
     LDY #$0F
     JSR  $FFBA  ;SET 15,8,15
     LDA #0      ;SET LENGTH OF NAME=0
     JSR  $FFBD  ;POINTERS IRRELEVANT
     JSR  $FFC0  ;OPEN 15,8,15
     LDX #$0F
     JSR  $FFC6  ;OPEN 15 FOR INPUT
                 ;INPUT#15
LOOP JSR  $FFCF  ;GET A BYTE
     CMP #$0D    ;EXIT IF RETURN
     BEQ EXIT
     JSR  $F282  ;PRINT TO SCREEN (OR FFD2)
     BNE LOOP
                 ;CLOSE 15
EXIT LDA #$0F
     JSR  $FFC3  ;CLOSE 15
     JSR  $FFCC  ;FILES NORMAL
     RTS
```

This routine can be used from BASIC or ML. In ML programming, as well as in BASIC, it is often useful to keep file 15 open while the program runs. Use the segment marked OPEN 15,8,15 to open. To test the error number, input two bytes using the portion marked INPUT#15 and check that both equal $30 (decimal 48).

It's almost as easy to send a command to channel 15. Simply open the channel for output (with $FFC9) and send bytes, finishing with RETURN. CLOSE will not work immediately after this; use JSR $FFCC or make the disk unlisten. For example, LDA #$49, JSR $FFD2, LDA #$0D, JSR $FFD2, JSR $FFCC initializes the disk, if channel 15 is OPEN for output, by sending I then RETURN (exactly like PRINT#15,"I").

Chapter 16

The Games Port

Chapter 16

The Games Port

This chapter explains the programming and use of devices that connect to the games port, notably joysticks, paddles, and light pens. Useful ML programs are included to help you write even more efficient programs.

Joysticks and the VIC-20

The VIC-20 has a single games or controller port, with the almost universal D-connector. Except perhaps for the tape drive, joysticks are by far the most common VIC accessory.

It is possible, and easy, to attach a second joystick, but little software can be expected to exist for such a nonstandard device. An additional stick has to be fitted to a different port, typically the user port, since there's no way to distinguish two sticks connected to the same port.

The principle of joystick operation is simple. The stick itself is grounded; moving it grounds sensors positioned up, down, left, or right. The fire button grounds another wire. Thus, the cable running to the VIC contains six wires, one of them grounded and the other five normally high but capable of being grounded (set low) by the controller. All the VIC has to do is to test for one or more wires being low.

Most joysticks are designed so that intermediate positions—for example, northeast or southwest—ground two wires at once. Therefore, the VIC may detect as many as three wires being low simultaneously (two "direction" wires and the fire button wire). Some combinations aren't normally possible, of course, like north with south.

Internally, the most common arrangement is a ring connected to ground with pressure-sensitive dimpled-metal switches which make contact with the ring when the stick or button moves. Heavy-duty models use other arrangements—for example, microswitches. Some models have two fire buttons, so they can be used in either hand, and converting some types for left-hand operation isn't hard.

The life of joysticks tends to be short, often because the cable contains the thinnest possible strands of wire which are prone to break just inside the casing. To test a joystick, use one of the programs in this chapter to insure that all eight directions can be found easily. Try the joystick with the program, and see if the response seems satisfactory. Remember, though, that problems may stem from nothing more exotic than a loose plug.

VIC programs generally are limited to one joystick. Two users or players can be accommodated only if they take turns (or if one uses the keyboard), unless an extra joystick is fitted. Paddles, which are sold in pairs, offer another possibility.

Programming Information

When programming joysticks, three locations need to be considered. They are shown in Table 16-1.

Table 16-1. Relevant Locations for Joystick Programming

	Bit#: 7	6	5	4	3	2	1	0
$911F (37151)			Btn	W	S	N		
$9120 (37152)	E							
$9122 (37154)								
	Decimal Value: 128	64	32	16	8	4	2	1

Reading $911F (or $9111, which is identical) requires only a simple PEEK. Enter FOR J=1 TO 999: PRINT PEEK (37151): NEXT to watch the effect of the joystick. Since 32+16+8+4=60, and since the bits are inverted, 60−(PEEK (37151) AND 60) looks more meaningful. Note that east has no effect.

To read east, note that the keyboard shares the port with the joystick. Bit 7 has to be reconfigured for input, typically by POKE 37154,127. After PEEKing port B, the keyboard can be returned to normal. The following line prints 128 when the stick is east, northeast, or southeast: POKE 37154,127: PRINT 128−(PEEK (37152) AND 128): POKE 37154,255.

BASIC is really too slow for such a test and will make the joystick seem unresponsive. In addition, the keyboard may do odd things. If location 37154 is not reset to 255, for example, only odd numbers can be typed. In addition, moving the joystick right while pressing 2 will simulate the STOP key. RUN/STOP–RESTORE returns operation to normal.

BASIC Programs

Joysticks are easily incorporated into your programs. When only the four cardinal points (N, E, S, W) are important, a routine like that shown in Program 16-1 will work well. However, where all eight directions are important, the routine shown in Program 16-2 will be more useful. As you see, ON-GOTO can help cut out a lot of IF-THEN testing.

Program 16-1. Testing for Four Directions

```
1000 P=PEEK(37151): IF (P AND 32)=0 THEN GOTO FIRE
     BUTTON ROUTINE
1010 IF (P AND 16)=0 THEN GOTO WEST
1020 IF (P AND 8)=0 THEN GOTO SOUTH
1030 IF (P AND 4)=0 THEN GOTO NORTH
1040 POKE 37154,127: P=PEEK(37152):POKE 37154,255
1050 IF (P AND 128)=0 THEN GOTO EAST
```

Program 16-2. Testing for Eight Directions

```
1000 P=PEEK(37151): IF (P AND 32)=0 THEN GOTO FIRE
     BUTTON ROUTINE
1010 P= P AND 28
```

```
1020 POKE 37154,127:PP=PEEK(37152) AND 128:POKE 37
     154,255
1030 P=P/2+PP/128: REM W,S,N,E NOW IN BITS 3,2,1,0
1040 ON P GOTO ,,SW,,NW,,W,,,SE,S,NE,N,E,CENTER
```

As a final BASIC example, Program 16-3 is designed to work with a graphics program using screen POKEs. North returns −22, East 1, Southeast 23, and so on. Adding one of these values to the screen position gives the new POKE address.

Program 16-3. Using the Joystick to Modify Screen POKEs

```
10000 POKE 37154,127: P=-((PEEK(37152) AND 128)=0)
      : POKE 37154,255
10010 PP=PEEK(37151)
10020 P=P + ((PP AND 16)=0) - 22*((PP AND 8)=0) +
      22*((PP AND 4)=0)
10030 RETURN
```

Machine Language Joystick Programs

ML is much faster than BASIC. Programs 16-4 and 16-5 both use ML but are different in approach.

Program 16-4 uses a SYS call to read the joystick. Like RJOY with the *Super Expander*, the result is stored and used by BASIC. It puts the result into $90, where ST reads it. This is simpler than using a PEEK. IF (ST AND 16) tests for the fire button; ON-GOTO separates the directions, which are 1, 2, 4, and 8 for E, N, S, and W.

Program 16-4. Joystick Reading Via ST

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
20 REM **{2 SPACES}SYS 828 READS JOYSTICK INTO ST
   {2 SPACES}**                            :rem 79
30 REM **{2 SPACES}1=E, 2=N, 4=S, 8=W, 16=BUTTON
   {3 SPACES}**                            :rem 31
40 REM **{2 SPACES}ADD FOR COMBINATIONS OF THESE
   {3 SPACES}**                            :rem 52
50 FOR J=828 TO 857: READ X: POKE J,X: NEXT:rem 26
100 DATA 173,31,145,73,255,41,60,162,127,120
                                          :rem 103
110 DATA 142,34,145,172,32,145,48,2,9,2,162:rem 55
120 DATA 255,142,34,145,88,74,133,144,96  :rem 187
```

As it stands, ST takes the current value, returning to zero whenever the stick is released. Sometimes it's easier to retain a value until it's changed to a new one; you can do this by replacing 133, 144, 96 in line 120 by 201, 0, 240, 2, 133, 144, 96, and changing the upper limit in line 50 to 861.

The second routine, Program 16-5, is initialized by SYS 862, which alters the interrupt vector so that ST is continuously reading the stick. No further SYS call is required; just test ST whenever you need to know the joystick reading. Add 60 PRINT ST: GOTO 60 to watch this in action. Again, if you wish to retain values in

ST (perhaps for a maze game where a direction is held unless altered) replace 234, 234, 234, 234 in line 120 with 201, 0, 240, 2.

Program 16-5. Continuous Joystick Reading Via ST

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
20 REM ** SYS 862 STARTS INTERRUPT ROUTINE **
                                              :rem 97
30 REM ** 1=E 2=N 4=S 8=W 16=BUTTON: IN ST **
                                              :rem 231
40 REM **{2 SPACES}ADD FOR COMBINATIONS OF THESE
   {3 SPACES}**                               :rem 52
50 FOR J=828 TO 874: READ X: POKE J,X: NEXT:rem 25
100 DATA 173,31,145,73,255,41,60,162,127  :rem 168
110 DATA 142,34,145,172,32,145,48,2,9,2,162:rem 55
120 DATA 255,142,34,145,74,234,234,234,234 :rem 16
130 DATA 133,144,76,191,234,120,169,3,141 :rem 219
140 DATA 21,3,169,60,141,20,3,88,96       :rem 181
```

Both routines are designed to leave the keyboard working correctly, so keyboard control is no problem with either GET or PEEK(197). In pure ML programs, parameters need not be passed back to BASIC; instead you can set the carry flag only if the button is pressed and use the accumulator to indicate direction. You can also add optional keyboard control by looking at location $C5 for keypresses.

Hardware Notes

The controller socket has five pins on top and four underneath. These are numbered 1 to 5 and 6 to 9 in conventional left-to-right order. Joysticks use only pins 1 (up), 2 (down), 3 (left), 4 (right), 6 (fire button), and 8 (ground).

Adding a second joystick is quite easy. A 24-pin edge connector fitting the user port has pins A, B, C, D, E, F, H, J, K, L, M, and N on the underside, reading left to right while looking at the VIC. Pins A and N are ground; pins C through L are the user port, wired to $9110 (37136).

If you PEEK this location, you can see that it is normally 255. However, grounding any pin C through L turns off a bit. Therefore, connecting the six wires from a joystick (ground to pin A or N and the other five to pins C through L) gives a simple PEEK-readable joystick connector which operates just like the game port joystick connector. If N, S, W, fire button, and E are connected to bits 2, 3, 4, 5, and 7 of the port (pins E, F, H, J, and L), respectively, then ordinary BASIC PEEKs to read the joystick can be directly converted, dropping the redundant POKEs of 127 and 255.

Paddles and the VIC-20

Game paddles are far less popular than joysticks, mainly because the simple up-down-left-right joystick motions of a joystick are generally easier to use than the rotational motion of paddles.

Commodore's paddle system consists of two separate hand-held paddles, each with a knob and fire button, which plug together into one games port. Two people can therefore use paddles to control the VIC—for example, in games like *Omega*

Race. The VIC reads the paddle position as a number value. Counterclockwise motion increases the value read by VIC, while clockwise rotation causes the number to decrease. It may be worth labeling the paddles; the X paddle is read by VIC chip location $9008 (36872), and the Y paddle is read by $9009 (36873). The VIC chip's registers are eight bits wide, so there's a potential resolution of 1 in 256, a useful range.

Paddles are analog (not digital) devices, so, unlike joysticks, there could be more compatibility problems between makes (and even between different models of the same make) because of differences in resistance between the units. Here is how they work. The games port has a 5-volt power supply at pin 7 (remember, looking at the VIC, pins are 1 to 5, then 6 to 9, in the games port). This voltage goes through the paddles; paddle output is connected to pin 5 (for one paddle) and to pin 9 (for the other).

These pins communicate directly with the VIC chip registers for Y and X, respectively. Each register performs an analog-to-digital conversion of voltage into a number ranging from 0 to 255. The higher the voltage, the smaller the number, so with nothing plugged in, PEEKing these registers shows a value of 255. In other words, the registers measure the resistance of each paddle. The paddles contain a simple potentiometer (a variable resistor) which is adjusted by turning the paddle knob.

Paddles also have an additional resistor, so the 5-volt line isn't fed directly into the VIC. It's simple to use the same principle with other resistances or potentiometers, but for safety keep a minimum resistance of several hundred ohms in the circuit. VIC's registers are supposed to detect changes at about 1000 ohm steps, so 255K and more will read as 255 in the VIC chip.

Why have two analog-to-digital converters? This makes sense for applications involving two-dimensional activities. For example, graphics tablets use electrical resistance to estimate the left-right and up-down coordinates. The same principle applies when using a mouse, which relies on an inverted trackball to produce two outputs.

Paddles, like joysticks, are best read with machine language. First, though, it is helpful to see how it's done in BASIC. To read paddle X, PEEK $9008 (36872) which holds 0–255. To read paddle Y, PEEK $9009 (36783). The fire button on paddle X turns off bit 4 of $911F (37151) when pressed. To read the fire button on paddle Y, bit 7 of $9122 (37154) must first be set low, so you can read bit 7 of $9120 (37152), which is turned off when the fire button is pressed.

Obviously, it is easy to read the paddle values using a line like 10 PRINT PEEK (36872); PEEK (36873): GOTO 10. However, reading the buttons is more trouble. Reading paddle buttons leads to interaction with the keyboard, just as with the joystick. If only one button is needed, use paddle X, which can be read with IF (PEEK (37151) AND 16) = 0 THEN GOTO FIRE BUTTON ROUTINE.

To read and display all the features, use Program 16-6. It shows the fire button as 0 (off) or 1 (on) and also shows the value of the paddles.

Program 16-6. Reading Paddles with BASIC

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
1 PRINT "{CLR}"                                    :rem 149
1Ø PRINT "{HOME}"                                   :rem 69
2Ø PRINT "X"; 1-(PEEK(37151) AND 16)/16; PEEK(3687
   2)"{LEFT}{2 SPACES}"                             :rem 88
3Ø PRINT "Y";                                      :rem 2ØØ
4Ø POKE 37154,127: PRINT 1 - (PEEK(37152) AND 128)
   /128;: POKE 37154,255                           :rem 243
5Ø PRINT PEEK(36873)"{LEFT}{2 SPACES}"              :rem 84
6Ø GET X$: IF X$="" GOTO 1Ø                         :rem 34
```

Program 16-6 also lets you investigate the paddles' accuracy. The values can change quite rapidly; a complete twist of a paddle takes about 1/12 second. Extreme values are generally least accurate; there is a lot of play. There's also "cross talk" between the two paddles, probably caused by the 5-volt supply being in common. In other words, when one paddle has a very low reading, the other paddle returns a lower reading even though it is untouched. If this is a problem, low values could be made out of range, so a player using them could lose a turn or whatever. Another possibility is to apply a correction, as shown in a subsequent program.

Commodore paddles don't use high-precision potentiometers, so the resolution is limited by the construction of the potentiometer. These paddles use so-called wire-wound potentiometers, in which a length of resistance wire is wrapped around a circular form and then tapped by a rotating arm, and the resistance depends on the point at which the coil of resistance wire is tapped. As a result, the resistance increases (or decreases) in a series of tiny steps, and you may find that certain values (which correspond to resistances falling between the taps) can't be read.

Another potential problem is that the inherent vagueness of analog-to-digital conversion returns occasional out-of-step series of data. One way around this is to smooth the values. An ML loop to simply read the value is not a very good approach; a better one is either to average the last few readings or to use a continuous averaging technique to smooth the readings, taking readings at each interrupt.

ML can do this much better than BASIC. The following ML routine, Program 16-7, is fairly sophisticated; it compensates for low values in the other channel and also applies smoothing (by combining 3/4 of the previous reading and 1/4 of the present reading). This introduces some damping; if it's too much, just increase the interrupt (IRQ) rate. Note that individual paddles may give slightly different results.

Program 16-7. Reading Paddles with ML

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
2 REM **{4 SPACES}AFTER SYS 828, MODIFIED VALUES A
  RE{2 SPACES}STORED IN 253 AND 254 **        :rem 86
3 REM **{4 SPACES}AND ST STORES BOTH FIREBUTTONS
  {6 SPACES}**                               :rem 155
1Ø FOR J=828 TO 929: READ X: POKE J,X: NEXT:rem 22
```

```
100 DATA 120,169,3,141,21,3,169,73,141,20,3,88,96,
    174,8,144                                  :rem 82
101 DATA 172,9,144,138,192,1,240,3,176,14,36,74,74
    ,74,74,74                                  :rem 97
102 DATA 56,109,8,144,144,2,169,255,56,101,253,106
    ,133,253                                   :rem 37
103 DATA 152,224,1,240,3,176,14,36,74,74,74,74,74,
    56,109,9                                   :rem 45
104 DATA 144,144,2,169,255,56,101,254,106,133,254,
    173,31,145                                :rem 133
105 DATA 73,255,41,16,162,127,142,34,145,172,32,14
    5,48,2,9                                   :rem 37
106 DATA 128,162,255,142,34,145,133,144,76,191,234
                                              :rem 162
```

Use this program by replacing \$9008 and \$9009 by \$FD (253) and \$FE (254). Generally, these PEEK values are similar but show less cross talk and less variability. The fire buttons are both put into ST. For consistency, bit 4 or bit 7 is set to 1 when button X or Y is pressed, so test with IF (ST AND 16)=16 or IF (ST AND−128) =−128.

Light Pens

A light pen is a pen-sized device, fitted with a cable, which plugs into the game port. Its internal electronics are more complex than those in joysticks or paddles. The tip of the pen is fitted with a small light-sensitive component, usually a phototransistor, which allows current to pass only when the sensor is exposed to light.

VIC light pens use the 5-volt and ground lines, plus a line into the VIC chip. When this line detects a drop in voltage, two registers in VIC are frozen, or latched, and remain unaltered until there's another drop. Obviously, since the TV phosphors are lit once then decay slowly, the light pen must detect (and the voltage drop must correspond to) a sudden increase in lighting.

The two registers hold the horizontal and vertical positions of the pen position, or rather the position inferred from the voltage change.

The August 1982 issue of *COMPUTE!* contains an article outlining home construction of a light pen. If you decide to build your own, remember that loose mounting can cause problems.

Using a Light Pen with the VIC

Whenever a range of alternatives is to be selected, particularly if the values are not alphanumeric, a light pen is potentially useful. Selecting alternative answers to multiple-choice problems and selecting options from a menu are obvious examples; so are board games (for example, chess) where light pen input is easier than keyboard input. Graphics design is commercially an important application, though VIC's fairly low resolution rules this out on any great scale. Numbers can be input with a fairly large numerical 0–9 "pad" on the screen. As you'll see, a light pen can be used to sketch on the screen and give some idea of the final appearance of different color combinations.

But light pens have a few disadvantages, too, generally due to the limited accuracy of the pens and the limitations of the computer. For example, the thick glass at the front of the TV tube reduces definition. In addition, unless the pen has an on/off switch, false readings can occur as the pen is moved around near the screen.

There are physical aspects too. It may be undesirable to sit right next to the screen, and tapping the screen with the pen may not be such a good idea. The cable length from VIC to the pen may be a problem, and the user may need to return to the keyboard for some operations.

Another difficulty, inherent in VIC's detection, is that some colors (black and generally red) cannot trigger the pen. That puts some limitations on color plotting.

Even so, it's generally possible to program around some of these difficulties by allowing reinput if a wrong value is accidentally read in.

Light Pen Programming

Location $9006 (36870) is the horizontal position register, and $9007 (36871) is the vertical position register. Both are PEEKed (POKEing has no effect); it's as simple as that. So 10 PRINT "{CLR}" PEEK (36870); PEEK (36871): GOTO 10 gives a legible output of both registers. As you move the pen down, one reading increases; as you move it to the right, the other reading increases.

An ordinary 22 row by 23 column screen returns a horizontal range of values from about 29 to 117, and a vertical range of roughly 24 to 116, from the edge of the display area. These figures need to be converted into values from 0 to 21 and from 0 to 22, respectively, or perhaps from 0 to 43 and from 0 to 45 if double-density plotting is being used.

Resolution is to two dots. There are 22*8=176 dots across, and 23*8=184 down. It's therefore quite easy to convert light pen readings to screen character positions. Just subtract 29 then divide by 4 to get the horizontal position, and subtract 24 and divide by 4 for vertical. These calculations are quite easy in machine language, although some light pen demonstration programs written in BASIC are painfully slow.

How reliable are the readings? The interrupt lets you read the pen 60 times a second; you can hold the pen still and watch for variations. The readings will vary within 2 or 3, so this removes any real chance of serious hi-res work. However, any pen should be accurate within normal VIC screen characters.

BASIC Programming

Program 16-8 is a subroutine that reads the light pen registers and converts them into CH%, the PEEK value of the screen location at which the pen points. As an example, if a menu has reversed numerals from 1 to 5, which only appear in one place on the screen, the subroutine can check which is selected; if none, the program can go back and retry.

Program 16-8. BASIC Light Pen Subroutine

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
3 REM ** EXAMPLE: GOSUB 10000: IF CH%=177 **
                                         :rem 203
4 REM ** TO TEST FOR REVERSE 1 **        :rem 215
10000 H%=PEEK(36870): V%=PEEK(36871)     :rem 138
10010 SC%=256*PEEK(648)                  :rem 201
10020 IF H%<29 THEN H%=29                :rem 218
10030 IF V%<24 THEN V%=24                :rem 237
10040 IF H%>117 THEN H%=117               :rem 58
10050 IF V%>116 THEN V%=116               :rem 85
10060 H%=(H%-29)/4: V%=(V%-24)/4          :rem 14
10070 CH%=PEEK(SC% + H% + 22*V%)         :rem 226
10080 RETURN                             :rem 217
```

Integer variables have been used for two reasons. First, if the main program uses normal variables, there's no possible conflict; second, the vertical reading must be converted to an integer before multiplying by 22.

The subroutine is easy to modify, if for example your pen gives slightly different readings at the display area edge. It's also easy to use it to plot characters—reversed square (160) could be useful—by POKEing the screen RAM position, CH%, with 160 or whatever and POKEing the color RAM at the same time.

Machine Language Programming

Mixed BASIC and ML is far faster than BASIC, because all the operations of checking limits, subtracting, dividing, and so on can be handled in very few ML commands.

Program 16-9 is a double-density (44 × 46) plotter that uses the ML routine from Chapter 12 to plot small squares of any color on VIC's screen. Line 700 is the SYS call to the ML routine; this value works with the unexpanded VIC, with the ML in the top of memory. If you wish, add DATA statements and POKEs to this program for a single combined program. The resolution of the resulting sketchy graphics is about the limit for even a good light pen.

Color keys 1–8 change the plotting color, function key f1 alters the background color, f3 changes the border color, f5 stops plotting, and f7 enables plotting.

Program 16-9. Double-Density Light Pen Plotter

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
8 REM * MUST HAVE CHAPTER 12'S 44-BY-46 DOUBLE- *
                                         :rem 112
9 REM * DENSITY PLOTTING ML ROUTINE IN PLACE *
                                         :rem 189
100 PRINT"{CLR}"                         :rem 245
500 H=PEEK(36870): V=PEEK(36871)         :rem 228
510 IF H<29 THEN H=29                     :rem 51
```

```
520 IF V<24 THEN V=24                              :rem 70
530 IF H>117 THEN H=117 :REM AVOIDS WRAPROUND
                                                   :rem 57
540 IF V>116 THEN V=116 :REM AND OUT-OF-RANGE ERRO
    RS                                            :rem 208
550 H=(H-29)/2 :REM DIVIDE BY 2, NOT 4, AS PLOTTIN
    G                                             :rem 211
560 V=(V-24)/2 :REM USES 4 DOTS BY 4              :rem 100
570 V=46-V :REM DOUBLE-DENSITY PLOT STARTS AT BOTT
    OM LEFT                                       :rem 247
600 GET X$                                        :rem 242
610 IF X$>"0" AND X$<"9" THEN COLOUR=ASC(X$)-49
                                                  :rem 218
620 IF X$=CHR$(133) THEN POKE 36879,(PEEK(36879) +
     16) AND 255                                   :rem 61
630 IF X$=CHR$(134) THEN POKE 36879,((PEEK(36879)
    {SPACE}AND 247) + 1) OR 8                      :rem 52
640 IF X$=CHR$(135) THEN GET Y$: IF Y$<>CHR$(136)
    {SPACE}THEN 640                               :rem 206
700 SYS 7505,H,V,1,COLOUR                         :rem 186
710 GOTO 500                                      :rem 102
```

As you've seen, it's easy to POKE characters to the screen at the light pen's position. ML subroutines can accelerate the process. First, you can use a program where a SYS call, say SYS 828, returns the H and V positions from 0 to 21 and from 0 to 22 in locations 253 and 254 and checks for (and ignores) out-of-range values. Program 16-10 can be used with SYS828: H=PEEK(253): V=PEEK(254) to speed up BASIC programs.

Program 16-10. An ML Light Pen Reader

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 FOR J=828 TO 862:READ X: POKE J,X: NEXT :rem 18
100 DATA 56,173,6,144,233,29,144,26,201,88,176,22,
    170                                            :rem 49
110 DATA 56,173,7,144,233,24,144,13,201,92,176,9,7
    4                                             :rem 205
120 DATA 74,133,254,138,74,74,133,253,96   :rem 188
```

Finally, you can combine ML with the routine given in Program 16-11 to plot standard-sized characters. For example, it will let you plot colored blobs at will on the screen (POKE 81) and then convert them to reverse spaces (POKE 160). In this way any color (even black) can be plotted, and the resulting screen appearance can be assessed.

Program 16-11. ML Plotting with a Light Pen

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 FOR J=828 TO 929: READ X: POKE J,X: NEXT:rem 22
100 DATA 56,173,6,144,233,29,144,93,201,88,176
                                                  :rem 225
```

```
110 DATA 89,170,56,173,7,144,233,24,144,80,201
                                             :rem 213
120 DATA 92,176,76,74,74,133,254,138,74,74,133,253
                                             :rem 178
130 DATA 173,172,3,240,62,165,254,10,101    :rem 158
140 DATA 254,10,10,56,229,254,10,133,251,169
                                             :rem 107
150 DATA 0,144,2,169,1,133,252,24,165,253,101
                                             :rem 146
160 DATA 251,133,251,169,0,101,252,133,252,160
                                             :rem 198
170 DATA 0,24,165,252,109,136,2,133,252,173,173
                                               :rem 1
180 DATA 3,145,251,165,252,41,3,9,148,133,252
                                             :rem 163
190 DATA 173,174,3,145,251,96               :rem 155
```

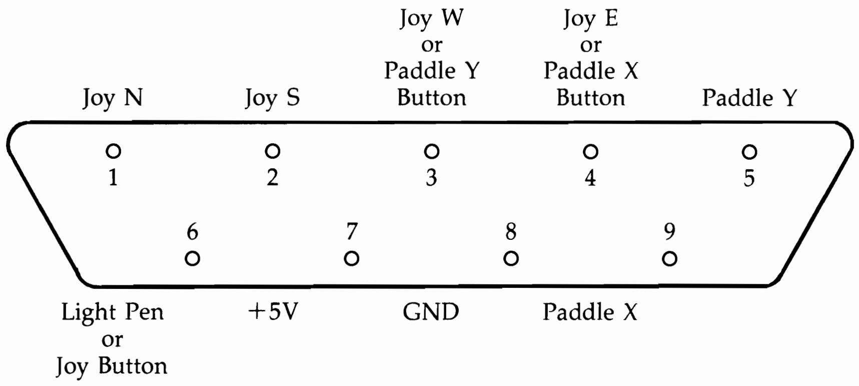
The routine uses three locations. Location 94O is on/off; POKE it with 0 to turn the pen off and with 1 to turn the pen on. Location 941 holds the character, and location 942 holds the color.

Note that the calculation part of the program is added on the end of the previous program which reads the pen. You can test the plotting portion with POKE 940,1: POKE 941,160: POKE 942,2: POKE 253,9: POKE 254,9: SYS862, which will plot a red square in position 9,9 on the screen. This works for any memory configuration. It multiplies V by 22, adds H, and adds the current screen position; then it POKEs the character, finds the color RAM position, and POKEs that. For use with the light pen, simply use SYS 828 in a loop.

The Games Port: Hardware

Figure 16-1 diagrams the nine pins of the VIC games port. The diagram is drawn as if you are looking at the VIC.

Figure 16-1. The VIC Games Port



A conventional joystick cannot be used at the same time as paddles (unless the paddle buttons aren't used) or with a light pen (unless the joystick button isn't used). However, a light pen and paddles can be used together. The user port's pins 1, 2, 4, 5, 6, and 7 duplicate pins 8, 7, 1, 2, 3, and 6 of the games port, so for some experiments an edge connector in the user port can replace the games port. However, neither potentiometer nor joystick east/paddle Y button connects to the user port.

Graphics Tablets

These analog devices are programmed like paddles. They may or may not have a button; when they do, it too is programmed like one of the paddle buttons. Generally such devices are used either to move characters on the screen or to select from a menu. A tablet can have a preprinted form laid over it, with boxes selectable by the stylus (and large enough to be distinguished without much chance of error). Programs for these devices use subroutines analogous to those used with light pens, allowing for the existence of both vertical and horizontal components.

Chapter 17

Major Peripherals

Chapter 17

Major Peripherals

This chapter covers printers, plotters, modems, and VIC's interfaces. Simple program examples are included for quick reference. VIC's serial and RS-232 interfaces are covered in the last section, which is primarily concerned with software.

Printers

Simple Commands

Commodore printers designed for the VIC plug into the serial port, the round port at the back of VIC next to the video output. At the simplest level, printers are controlled with OPEN 4,4 (which opens file number four to the printer), PRINT#4,"HELLO" (which prints a message to file four), and CLOSE 4 (which closes the file). Any number of PRINT#4 statements can be issued. PRINT statements, which send output to the screen, can also be mixed in.

The VIC has no LIST#4 statement. To LIST programs to the printer, use OPEN 4,4: CMD 4 (which directs output to file four), LIST, and PRINT#4: CLOSE 4 (which closes the file). As you'll see, PRINT#4 is needed to close the file properly. It's not necessarily important to close the file to a printer—nothing disastrous will happen if you don't—but if you leave it open some output is liable to appear on the printer rather than on the screen.

With *VICMON*, all output can be sent to a printer with OPEN 4,4: CMD 4: SYS 6*4096. Then enter M 1000 1200, for example, and output for the desired memory will be made to the printer. The commands have to be typed in blind, since they don't echo to the screen, but this isn't a big problem. Enter X, then CLOSE 4 and POKE 43,1, to return to BASIC.

Non-Commodore Printers

For the most-used applications, many non-Commodore printers use commands identical to those for CBM printers. The exceptions are printers which use the RS-232 port (that is, which plug into the user port at the back left of the VIC, usually with an RS-232 converter cartridge). Any RS-232 device has device number 2, not 4.

To use such a device, the following sequence is typical: OPEN 2,2,0,CHR$(6) (to open file number 2 with baud rate 300), then PRINT#2, "HELLO" and then CLOSE 2 (to close the file). CMD 2: LIST will list to such a printer. See the notes later in this chapter for more on RS-232 printers, which may not always work with VIC.

PRINT#

PRINT# statements are exactly similar to ordinary PRINT statements, but watch out for the distinction between the carriage return character, CHR$(13), and the linefeed character, CHR$(10), which advances the paper in the printer. CBM printers are designed to treat PRINT# followed by a semicolon as an instruction to remain on the same line. PRINT# followed by a colon or end-of-line is treated as a combined carriage return and linefeed, so PRINT# behaves just like PRINT to the screen. Not all printers have an automatic linefeed; if your non-CBM printer overprints lines on top of each other, use a file number of 128 and up (OPEN 128,4 for example) with PRINT#128,"HELLO" to cause the VIC to output the linefeed.

Control characters to the printer (to print reversed text, lowercase, and so on) are sent as special characters which the printer recognizes, typically as PRINT#4, CHR$(27) or as PRINT#4,A$ (where A$ is a string of CHR$ values). Printer programs are liable to contain a certain number of PRINT# statements which are only meaningful with reference to the printer in use. It is best to use special printer features sparingly, if at all, because any subsequent change of printer may wreck a successful program's output.

Choosing and Using Printers

A printer is simply a device to convert a stream of bytes into text. Unlike other CBM devices, non-CBM printers can often substitute for Commodore equipment. This is not likely to be worthwhile except where special print quality is not offered by Commodore, or where a user has a printer already or doesn't want to be restricted to CBM printers in future. In all these cases, some sort of interface (which may be quite expensive) will be necessary, because neither of the VIC's printer ports is standard.

Printers are mainly used for word processing applications. But the VIC, without 40-column software or hardware, cannot realistically handle such assignments. Thus VIC printers are used for program listings and for miscellaneous output, such as labels, notes, calculation printouts, and short reports.

Commodore printers for the VIC include the old 1515, the newer 1525, and the more versatile 1526. Their features are summarized in Table 17-1. All are designed to be plug-compatible with both the VIC and the 64. They are made by Seikosha; all CBM printers are made by other companies.

Table 17-1. Commodore Printers for the VIC

	1515	1525	1526
Dot Resolution of Characters	7 up×6 across	7×6	8×8
Characters per Inch	12	10	11
Paper Widths (inches)	Up to 8	Up to 10	Up to 10
True Descenders on g, j, p, q, y?	No	No	Partly
Approx. Speed, Characters per Sec.	30	30	60
Separation Between Lines	2 dots	2 dots	Programmable
Programmable Formatting of Output?	No	No	Yes
Programmable Top-of-Form Feed?	No	No	Yes
Ribbon Type	Cloth	Cloth	Carbon Film

Each of these printers has the complete range of ROM graphics, as might be expected, although none gives an impression identical to the VIC's characters. The 1515's reversed characters, for example, lack the solid underline of the screen characters and look a little ragged. In addition, lowercase letters like g are perched up in the air. The printers have built-in ROM, to process incoming commands and store graphics patterns, as well as RAM, to act as a buffer, storing data while the printer deals with it.

ROMs may be changed, if bugs come to light, without warning. There's no guarantee that some models won't differ from others. CBM's printers haven't consistently used identical commands in the past, so, for preference, stick to commands which seem firmly established. These are discussed further in the next section.

Each printer allows 80 normal-width characters to the line, but note that the 1515 used a smaller typeface, and also nonstandard 8-inch wide paper, in order to do so. It is possible to use 8½-inch paper on the 1515, by loosening the Perspex paper guide, removing the lid and the guide, and taking out the bar so only the paper holders touch the paper. However, the result is a very noisy printer.

The number of lines per page has to be counted with the earlier printers. A total of (typically) 66 lines, including linefeeds, has to be arranged per page if neat output is wanted. Six lines per inch is standard.

In addition to VIC graphics, these printers have a single user-defined character. It is invariably demonstrated by the Commodore symbol in the manuals. Printing a page of graphics therefore requires that the character be continually redefined.

Most software assumes device number 4 for a printer. However, that can be switched to device 5, so two printers can be used simultaneously, with PRINT#4 selecting one printer and PRINT#5 selecting the other.

The printers have a self-test facility, a loop in internal ROM which outputs the character set (except for reversed characters, which may cause overheating if used excessively). They also have a switch-on sequence. The older 1515 was liable to jam and appear completely dead when turned on, because the cam driving the ribbon stuck. If this happens, lightly flick the pivoting part of the cam to loosen it.

Other CBM printers include the MPS 801 for either the VIC or the 64 and a series of printers for the earlier PET/CBM machines. All PET/CBM printers require an IEEE interface connected to VIC's normal printer port to operate. The 4022 (largely an Epson printer) is the main PET/CBM printer, with a considerable number of features, including ten secondary addresses. A heavy-duty German printer and a very slow modified Olympia daisywheel are sometimes encountered, too.

Other Printers

Most printers have an RS-232 or Centronics interface; the latter is a parallel interface which uses multiwire ribbon-style cabling. IEEE interfaces are rarer; current loop interfaces are another relatively uncommon type. All can be connected to the VIC, with the VIC 1011A for RS-232, the VIC 1011B for current loop, proprietary Centronics adapters for the user port, and IEEE adapters for the serial port.

Note that RS-232 printers without a RAM buffer may not work correctly, because the VIC's handshaking has bugs. It is always advisable to test non-VIC combinations of equipment before a purchase is made, particularly if packaged software is to be used. A reliable word processor, such as *EasyScript*, makes some allowance for printer type, but other programs may not work correctly with all printers, particularly with features like margin and tab.

Printer Types

Several different printers are now available for the VIC. They are described below.

Teletypes. These are old-fashioned terminals, uppercase only, which communicate with computers via RS-232. In industry, they have been superseded by VDTs, but they can sometimes be found very cheaply.

Modified typewriters. Many typewriter manufacturers are now including interface sockets on their machines, so daisywheel machines with this dual function are likely to become popular. Golfball typewriters with interfaces are slower, though the impression is often slightly better.

Thermal and spark printers. These printers make up characters from columns of dots, like dot-matrix printers, but use methods that are less demanding mechanically. Thermal printers use short bursts of high temperature, while spark printers use short bursts of high voltage. These printers are cheap, but the paper is relatively costly and generally supplied in narrow rolls of limited usefulness.

Dot-matrix printers. These are by far the most widely used computer printers. The print head has typically seven to nine wires arranged vertically, and each wire is separately controlled by its own solenoid which drives the wire briefly into contact with ribbon and paper. Higher quality machines have more dots, so the image quality is better, although the delicacy of serifs and other features of typefaces is lost. An advantage of this method is that any characters within the limits imposed by the dot resolution can be generated, so dot-matrix printers often have internal switches for assorted European, Cyrillic, and other alphabets.

Daisywheel printers. A daisywheel has approximately 100 radial spokes, most (or all) of which hold a character at the tip. The wheels have low rotational inertia so they can be spun rapidly; common letters (e, t, a, i, o, n, s) are clustered together to reduce search time. A solenoid drives the letter against ribbon and paper; commonly used spokes will eventually fail and the wheel will need replacing. Speeds of 50 or 60 characters per second are common. Wheels and ribbons aren't standardized to any extent.

Daisywheel printers designed specifically for computers are expensive, and it may be that computer-compatible typewriters will become more popular for home use. Daisywheels cannot print graphics except by very tedious programming of single dots.

Some General Remarks

Printers normally use continuous fanfold paper. "Pin feed" or "sprocket feed" usually implies that the printer feed mechanism has fixed sprockets; "tractor feed" often implies that variable width paper is usable. "Pinchfeed" or "friction feed" indicates that rolls or sheets of unperforated stationery are accepted.

Most printers use endless-loop cartridge ribbons or, for higher quality, fixed-length carbon film ribbons. Ribbon cartridges are not standardized, so be sure that you have access to a reliable supplier.

External switches can range from simple paper control (linefeed, set top-of-form, formfeed, and so on) up to complete control over baud rate, parity, horizontal and vertical spacing, and so on. Some printers—Epson's RX-80 is one—have an automatic linefeed switch inside the machine. The switch is inaccessible without removing the lid and can be a liability if a printer is shared between computers.

Maintenance generally requires return of the machine to the manufacturer, often via a dealer. Fortunately, most printers are quite reliable. But it is still a good idea to be sure that some maintenance is possible and that it is not too costly.

The speed of a printer is usually quoted in characters per second or lines per minute; neither measure is very satisfactory. A lot depends on the density of the text to be output. Moreover, the figures quoted often seem to be untrue. If speed is important, check a printer before you buy it to assess its speed under actual conditions.

VIC compatibility is difficult with regard to graphics and upper/lowercase switching. Few printers offer the entire VIC character set, and interfaces may not handle the VIC upper/lowercase switch. However, in some cases, the interfaces themselves are programmable to allow for this.

Programming for Printers

The following discussion is not intended to replace printer manuals; there are too many possible variations to cover each one completely. Instead, it offers suggestions and hints on using printers correctly.

Commodore printers are controlled in two ways: by the secondary address or by special characters with an ASCII value usually below 32. The table of ASCII codes in Appendix I shows the conventional meanings of codes 0–31, most of which are more relevant to teletypes than to printers. CHR$(27), Escape, is widely used with non-CBM printers; anything following Escape is treated by a special routine independent of the rest and can be used to set any feature of a printer. In that respect, it works much like channel 15 of CBM disks. VIC printers could have used this method, rather than secondary addresses.

Some VIC control characters, like clear screen and delete, mean nothing to CBM printers and may cause them to hang up. A number of the ASCII control characters are irrelevant; the special characters controlling VIC printers therefore are chosen from those characters. The characters, and their functions, are given in Table 17-2.

Table 17-2. Printer Control Characters

CHR$(10)	Linefeed
CHR$(13)	Return
CHR$(17)	Lowercase
CHR$(18)	Reverse characters
CHR$(145)	Uppercase
CHR$(146)	Normal characters

All other controls have varied among different models of CBM printers, and it is risky to assume they will remain the same as they are on your model. For example, the user-definable single character is CHR$(8) but in the past has been CHR$(254). Secondary addresses too have varied; the 1515 uses OPEN 4,4,7 to set lowercase mode; and earlier models used this for uppercase.

To avoid such problems, you should let the VIC do the work of formatting and so on as much as possible. Otherwise, if you give your program to another user or change printers for some reason, you may be faced with the irritating job of rewriting PRINT# statements.

PRINT# and CMD

These two commands often cause confusion. They have almost identical effects; for example, PRINT#4,"HELLO" and CMD4, "HELLO" each print HELLO to file 4. The difference is that CMD leaves the printer in a "listening" mode, so future PRINT statements tend to be output to the printer.

However, CMD isn't really implemented properly. Although it works well with LIST (OPEN 4,4: CMD4: LIST lists to a printer) CMD 4 followed by a program with PRINT statements isn't reliable. GET, for example, makes the printing revert to the screen. It's usually best to use PRINT#. If you wish to divert some output to the screen, use something like the routine shown in Program 17-1.

Program 17-1. Diverting Output to the Screen

```
10 PRINT "OUTPUT TO PRINTER OR SCREEN (P/S)" : INP
   UT X$
20 IF X$="P" THEN D=4
30 IF X$="S" THEN D=3 : REM SCREEN DEVICE IS 3
40 OPEN D,D:REM NOW USE PRINT#D
```

The same method can select device 5 rather than device 4, if appropriate, and OPEN 128+D,D with PRINT#128+D can add an extra linefeed which some printers may need.

Use PRINT# after CMD to "unlisten" the printer, and return everything to normal, before CLOSE 4. Note that CMD4,; and PRINT#4,; each output nothing and can be used if it is important not to linefeed when these commands are executed.

Upper- and Lowercases

CBM printers don't generally behave like VIC printers which remain in either uppercase or lowercase until changed. They revert to uppercase unless specifically told otherwise. After a RETURN, the lowercase mode is canceled. Therefore, PRINT#4,CHR$(17); has to precede lowercase material, and PRINT#4,CHR$(145) must precede uppercase, if the two are mixed on a line (for example, lowercase letters mixed with graphics).

Formerly, LISTing a program in lowercase was difficult, but secondary address 7 allows this with some printers—OPEN 4,4,7:CMD4,"TITLE": LIST.

Formatting

PRINT USING in Chapter 6 can format numbers, inserting leading spaces and trailing zeros (as in 100.00). Alternatively, in BASIC, it's best to use something like SP$="{10 SPACES}": PRINT#4,RIGHT$(SP$+X$,10) instead of TAB. That right justifies a string (or numeral held as a string) by padding with spaces, then selecting a fixed length.

The simplest way to truncate numerals is to use an expression like PRINT#4, INT(X*100 + .5)/100 which rounds to the nearest hundredth. Some CBM printers have formatting, typically allowing one format at a time to be defined in COBOL-like form (for instance, OPEN 2,4,2: PRINT#2,"S$$$$$9.99" and OPEN 1,4,1). PRINT#1 then prints in a format defined by secondary address 2, so that 123.456 prints as +$123.45.

User-Defined Graphics/Screen Dump

Only one character can be defined at once. The 1515/1525 use CHR$(8); obviously six columns of seven dots have to be defined. The 1526 requires that you define eight columns of eight dots. In either case, all that's needed is PRINT#4, CHR$(8) followed by six bytes (or eight). You can do this with PRINT#4, CHR$(8) CHR$(22) CHR$(54) CHR$(96) CHR$(96) CHR$(54) CHR$(22). It can also be done with PRINT#4, CHR$(8)"!MM**&", or you can use PRINT#4,X$ where X$ is built from values in a DATA statement, starting with 8. An interesting use for this is to dump a high-resolution screen to the printer. Multicolor mode is less easy, since the printer can't distinguish four colors.

A different type of screen dump is the ML routine given as Program 17-2. It works with most printers and assumes the ordinary ASCII characters (no VIC graphics). It includes tests for the screen start position, and for lower- or upper-case mode. Use OPEN 4,4: CMD 4: SYS 828: PRINT#4: CLOSE 4.

Program 17-2. ML Screen Dump

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
10 FOR J=828 TO 930:READ X:POKE J,X:NEXT   :rem 14
20 DATA 169,0,133,140,133,141,133,142,173,136,2,13
   3,143                                   :rem 121
30 DATA 169,64,133,139,230,140,165,140,201,23,208,
   15                                      :rem 242
40 DATA 230,141,165,141,201,23,240,67,169,1,133,14
   0,32                                    :rem 72
50 DATA 215,202,162,0,161,142,41,127,36,139,208,6,
   36                                      :rem 241
60 DATA 129,240,19,208,33,36,129,208,9,72,169,2,44
   ,5                                      :rem 8
70 DATA 144,208,13,104,169,35,208,16,72,169,2,44,5
   ,144                                    :rem 99
80 DATA 208,5,104,9,64,208,3,104,9,96,32,210,255,2
   30                                      :rem 248
90 DATA 142,208,177,230,143,208,173,76,215,202
                                           :rem 218
```

Repeat

Some printers allow repetition of characters, notably of a single column of dots to build up a horizontal bar. A command like PRINT#4, CHR$(8) CHR$(26) CHR$(X) CHR$(255) CHR$(15); turns graphics on, turns repeat mode on, specifies the number of repetitions (1–255), specifies character definition (255 gives a solid column of dots), and returns to normal graphics.

As Chapter 12 shows, you can get the same result using ordinary VIC graphics, so this feature isn't enormously valuable.

Printer Presence

Some programmers find this useful as a reminder to users to switch on the printer. In its simplest form, the command is OPEN 4,4: POKE 154,4: SYS 65490: POKE 154,3: CLOSE 4: S=ST. When the printer is on, ST should be 0; when off, ST is −128, corresponding to ?DEVICE NOT PRESENT. SYS 65490 is the output routine at $FFD2, and the above routine in effect tries to output to file 4, but avoids crashing in the way that PRINT#4 does.

Spooling

The idea of spooling is that a file can be read from disk (and printed) while the VIC is left free to run programs normally (except that accessing the serial bus is prohibited). In principle this seems easy—the disk talks and the printer listens—but there is no simple way to accomplish it. The commands OPEN 8,8,8,"SEQ FILE": POKE 149,72: SYS 60974: POKE 149,104: SYS 60974: OPEN 4,4: CMD 4: POKE 154,3: POKE 152,0 (equivalent to a PET/CBM version) fail when used on the VIC.

Plotters

Plotters are unusual peripherals, most commonly used commercially for technical drawings of various types. Flat-bed plotters have two step motors controlling pen movement across and up/down, with a mechanism to lift the pen off the paper and reposition it as needed. Typically, eight directions of motion (N, NE, E, SE, etc.) can be selected. Small step sizes make for finer drawings, if the pen itself is fine enough, but tend to be slow. The fastest rate of plotting with cheap plotters is something like three inches per second, so be prepared for long delays, particularly if the interface is slow and if commands are sent with BASIC. Flat-bed plotters can be connected to the VIC and driven by PRINT# commands.

CBM's 1520 plotter uses 4½-inch wide unsprocketed paper and has four pens (typically black, red, blue, and green). It has built-in alphanumerics which can be scaled to four sizes; the smallest draws 22 characters per inch. The pens move across the paper, and up-and-down motion is provided by a roller that moves the paper itself. It connects to the serial port as device 6.

These plotters can be used to draw perspective pictures, including color-separation pairs in red and green. They can also draw geometrical patterns of the pins-and-string and Islamic styles, as described in Chapter 12. Yellow, magenta, and cyan pens could give an imitation of color separation printing.

Programming Plotters

Plotter programming is a specialized topic. Several programming methods are outlined below.

Lines

Program 17-3 is a subroutine that assumes a line, having a slope between zero and one, is to be drawn from left to right. (Other slopes, including vertical lines, are treated by analogous routines.) XD and YD are the distances to be plotted in the X and Y directions, M is the slope, and XP and YP keep track of the current X and Y positions relative to the start of the plot. Line 120 plots northeast whenever that gives a better approximation than east.

Program 17-3. Line Plotter

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
1ØØ M=YD/XD:YP=Ø                                   :rem 211
11Ø FOR XP=1 TO XD:PRINT#N,EAST                     :rem 95
12Ø IF M*XP>YP THEN PRINT#N,NORTHEAST:YP=YP+1:XP=X
    P+1:IF XP<XD GOTO 12Ø                          :rem 15Ø
13Ø NEXT                                           :rem 211
14Ø IF YP-1<YD THEN PRINT#N,NORTHEAST:YP=YP+1:GOTO
     14Ø                                            :rem 66
```

Circles

There are several methods to plot circles; one useful circle plotting subroutine is given in Program 17-4.

Program 17-4. Circle Plotter

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
1Ø Q=1Ø                                             :rem 8Ø
5ØØ REM Q=DEGREES SUBTENDED BY EACH STRAIGHT-LINE
    {SPACE}SEGMENT. Q=1Ø PLOTS A 36-SIDED FIGURE
                                                   :rem 2Ø7
51Ø G=R: H=Ø: REM R=RADIUS                         :rem 2Ø8
52Ø N=36Ø/Q                                         :rem 59
53Ø F=COS(Q*(↑)/18Ø):I=SIN(Q*(↑)/18Ø)              :rem 114
54Ø FOR J= Ø TO N                                   :rem 4Ø
55Ø C=G*F-H*I: A=G*I+H*F                            :rem 14
56Ø REM INSERT LINE ROUTINE HERE TO DRAW FROM G,H
    {SPACE}TO C,A                                  :rem 217
57Ø G=C: H=A                                        :rem 99
58Ø NEXT J                                          :rem 38
```

Modems

Most VIC/64 modems users have CBM equipment, either the VICmodem or the 1650 Automodem. Both are designed for the American phone system. The 1650 plugs into the base of the phone, while VICmodem needs a modular handset in order to dial correctly. Other modems can be used, notably acoustic modems, if your VIC has an RS-232 interface.

When the VIC is used to communicate with another computer, the users must decide which computer will "originate" the communication and which will "answer." For example, when a bulletin board system like CompuServe is to be accessed, the VIC is always set to "originate" while talking to the system.

To use a modem, first connect the modem to the computer (with the VIC turned off). Typically, plug the CBM modem into the user port and connect it to the phone. Then load and run the terminal software. Terminal software is the program that facilitates computer-to-computer talking via the modems. It may be on cartridge or on tape or disk. (Note: If you want to use *VICTERM* with memory expansion, you may find your version doesn't work properly. The solution is to use a loader, like the one given in Program 17-5, which reconfigures the VIC-20 like the unexpanded VIC while keeping the extra RAM.)

Program 17-5. *VICTERM* Loader

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
Ø REM ******{6 SPACES}VICTERM LOADER{6 SPACES}****
  ***                                        :rem 7
1 REM ***{4 SPACES}WORKS FOR ANY VIC-2Ø RAM
  {5 SPACES}***                              :rem 199
2 REM *** PUT THIS BEFORE VICTERM ON TAPE ***
                                             :rem 215
1Ø POKE 641,Ø: POKE 642,16: POKE 643,Ø: POKE 644,3
   Ø: POKE 648,3Ø                            :rem 177
2Ø POKE 198,1: POKE 631,131                   :rem 84
3Ø SYS 64818                                 :rem 1Ø9
```

You may want to use your own terminal software. BASIC, although slow, is about as fast as the modem, so this is often a useful thing to do, notably when talking to computers with slightly unusual characteristics or when trying out unusual maneuvers like transferring files of data.

Once the software has been loaded and run, call the number (either by dialing yourself or by inputting the number into the VIC and allowing the 1650 to dial for you). Wait for the carrier signal (a high-pitched tone). You may of course get a wrong number, outdated number, no reply, or a reply from a system operator (sysop). With some modems, you'll need to set the voice/data switch to be on "voice" at this stage.

Now, wait for the "carrier detected" red light to come on. Your software may print something like VIC CONNECTED. Either (or both) signifies that your modem has recognized the incoming frequency. Again, the actual procedure varies between modems; it's automatic with 1650s, but acoustic modems require you to put the handset into the cups of the modem and switch to "data" (so-called on-line) mode.

Wait for the system's first welcome frame to appear. Public systems ask for a password; it allows access, at a price, to their facilities. Keep yours secret. Then use the menu to select an item from what's available. CompuServe and other systems provide a large directory to help you with this; GO CBM 310 is a shortcut command with CompuServe.

These final two steps apply only to full-fledged systems. VIC's 22-column screen can be a problem, since most systems assume 40 columns, so always check on this if you don't want 40-column software.

Notes on Modems

It's helpful to know something about how modems work before looking at programming.

Modems and their software are designed around phone systems. This has several consequences: Data has to be transmitted serially, as bits rather than as bytes, so each end of the line needs a way to convert between parallel and serial transmission. In addition, the system needs some means of identifying the start of a byte; it also needs timing conventions so it can reliably detect individual bits.

Certain technical parameters are also important. Phone companies must maintain control over certain technical details of their lines: They cannot permit excessive voltages to get to exchanges, and some tones and signals may be reserved for diagnostic use. Such standards vary internationally; as a result, modems in different countries are liable to be incompatible. It may in fact be illegal to attach modems made in one country to phone lines in some other country.

Usually this isn't a problem. Direct-connect modems are designed to be isolated from the phone line so high voltages cannot pass either way. Acoustic modems, which generate and receive sounds and communicate them through telephone handsets, also present no voltage problems but face similar international problems over tone compatibility.

The most common American modem convention is the Bell 103, which is used by Commodore modems and by many computers. It is slow; generally characters are transmitted at 300 bits per second, or 300 baud. In practice, this amounts to 30 characters per second at most; if the phone connection is weak, the transmission rate drops, since characters have to be retransmitted. Even 30 characters a second takes half a minute to fill a 40-column screen, and some characters are likely to be used for information on color, screen format, and so on, so the overall rate is not dramatic. Still, it's faster than many people can read or talk.

Bell 103 uses a system called frequency shift keying, or FSK. It means that an "on" bit is transmitted with one frequency tone, while an "off" bit is transmitted with another. The tone of the signal carries information. In order that both ends of the line can talk, Bell 103 uses four tones altogether; in this way, messages can be simultaneously sent both ways.

The receiving equipment at either end has the job of sorting out which frequency is being received. All frequencies are relatively high pitched, in order to carry as much information as possible while still being within the frequency range handled by the phone system. The actual frequencies in originate mode are 1270 Hz to transmit "mark"; 1070 Hz to transmit "space"; 2225 Hz to receive mark; and 2025 Hz to receive space. In answer mode, the frequencies are the other way around. Note that the mark signal is the idle or carrier signal, present when nothing is happening but the system is ready and waiting.

When a modem is in operation, these tones are exchanged and deciphered. Conversion of bits into tones is called modulation, and the reverse process is called demodulation. The term *modem* is thus a shortened combination of the words *modulator* and *demodulator.*

At 300 baud, the VIC's modem receives 300 tones of 2225 or 2025 Hz every second. The VICmodem handles all of this with a single chip, using some other components to filter the four frequencies, and draws its power from the user port. (Note that the VIC's tape system is practically a modem. However, it sends square waves, not sine waves, which aren't suited to phone lines. But digital-to-analog converters make it very feasible to run a modem from the cassette port.)

Bytes (or "words") can be formatted in different ways, and every pair of communicating modems must be set to the same convention. Standard RS-232 has one start bit, seven data bits, one even parity bit, and one stop bit—a total of ten bits per byte sent. "Even parity" means that the eighth bit is set to 0 or 1 to make the total number of 0 (and 1) bits even. For example, the ASCII pattern for lowercase *a* is 1100001 (97 in decimal). For even parity, a parity bit of 1 is added, so the resulting pattern, 11100001, has an even number of 0's and 1's. (Bits are transmitted from the low bit first, so the parity bit is calculated and sent last.)

This is a security measure. Any received byte that doesn't conform to this pattern must be wrong and is retransmitted. Note that some seven-bit codes always send the same parity bit, either a space or mark, ignoring the security aspect. The start and stop bits are both signaled by transmitting a space (rather than a mark), so synchronization is always OK. OPENing an RS-232 file allows all these variables to be controlled by the programmer, within limits.

Note that VIC characters use eight bits, so standard ASCII isn't enough. In fact, much software simply ignores parity bits, using other error-checking methods instead. Getting the baud rate, the number of bits per word, and the number of start and stop bits right is necessary to successful modem communication; this is sometimes wrongly called "getting parity."

Converting bytes into bits and sending them, and the converse process of assembling bits into bytes, can be performed in software (as VIC's RS-232 handling does) or by chips like the UART (Universal Asynchronous Receiver-Transmitter; the "asynchronous" part means it can process data by watching for a start bit).

Error checking is a complex process, which basically uses hash totals sent after data as a check. With any system there must be some chance of completely random data just happening to conform to the check, and such events constitute undetected errors. This topic is far too elaborate to detail here. Generally, note that data is sent in batches (called "records") of 256 bytes each. Records with errors are retransmitted, and the overhead spent on this process can be as much as 50 percent of the ideal error-free transmission time, depending on the quality of the phone link. Error correction may be automatic, or software may use a recall feature if a frame is unacceptable.

Bell 103 modems use full-duplex, which means that either terminal can communicate at any time. Half-duplex is analogous to radio communication, where either direction is available, but normally only one at a time. The half-duplex switch turns off the so-called "echo-plexing" feature, a verification system which returns characters when they're received; if characters appear double, use this switch or use software which verifies the echoed characters. True half-duplex needs a line like RS-232's secondary channel to be able to interrupt unwanted messages.

"Smart" software implies that a system can download programs (get them and either run them or store them on disk or tape). Data files are more difficult to handle, because they don't transfer as simply as programs, having RETURN characters and so on embedded in them. They are also liable to exceed RAM storage.

"Downloading files," therefore, generally refers to programs and frames from data bases.

The two other common modem standards are the 202 and the 212A, which are faster than the 103. The 212A can work with the 103. However, the 202 and 212A are nowhere near as popular as the 103. Incidentally, the 103 system can operate at 600 baud; this may be worth trying. Non-CBM modems need an RS-232 adapter and cable; all acoustic modems come into this category.

One problem with acoustic modems may be getting the two cups which are supposed to fit the handset actually into place. A few modems forgo the rigid body in favor of a pair of cups on leads, so they can fit many shapes of phone. Incidentally, over short distances it's not necessary even to use a modem—two VICs or 64s can be connected by three lines between their user ports, or with RS-232 adapters.

Programming Modems

Programs for use with modems must allow for two things. First, the RS-232 file must be opened properly; second, transmissions both to and from the VIC must be allowed for. Both are fairly straightforward, though they may appear difficult.

Opening an RS-232 file. Only one file can be opened; the syntax is typically OPEN 2,2,0,CHR$(6). The device number must be 2. File number 2 is simplest, allowing PRINT#2 and GET#2 for output and input via the modem. The secondary address is irrelevant.

The filename consists of one or two characters in a string; the example is equivalent to CHR$(6)+CHR$(0). These parameters are explained fully in the next section; they assign eight bits of data per word, with one stop bit (and a start bit, implicit in the whole process), 300 baud transmission, no parity, and full-duplexing. So-called "three-line handshaking" is assumed.

This is the most common combination; it also evades a few bugs that are lurking in VIC's RS-232. OPEN 2,2,0,CHR$(38)CHR$(96) is the ASCII equivalent, assuming a seven-bit word and even parity.

Transmitting and receiving characters. All that's needed is a loop to get characters from the keyboard (and print them using PRINT#2) and to get (using GET#2) characters from the modem. BASIC may need delay loops in its output, or it may send characters too fast. For most purposes, some characters have to be converted, and BASIC provides an adaptable and quite easy means to do this. There are two reasons. One is that VIC ASCII is a bit different from true ASCII and from 64 ASCII, so unless you're happy with strange-looking lettering, conversion is necessary. The other is that it's useful to define some keys so they perform modem-specific things.

There's insufficient room here to list all possible terminal program permutations. However, Program 17-6 is a good example of a VIC program for use with a modem.

Program 17-6. VIC Terminal Program

Refer to the "Automatic Proofreader" article (Appendix C) before typing in this program.

```
100 OPEN 2,2,0,CHR$(6):REM OPENS 300 BAUD, 8 BIT,
    {SPACE}NO PARITY FILE.                  :rem 232
101 REM OPEN 2,2,0,CHR$(38)+CHR$(96) FOR ASCII 7 B
    ITS + EVEN PARITY                        :rem 84
200 DIM F%(255), T%(255)                     :rem 86
210 FOR J=0 TO 64: T%(J)=J: NEXT            :rem 140
220 FOR J=65 TO 90: T%(J)=J+32: NEXT :REM LOWER-CA
    SE                                       :rem 71
230 FOR J=91 TO 95: T%(J)=J: NEXT           :rem 204
240 FOR J=193 TO 218: T%(J)=J-128: NEXT :REM VIC U
    PPER-CASE                               :rem 202
250 T%(133)=27: T%(134)=3: T%(135)=19: T%(136)=17
                                             :rem 49
251 REM THESE ARE TRUE ASCII: IE ESC, DEL, CONTROL
    -C BREAK, CNTL-Q                         :rem 23
260 T%(137)=17: T%(138)=144                  :rem 24
261 REM THESE ARE ALL VIC: IE HOME, BLACK     :rem 3
300 FOR J=0 TO 255                          :rem 112
310 IF T%(J)>0 THEN F%(T%(J))=J              :rem 43
320 NEXT                                    :rem 212
400 PRINT CHR$(147) CHR$(14):REM CLEAR; LOWER-CASE
                                             :rem 88
500 IF PEEK(669)<PEEK(670) THEN 500          :rem 82
510 GET OUT$: IF OUT$>"" THEN PRINT#2,CHR$(T%(ASC(
    OUT$)));: PRINT OUT$;                   :rem 239
520 GET#2,IN$: IF IN$>"" THEN PRINT CHR$(F%(ASC(IN
    $)));                                   :rem 161
521 REM IN$=IN$ AND 127 FOR 7 BIT CODE.     :rem 226
530 GOTO 500                                :rem 102
```

Line 100 opens the file; line 101 is a typical alternative OPEN statement. Lines 200–261 allow for conversion between input and output characters. An alternative way to do this is to use several IF-THEN range comparisons; however, arrays are faster, since the correct value can be simply looked up. Integer arrays save space.

Lines 300–320 convert the "From" array into the inverse of the "To" array. Line 500 tests that output data has actually been sent.

The status byte ST can also be tested (but use PEEK(663) with VIC). The variable IN$ comes from the modem; OUT$ is actually fetched from the keyboard but is called OUT$ because it is to be sent to the other computer.

The RS-232 Interface

RS-232-C is a standard of the Electronic Industries Association. Its voltage convention is this: Negative means "mark," bit value 1, or OFF; positive means "space," bit value 0, or ON.

Pin numbering is from 1 to 13 (top) and 14 to 25 (bottom). Sometimes it is helpful to know their functions, which are listed in Table 17-3.

Table 17-3. RS-232 Pin Functions

Pin Number		Description
1	GND	Protective Ground
2	TX	Transmitted Data
3	RX	Received Data
4	RTS	Request to Send
5	CTS	Clear to Send
6	DSR	Data Set Ready
7	GND	Signal Ground (Common Return)
8	CD	Carrier Detector
9	CL+	Direct Current Loop (+)
10	CL−	Direct Current Loop (−)
11		Unassigned
12		Sec. Rec'd. Line Sig. Detector
13		Sec. Clear to Send
14		Secondary Transmitted Data
15		Transmission Signal Element Timing (DCE Source)
16		Secondary Received Data
17		Receiver Signal Element Timing (DCE Source)
18		Unassigned
19		Secondary Request to Send
20	DTR	Data Terminal Ready
21		Signal Quality Detector
22		Ring Indicator
23		Data Signal Rate Selector (DTE/DCE Source)
24		Transmit Signal Element Timing (DTE Source)
25		Unassigned

OPEN to RS-232 initializes a number of RAM locations and prepares for NMI interrupts which are used with RS-232. These interrupts disturb disk and tape timing, which is one reason neither the disk drive nor the Datassette can be used during transmission.

RS-232's OPEN ($F4C7 in the VIC) sets the parameters indicated in Table 17-4. If you OPEN and then PEEK, you'll see some of them. Most are reasonably straightforward. Two points are worth noting, however: OPEN to RS-232 lowers the top of memory by 512 bytes, making room for two first-in, first-out 256-byte buffers. BASIC pointers are altered to clear variables, so it's best to OPEN the file early in the program. The baud rate is controlled by reference to tables in ROM, which in VIC has 11 usable values. It is possible to use other baud rates.

ML programmers may want to alter the NMI vector into RAM so the tables can be changed. Alter the first tabled value to generate the new baud rate; and remember, after OPEN, to POKE the vector at $299 with twice that value, plus 200. To convert ROM values into equivalent baud rates, use 50*EXP(9.23308 − LOG (VALUE + 100)).

Table 17-4. Locations Set by OPEN to RS-232

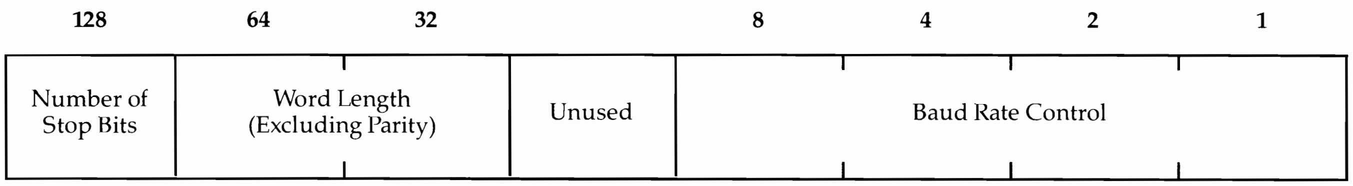
Location		Explanation
$A7	167	Receive bit storage
$A8	168	RX bit count
$A9	169	RX start bit flag
$AA	170	RX byte shifts in here
$AB	171	RX parity bit
$B4	180	TX bit count
$B5	181	Next bit for TX
$B6	182	TX byte shifts out from here
$F7/F8	247/248	Pointer to start of input buffer
$F9/FA	249/250	Pointer to start of output buffer
$0293	659	Control Register (e.g., 6)
$0294	660	Command Register (e.g., 0)
$0295/0296	661/662	Two other unused parameters
$0297	663	ST value for RS-232
$0298	664	9, 8, 7, or 6 bits in word + 1
$0299/029A	665/666	2*timer value + 200
$029B	667	End of Receive FIFO Buffer
$029C	668	Start of Receive Buffer
$029D	669	Start of Transmit Buffer
$029E	670	End of Transmit Buffer

Control Register and Command Register

Values in these registers control the way RS-232 is configured. There are six parameters involved. For example, OPEN 2,2,2,CHR$(6)+CHR$(0) assumes one stop bit, eight bits per word, 300 baud, no parity bit, full duplex and the usual three-line handshake. The control register is set by the first CHR$ value, and the command register is set by the second. Figures 17-1 and 17-2 give details on the control register and command register.

Finally, six bits of location 663 report conditions resulting from RS-232 use. If bit 0 is set, there is a parity bit error or some inconsistency if parity is set incorrectly. Bit 1 being set indicates an error in structure of received bits, perhaps due to noise. Bit 2 is set when the receive buffer is full (that is, when data is coming in too fast). Bit 4 is set when the clear to send signal is off (when the remote terminal is not ready to receive); bit 6 is set when the remote terminal is not ready to send. When bit 7 is set, a break has been detected. VIC's implementation of some of these features in fact has bugs, most of which were taken out for the 64's ROM.

Figure 17-1. The Control Register

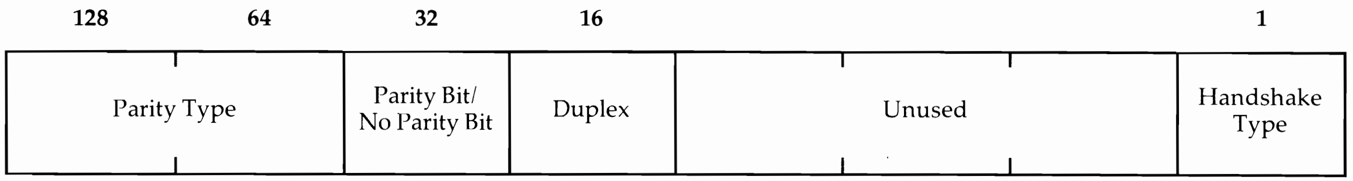


0 = single
1 = two

00 = 8 bits
01 = 7 bits (e.g., ASCII)
10 = 6 bits
11 = 5 bits (e.g., Baudot)

0001 = 50 Baud
0010 = 75
0011 = 110
0100 = 134.5
0101 = 150
0110 = 300
0111 = 600
1000 = 1200
1001 = 1800
1010 = 2400
1011 = 3600

Figure 17-2. The Command Register



Any
00 = Odd Parity
01 = Even Parity
10 = Mark Bit
11 = Space Bit

0 = None
1 = Parity Bit

0 = Full
1 = Half

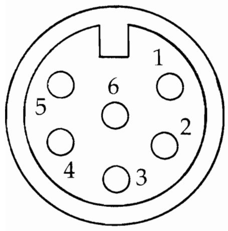
0 = 3 – Line
1 = X – Line

The Serial Port

The VIC's serial port is peculiar to Commodore; it's adapted from the IEEE interface of CBM machines. IEEE-488 is the document describing this interface.

The VIC uses a simplified, nonstandard version of this, which carries serial (not parallel) data and is comparatively slow. Figure 17-3 shows the port's six connections as they appear looking at the VIC from the back (the serial port is next to the cassette port).

Figure 17-3. VIC Serial Port



Pin 1	SRQ in	CB1 of VIA 2
Pin 2	Ground	
Pin 3	ATN in	Connects with user port
Pin 3	ATN out	PA7 of VIA 1
Pin 4	CLK in	PA0 of VIA 1
Pin 4	CLK out	CA2 of VIA 2
Pin 5	Data in	PA1 of VIA 1
Pin 5	Data out	CB2 of VIA 2
Pin 6	RESET	Connected to VIC reset

Pin 6 is connected to the VIC's reset line, which is why the disk drive resets when the VIC is switched on. Pin 5 transmits data bits. Pin 1, "Service Request," allows devices to request service from the VIC; CLK is a clock signal. ATN (Attention) is described below.

Both VIAs are used in processing. The part of ROM handling this can be inspected in detail by looking at the places where bit 1 of $911F is used; this line inputs data bits. Data is transmitted by line CB2 of VIA 2, so $912C controls data output. ML loops like LDA $911F/ CMP $911F/ BNE −8 test the clock. ATN out is set high with LDA $911F/ ORA #$80/ STA $911F; in BASIC, POKE 37151,128 OR PEEK (37151).

Briefly, the serial bus is controlled by the VIC; devices on the bus are talkers, listeners, or talkers/listeners. Printers listen; disks both talk and listen. The Kernal has routines to make devices talk, listen, untalk, and unlisten, meaning in effect that they're on or off. BASIC handles all this itself, apart from a few special effects.

Commands are sent to devices when ATN is low (the bit value is 0). When ATN is set high again, all the bytes sent are interpreted as data. When ATN is low, typically a single byte is sent as a command; that byte is interpreted by the device as follows: If it is in the range $20–$3E, it means listen; if it's $3F, it means unlisten. $40–$5E mean talk; $5F means untalk. $60–$7F indicate secondary addresses. This is why secondary addresses are stored in the VIC with 96 decimal added, and why the Kernal LISTEN and TALK routines begin with ORA #$20 and ORA #$40.

A printer can be made to print, without opening a file, by setting the device number to 4, calling Kernal LISTEN, setting ATN out high, sending characters with CHROUT, and finally unlistening with CLRCHN. Whenever files are open to a device, the device is first made a talker or a listener. Then the secondary address is sent (the Kernal has two routines for this purpose) so the device knows which file to address.

Appendices

A Beginner's Guide to Typing In Programs

What Is a Program?

A computer cannot perform any task by itself. Like a car without gas, a computer has potential. But without a program, it isn't going anywhere.

Most of the programs published in this book are written in a computer language called BASIC. BASIC is easy to learn and is built into all VIC-20s.

BASIC Programs

Computers can be picky. Unlike the English language, which is full of ambiguities, BASIC usually has only one right way of stating something. Every letter, character, or number is significant. A common mistake is substituting a letter such as O for the numeral 0, a lowercase l for the numeral 1, or an uppercase B for the numeral 8. Also, you must enter all punctuation such as colons and commas just as they appear in the book. Spacing can be important. To be safe, type in the listings *exactly* as they appear.

Braces and Special Characters

The exception to this typing rule is when you see something inside braces, such as {DOWN}. Anything within a set of braces is a special character or characters that cannot easily be listed on a printer. When you come across such a special statement, refer to Appendix B, "How to Type In Programs."

About DATA Statements

Some programs contain a section or sections of DATA statements. These lines provide information needed by the program. Some DATA statements contain actual programs in machine language; others contain graphics codes. These lines are especially sensitive to errors.

If a single number in any one DATA statement is mistyped, your machine could lock up or crash. If this happens, the keyboard and STOP key may seem dead, and the screen may go blank.

But don't panic; no damage has been done. To regain control, you have to turn off your computer, then turn it back on. This will erase whatever program was in memory, *so always save a copy of your program before you run it.* If your computer crashes, you can reload the program and look for your mistake.

Sometimes a mistyped DATA statement will cause an error message when the program is run. The error message may refer to the program line that READs the data. *However, the error is still in the DATA statements.*

Get to Know Your Machine

You should familiarize yourself with your computer before attempting to type in a program. Learn the statements you use to store and retrieve programs from tape or disk. You'll want to save a copy of your program, so that you won't have to type it in every time you want to use it. Learn to use your machine's editing functions. How do you change a line if you made a mistake? You can always retype the line, but you at least need to know how to backspace. Do you know how to enter reverse video, lowercase, and control characters? It's all explained in your VIC's manual, *Personal Computing on the VIC.*

A Quick Review

1. Type in the program a line at a time, in order. Press RETURN at the end of each line. Use the INST/DEL key to correct mistakes.
2. Check the line you've typed against the line in the book. You can check the entire program again if you get an error when you run the program.
3. Make sure you've entered statements in braces using the appropriate control key (see Appendix B, "How to Type In Programs").

Appendix B

How to Type In Programs

Many of the programs in this book contain special control characters (cursor control, color keys, reverse characters, and so on). To make it easy to know exactly what to type when entering one of these programs into your computer, we have established the following listing conventions.

Generally, VIC-20 program listings contain words within braces which spell out any special characters: {DOWN} means to press the cursor down key, while {5 SPACES} tells you to press the space bar five times.

To indicate that a key should be *shifted* (hold down the SHIFT key while pressing the other key), the key would be underlined in our listings. For example, S would mean to type the S key while holding down the SHIFT key. This would appear on your screen as a heart symbol. If you find an underlined key enclosed in braces (e.g., {10 N}), you should type the key as many times as indicated. In this case, you would enter ten shifted N's.

If a key is enclosed in special brackets, [<>], you should hold down the *Commodore key* while pressing the key inside the special brackets. (The Commodore key is the key in the lower-left corner of the keyboard.) Again, if the key is preceded by a number, you should press the key as many times as necessary.

About the *quote mode:* You know that you can move the cursor around the screen with the CRSR keys. Sometimes a programmer will want to move the cursor under program control. That's why you see all the {LEFT}'s, {HOME}'s, and {BLU}'s in our programs. The only way the computer can tell the difference between direct and programmed cursor control is the quote mode.

Once you press the quote (the double quote, SHIFT-2), you are in the quote mode. For instance, if you type something and then try to change it by moving the cursor left, you'll only get a bunch of reverse-video lines. These are the symbols for cursor left. The only editing key that isn't programmable is the DEL key; you can still use DEL to back up and edit the line. Once you type another quote, you are out of quote mode.

You also go into quote mode when you INSerT spaces into a line. In any case, the easiest way to get out of quote mode is simply to press RETURN. You'll then be out of quote mode and can cursor up to the mistyped line and fix it.

Use the following table when entering cursor and color control keys:

When You Read:	Press:		See:	When You Read:	Press:		See:
{CLR}	SHIFT	CLR/HOME	♥	{GRN}	CTRL	6	↑
{HOME}		CLR/HOME	S	{BLU}	CTRL	7	←
{UP}	SHIFT	↑ CRSR ↓	●	{YEL}	CTRL	8	π
{DOWN}		↑ CRSR ↓	Q	{ F1 }		f1	▔
{LEFT}	SHIFT	← CRSR →	│	{ F2 }	SHIFT	f1	╮
{RIGHT}		← CRSR →	]	{ F3 }		f3	─
{RVS}	CTRL	9	R	{ F4 }	SHIFT	f3	╰
{OFF}	CTRL	0	▁	{ F5 }		f5	│
{BLK}	CTRL	1	▕	{ F6 }	SHIFT	f5	╯
{WHT}	CTRL	2	E	{ F7 }		f7	│
{RED}	CTRL	3	£	{ F8 }	SHIFT	f7	▏
{CYN}	CTRL	4	◣	←	←		←
{PUR}	CTRL	5	▒	↑	SHIFT	↑	π

Appendix C

The Automatic Proofreader

Charles Brannon

"The Automatic Proofreader" will help you type in program listings without typing mistakes. It is a short error-checking program that hides itself in memory. When activated, it lets you know if you have made a mistake immediately after typing a line from a program listing.

Preparing the Proofreader

Please read these instructions carefully before typing in any programs in this book.

1. Using the listing below, type in the Proofreader. Be very careful when entering the DATA statements—don't type an l instead of a 1, an O instead of a 0, extra commas, etc.
2. Save the Proofreader on tape or disk at least twice *before running it for the first time.* This is very important because the Proofreader erases part of itself when you first type RUN.
3. After the Proofreader is saved, type RUN. It will check itself for typing errors in the DATA statements and warn you if there's a mistake. Correct any errors and save the corrected version. Keep a copy in a safe place. You'll need it again and again, every time you enter a program from this book, *COMPUTE!'s Gazette,* or *COMPUTE!* magazine.
4. When a correct version of the Proofreader is run, it activates itself and you are then ready to enter a program listing. If you press RUN/STOP–RESTORE, the Proofreader is disabled. To reactivate it, just type the command SYS 886 and press RETURN.

Using the Proofreader

Most listings in this book have a *checksum number* appended to the end of each line, for example, *":rem 123". Don't enter this statement when typing in a program.* It is just for your information. The rem makes the number harmless if someone does type it in. It will, however, use up memory if you enter it, and it will confuse the Proofreader, even if you entered the rest of the line correctly.

When you type in a line from a program listing and press RETURN, the Proofreader displays a number at the top of your screen. *This checksum number must match the checksum number in the printed listing.* If it doesn't, it means you typed the line differently than the way it is listed. Immediately recheck your typing. Remember, don't type the rem statement with the checksum number; it is published only so you can check it against the number which appears on your screen.

The Proofreader is not picky with spaces. It will not notice extra spaces or missing ones. This is for your convenience, since spacing is generally not important. But occasionally proper spacing *is* important, so be extra careful with spaces.

One sort of error that the Proofreader will *not* catch is transposition. If you type PIRNT in a program line instead of PRINT, the Proofreader will not detect the error because all the proper characters are present (even if they are in the wrong order). If a program fails to work even though the Proofreader says all the lines are correct, look for an error of this type.

There's another thing to watch out for: If you enter the line by using abbreviations for commands, the checksum will not match up. But there is a way to make the Proofreader check it. After entering the line, LIST it. This eliminates the abbreviations. Then move the cursor up to the line and press RETURN. It should now match the checksum. You can check whole groups of lines this way.

Special Tape SAVE Instructions

When you're through typing a listing, you must disable the Proofreader before saving the program on tape. Disable the Proofreader by pressing RUN/STOP–RESTORE (hold down the RUN/STOP key and sharply hit the RESTORE key). This procedure is not necessary for disk SAVEs, *but you must disable the Proofreader this way before a tape SAVE.*

SAVE to tape erases the Proofreader from memory, so you'll have to load and run it again if you want to type another listing. SAVE to disk does not erase the Proofreader.

Hidden Perils

The proofreader's home in the VIC is not a very safe haven. Since the cassette buffer is wiped out during tape operations, you need to disable the Proofreader with RUN/STOP–RESTORE before you save your program. This applies only to tape use. Disk users have nothing to worry about.

Not so for VIC owners with tape drives. What if you type in a program in several sittings? The next day, you come to your computer, load and run the Proofreader, then try to load the partially completed program so you can add to it. But since the Proofreader is trying to hide in the cassette buffer, it is wiped out.

What you need is a way to load the Proofreader after you've loaded the partial program. The problem is that a tape LOAD to the buffer destroys what it's supposed to load.

After you've typed in and run the Proofreader, enter the following lines in direct mode (without line numbers) exactly as shown:

```
A$="PROOFREADER.T": B$="{10 SPACES}": FOR X = 1 TO
   4: A$=A$+B$: NEXT X
FOR X = 886 TO 1018: A$=A$+CHR$(PEEK(X)): NEXT X
OPEN 1,1,1,A$:CLOSE 1
```

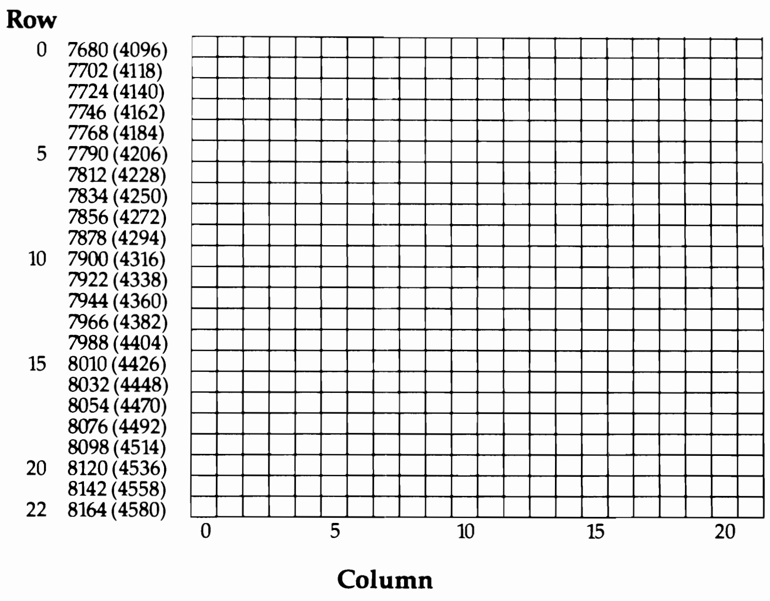
After you enter the last line, you will be asked to press RECORD and PLAY on your cassette recorder. Put this program at the beginning of a new tape; this gives you a new way to load the Proofreader. Anytime you want to bring the Proofreader into memory without disturbing anything else, put the cassette in the tape drive, rewind, enter OPEN1:CLOSE1, and press PLAY on the recorder. You can then start the Proofreader by typing SYS 886. To test this, type in PRINT PEEK (886). It should return the number 173. If it does not, repeat the steps above, making sure that A$ contains 13 characters (PROOFREADER.T) and that B$ contains 10 spaces.

You can now reload the Proofreader into memory whenever LOAD or SAVE destroys it, restoring your personal typing helper.

Automatic Proofreader

```
100 PRINT"{CLR}PLEASE WAIT...":FORI=886TO1018:READ
    A:CK=CK+A:POKEI,A:NEXT
110 IF CK<>17539 THEN PRINT"{DOWN}YOU MADE AN ERRO
    R":PRINT"IN DATA STATEMENTS.":END
120 SYS886:PRINT"{CLR}{2 DOWN}PROOFREADER ACTIVATE
    D.":NEW
886 DATA 173,036,003,201,150,208
892 DATA 001,096,141,151,003,173
898 DATA 037,003,141,152,003,169
904 DATA 150,141,036,003,169,003
910 DATA 141,037,003,169,000,133
916 DATA 254,096,032,087,241,133
922 DATA 251,134,252,132,253,008
928 DATA 201,013,240,017,201,032
934 DATA 240,005,024,101,254,133
940 DATA 254,165,251,166,252,164
946 DATA 253,040,096,169,013,032
952 DATA 210,255,165,214,141,251
958 DATA 003,206,251,003,169,000
964 DATA 133,216,169,019,032,210
970 DATA 255,169,018,032,210,255
976 DATA 169,058,032,210,255,166
982 DATA 254,169,000,133,254,172
988 DATA 151,003,192,087,208,006
994 DATA 032,205,189,076,235,003
1000 DATA 032,205,221,169,032,032
1006 DATA 210,255,032,210,255,173
1012 DATA 251,003,133,214,076,173
1018 DATA 003
```

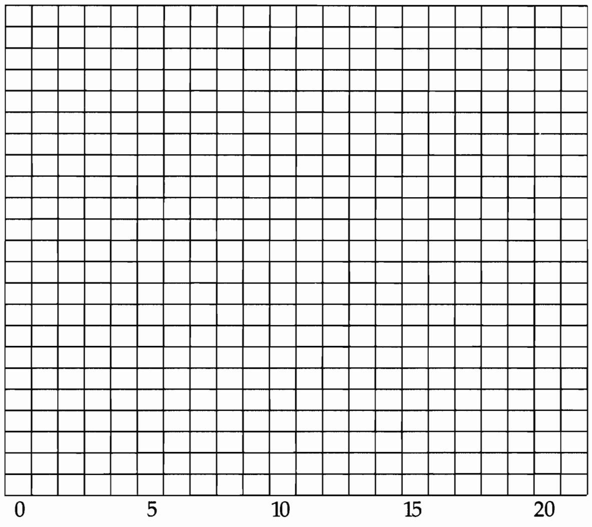
Screen Location Table



Note: Numbers in parentheses are for VICs with 8K or more of memory expansion.

Screen Color Memory Table

Row	
0	38400 (37888)
	38422 (37910)
	38444 (37932)
	38466 (37954)
	38488 (37976)
5	38510 (37998)
	38532 (38020)
	38554 (38042)
	38576 (38064)
	38598 (38086)
10	38620 (38108)
	38642 (38130)
	38664 (38152)
	38686 (38174)
	38708 (38196)
15	38730 (38218)
	38752 (38240)
	38774 (38262)
	38796 (38284)
	38818 (38306)
20	38840 (38328)
	38862 (38350)
22	38884 (38372)



Column

Note: Numbers in parentheses are for VICs with 8K or more of memory expansion.

Appendix F

Screen Color Codes

Color:	Black	White	Red	Cyan	Purple	Green	Blue	Yellow
Code:	0	1	2	3	4	5	6	7

Appendix G

Usable Graphics and Screen Combinations

Usable Graphics And Screen Combinations (Decimal)

Start of Screen Memory	Start of Color Memory			Start of Graphics Character Memory								
				Built-In ROM Characters				User-Defined Graphics in RAM				
				Uppercase 32768	Reverse 33792	Lowercase 34816	Reverse 35840	0	4096	5120	6144	7168
		POKE 648	POKE 36866	POKE 36869								
0	37888	0	22	128	129	130	131	136	140	141	142	143
512	38400	2	150									
4096	37888	16	22	192	193	194	195	200	204	205	206	207
4608	38400	18	150									
5120	37888	20	22	208	209	210	211	216	220	221	222	223
5632	38400	22	150									
6144	37888	24	22	224	225	226	227	232	236	237	238	239
6656	38400	26	150									
7168	37888	28	22	240	241	242	243	248	252	253	254	255
7680	38400	30	150									

Usable Graphics And Screen Combinations (Hex)

Start of Screen Memory	Start of Color Memory			Start of Graphics Character Memory								
				Built-In ROM Characters				User-Defined Graphics in RAM				
				Uppercase $8000	Reverse $8400	Lowercase $8800	Reverse $8C00	$0	$1000	$1400	$1800	$1C00
		POKE $288	POKE $9002	POKE $9005								
$0000	$9400	$00	$16	$80	$81	$82	$83	$88	$8C	$8D	$8E	$8F
$0200	$9600	$02	$96									
$1000	$9400	$10	$16	$C0	$C1	$C2	$C3	$C8	$CC	$CD	$CE	$CF
$1200	$9600	$12	$96									
$1400	$9400	$14	$16	$D0	$D1	$D2	$D3	$D8	$DC	$DD	$DE	$DF
$1600	$9600	$16	$96									
$1800	$9400	$18	$16	$E0	$E1	$E2	$E3	$E8	$EC	$ED	$EE	$EF
$1A00	$9600	$1A	$96									
$1C00	$9400	$1C	$16	$F0	$F1	$F2	$F3	$F8	$FC	$FD	$FE	$FF
$1E00	$9600	$1E	$96									

Usable Graphics and Screen Combinations.

To use this table:

1. To determine the start of screen memory and graphics definitions, PEEK the contents of locations 36869 and either 648 or 36866, then compare the results to the table. For example, when 36869 contains 194 and 648 contains 16, screen memory starts with location 4096 and lowercase characters are in use.
2. To set the start of screen memory and graphics definitions, POKE values from the table into locations 36869, 648, and 36866. For example, to set up a user-defined graphics area starting at location 7168, with the screen starting at 7680, you would use:

   POKE 36869,255: POKE 648,30: POKE 36866,150

   This establishes 512 bytes for redefined graphics, enough for 64 characters. With BASIC, remember to protect the graphics definition area from being overwritten by using:

   POKE 56,(7168/256): CLR

   to lower the top of memory.

Screen and Border Colors

	Border							
Screen	Black	White	Red	Cyan	Purple	Green	Blue	Yellow
Black	8	9	10	11	12	13	14	15
White	24	25	26	27	28	29	30	31
Red	40	41	42	43	44	45	46	47
Cyan	56	57	58	59	60	61	62	63
Purple	72	73	74	75	76	77	78	79
Green	88	89	90	91	92	93	94	95
Blue	104	105	106	107	108	109	110	111
Yellow	120	121	122	123	124	125	126	127
Orange	136	137	138	139	140	141	142	143
Light Orange	152	153	154	155	156	157	158	159
Pink	168	169	170	171	172	173	174	175
Light Cyan	184	185	186	187	188	189	190	191
Light Purple	200	201	202	203	204	205	206	207
Light Green	216	217	218	219	220	221	222	223
Light Blue	232	233	234	235	236	237	238	239
Light Yellow	248	249	250	251	252	253	254	255

ASCII Codes

Hex	Dec	Character
05	5	WHITE
08	8	DISABLE SHIFT-COMMODORE
09	9	ENABLE SHIFT-COMMODORE
0D	13	RETURN
0E	14	LOWERCASE
11	17	CURSOR DOWN
12	18	REVERSE VIDEO ON
13	19	HOME
14	20	DELETE
1C	28	RED
1D	29	CURSOR RIGHT
1E	30	GREEN
1F	31	BLUE
20	32	SPACE
21	33	!
22	34	"
23	35	#
24	36	$
25	37	%
26	38	&
27	39	'
28	40	(
29	41	)
2A	42	*
2B	43	+
2C	44	,
2D	45	–
2E	46	.
2F	47	/
30	48	0
31	49	1
32	50	2
33	51	3
34	52	4
35	53	5
36	54	6
37	55	7
38	56	8
39	57	9
3A	58	:
3B	59	;
3C	60	<
3D	61	=
3E	62	>
3F	63	?
40	64	@
41	65	A
42	66	B
43	67	C
44	68	D
45	69	E
46	70	F
47	71	G
48	72	H
49	73	I
4A	74	J
4B	75	K
4C	76	L
4D	77	M
4E	78	N
4F	79	O
50	80	P
51	81	Q
52	82	R

Hex	Dec	Character	Hex	Dec	Character
53	83	S	78	120	♣
54	84	T	79	121	▕
55	85	U	7A	122	♦
56	86	V	7B	123	┼
57	87	W	7C	124	▒
58	88	X	7D	125	│
59	89	Y	7E	126	π
5A	90	Z	7F	127	◥
5B	91	[	85	133	f1
5C	92	£	86	134	f3
5D	93	]	87	135	f5
5E	94	↑	88	136	f7
5F	95	←	89	137	f2
60	96	─	8A	138	f4
61	97	♠	8B	139	f6
62	98	│	8C	140	f8
63	99	─	8D	141	SHIFT-RETURN
64	100	▔	8E	142	UPPERCASE
65	101	▔	90	144	BLACK
66	102	▁	91	145	CURSOR UP
67	103	▏	92	146	REVERSE VIDEO OFF
68	104	▕	93	147	CLEAR SCREEN
69	105	╮	94	148	INSERT
6A	106	╰	9C	156	PURPLE
6B	107	╯	9D	157	CURSOR LEFT
6C	108	⌞	9E	158	YELLOW
6D	109	╲	9F	159	CYAN
6E	110	╱	A0	160	SHIFT-SPACE
6F	111	⌜	A1	161	▌
70	112	⌝	A2	162	▄
71	113	●	A3	163	▔
72	114	▁	A4	164	▁
73	115	♥	A5	165	▏
74	116	▏	A6	166	▒
75	117	╭	A7	167	▕
76	118	╳	A8	168	▒
77	119	○	A9	169	◤

Hex	Dec	Character	Hex	Dec	Character
AA	170	▐	CF	207	□
AB	171	⊞	D0	208	□
AC	172	▗	D1	209	●
AD	173	┖	D2	210	▭
AE	174	◲	D3	211	♥
AF	175	▂	D4	212	▯
B0	176	┌	D5	213	◿
B1	177	⊟	D6	214	⊠
B2	178	⊞	D7	215	○
B3	179	⊞	D8	216	♣
B4	180	▏	D9	217	▯
B5	181	▌	DA	218	♦
B6	182	▐	DB	219	⊞
B7	183	▔	DC	220	▒
B8	184	▀	DD	221	◫
B9	185	▄	DE	222	π
BA	186	▕	DF	223	◥
BB	187	▖	E0	224	SPACE
BC	188	▜	E1	225	▌
BD	189	┘	E2	226	▄
BE	190	▘	E3	227	□
BF	191	▚	E4	228	□
C0	192	⊟	E5	229	▯
C1	193	♠	E6	230	▒
C2	194	◫	E7	231	▯
C3	195	⊟	E8	232	▒
C4	196	▭	E9	233	◤
C5	197	□	EA	234	▐
C6	198	⊟	EB	235	⊞
C7	199	▯	EC	236	▗
C8	200	▯	ED	237	┖
C9	201	◺	EE	238	◲
CA	202	◹	EF	239	▂
CB	203	◸	F0	240	┌
CC	204	□	F1	241	⊞
CD	205	⧅	F2	242	⊞
CE	206	⧄	F3	243	⊞

Hex	Dec	Character
F4	244	▎
F5	245	▍
F6	246	▐
F7	247	▔
F8	248	▀
F9	249	▃
FA	250	▕
FB	251	▖
FC	252	▝
FD	253	┘
FE	254	▘
FF	255	π

1. 0–4, 6–7, 10–12,15–16, 21–27, 128–132, 143, and 149–155 have no effect.
2. 192–223 same as 96–127, 224–254 same as 160–190, 255 same as 126.

Screen Codes

Hex	Dec	Uppercase and Full Graphics Set	Lower- and Uppercase	Hex	Dec	Uppercase and Full Graphics Set	Lower- and Uppercase
00	0	@	@	1F	31	←	←
01	1	A	a	20	32	-space-	
02	2	B	b	21	33	!	!
03	3	C	c	22	34	"	"
04	4	D	d	23	35	#	#
05	5	E	e	24	36	$	$
06	6	F	f	25	37	%	%
07	7	G	g	26	38	&	&
08	8	H	h	27	39	'	'
09	9	I	i	28	40	(	(
0A	10	J	j	29	41	)	)
0B	11	K	k	2A	42	*	*
0C	12	L	l	2B	43	+	+
0D	13	M	m	2C	44	,	,
0E	14	N	n	2D	45	–	–
0F	15	O	o	2E	46	.	.
10	16	P	p	2F	47	/	/
11	17	Q	q	30	48	0	0
12	18	R	r	31	49	1	1
13	19	S	s	32	50	2	2
14	20	T	t	33	51	3	3
15	21	U	u	34	52	4	4
16	22	V	v	35	53	5	5
17	23	W	w	36	54	6	6
18	24	X	x	37	55	7	7
19	25	Y	y	38	56	8	8
1A	26	Z	z	39	57	9	9
1B	27	[	[	3A	58	:	:
1C	28	£	£	3B	59	;	;
1D	29	]	]	3C	60	<	<
1E	30	↑	↑	3D	61	=	=

Hex	Dec	Uppercase and Full Graphics Set	Lower- and Uppercase	Hex	Dec	Uppercase and Full Graphics Set	Lower- and Uppercase
3E	62	>	>	5F	95	◥	▨
3F	63	?	?	60	96	- -space- -	
40	64	─	─	61	97	▌	▌
41	65	♠	A	62	98	▄	▄
42	66	│	B	63	99	▔	▔
43	67	─	C	64	100	▁	▁
44	68	─	D	65	101	▏	▏
45	69	─	E	66	102	▒	▒
46	70	─	F	67	103	▕	▕
47	71	│	G	68	104	▒	▒
48	72	│	H	69	105	◤	▨
49	73	╮	I	6A	106	▐	▐
4A	74	╰	J	6B	107	├	├
4B	75	╯	K	6C	108	▗	▗
4C	76	└	L	6D	109	└	└
4D	77	╲	M	6E	110	┐	┐
4E	78	╱	N	6F	111	▂	▂
4F	79	┌	O	70	112	┌	┌
50	80	┐	P	71	113	┴	┴
51	81	●	Q	72	114	┬	┬
52	82	─	R	73	115	┤	┤
53	83	♥	S	74	116	▎	▎
54	84	│	T	75	117	▌	▌
55	85	╭	U	76	118	▐	▐
56	86	╳	V	77	119	▔	▔
57	87	○	W	78	120	▀	▀
58	88	♣	X	79	121	▃	▃
59	89	│	Y	7A	122	▁	✓
5A	90	♦	Z	7B	123	▖	▖
5B	91	┼	┼	7C	124	▝	▝
5C	92	▒	▒	7D	125	┘	┘
5D	93	│	│	7E	126	▘	▘
5E	94	π	▒	7F	127	▚	▚

128–255 are reverse video of 0–127.

VIC Chip Registers

Addresses:	Functions of Registers:			Summary:
$9000 36864	Interlace 0 off/1 on	Left Margin (4 pixels accuracy)		Display Size/Shape
$9001 36865	Top Margin (2 pixels accuracy)			
$9002 36866	Bit 9 of Screen Start	Number of Columns (usually 22)		
$9003 36867	Bit 0 of Scan Line	Number of Rows (usually 23)	Char. Size 0 8×8 1 8×16	
$9004 36868	Current TV Scan Line, bits 8–1			TV Line
$9005 36869	Screen Address Start $\overline{\text{Bit 15}}$ Bit 12 Bit 11 Bit 10	Character Definitions Start $\overline{\text{Bit 15}}$ Bit 12 Bit 11 Bit 10		Screen/Graphics Control
$9006 36870	Light Pen—Horizontal			Read-Only Input Registers
$9007 36871	Light Pen—Vertical			
$9008 36872 (37000)	First Potentiometer (Paddle X) Reading			
$9009 36873 (37001)	Second Potentiometer (Paddle Y) Reading			
$900A 36874 (37002)	0 off 1 on	Low-Frequency Sound Countdown		Sound
$900B 36875 (37003)	0 off 1 on	Medium-Frequency Sound Countdown		
$900C 36876 (37004)	0 off 1 on	High-Frequency Sound Countdown		
$900D 36877 (37005)	0 off 1 on	Noise Countdown		
$900E 36878 (37006)	Auxiliary Color	Sound Volume		Color
$900F 36879 (37007)	Background Color	Reverse 0 on 1 off	Border Color	

Note: The addresses in parentheses are alternate addresses that have the same effect, due to incomplete address decoding. For example, *POKE 37006,15* is equivalent to *POKE 36878,15*, and may be easier to remember.

Appendix L

Device Numbers

Table of second parameter in OPEN. Example: OPEN 5,4 opens file #5 to printer.

0	Keyboard
1	Tape
2	RS-232, usually modem
3	Screen
4	Printer
5	Printer—alternative setting
6	Plotter
8	Disk Drive
9	Disk Drive—alternative
10	Disk Drive—alternative
11	Disk Drive—alternative

Decimal-Hexadecimal Inter-conversion Table

	Low	High		Low	High		Low	High		Low	High
Hex	Dec.	Dec.	Hex	Dec.	Dec.	Hex	Dec.	Dec.	Hex	Dec.	Dec.
$00	0	0	$40	64	16384	$80	128	32768	$C0	192	49152
$01	1	256	$41	65	16640	$81	129	33024	$C1	193	49408
$02	2	512	$42	66	16896	$82	130	33280	$C2	194	49664
$03	3	768	$43	67	17152	$83	131	33536	$C3	195	49920
$04	4	1024	$44	68	17408	$84	132	33792	$C4	196	50176
$05	5	1280	$45	69	17664	$85	133	34048	$C5	197	50432
$06	6	1536	$46	70	17920	$86	134	34304	$C6	198	50688
$07	7	1792	$47	71	18176	$87	135	34560	$C7	199	50944
$08	8	2048	$48	72	18432	$88	136	34816	$C8	200	51200
$09	9	2304	$49	73	18688	$89	137	35072	$C9	201	51456
$0A	10	2560	$4A	74	18944	$8A	138	35328	$CA	202	51712
$0B	11	2816	$4B	75	19200	$8B	139	35584	$CB	203	51968
$0C	12	3072	$4C	76	19456	$8C	140	35840	$CC	204	52224
$0D	13	3328	$4D	77	19712	$8D	141	36096	$CD	205	52480
$0E	14	3584	$4E	78	19968	$8E	142	36352	$CE	206	52736
$0F	15	3840	$4F	79	20224	$8F	143	36608	$CF	207	52992
$10	16	4096	$50	80	20480	$90	144	36864	$D0	208	53248
$11	17	4352	$51	81	20736	$91	145	37120	$D1	209	53504
$12	18	4608	$52	82	20992	$92	146	37376	$D2	210	53760
$13	19	4864	$53	83	21248	$93	147	37632	$D3	211	54016
$14	20	5120	$54	84	21504	$94	148	37888	$D4	212	54272
$15	21	5376	$55	85	21760	$95	149	38144	$D5	213	54528
$16	22	5632	$56	86	22016	$96	150	38400	$D6	214	54784
$17	23	5888	$57	87	22272	$97	151	38656	$D7	215	55040
$18	24	6144	$58	88	22528	$98	152	38912	$D8	216	55296
$19	25	6400	$59	89	22784	$99	153	39168	$D9	217	55552
$1A	26	6656	$5A	90	23040	$9A	154	39424	$DA	218	55808
$1B	27	6912	$5B	91	23296	$9B	155	39680	$DB	219	56064
$1C	28	7168	$5C	92	23552	$9C	156	39936	$DC	220	56320
$1D	29	7424	$5D	93	23808	$9D	157	40192	$DD	221	56576
$1E	30	7680	$5E	94	24064	$9E	158	40448	$DE	222	56832
$1F	31	7936	$5F	95	24320	$9F	159	40704	$DF	223	57088
$20	32	8192	$60	96	24576	$A0	160	40960	$E0	224	57344
$21	33	8448	$61	97	24832	$A1	161	41216	$E1	225	57600
$22	34	8704	$62	98	25088	$A2	162	41472	$E2	226	57856
$23	35	8960	$63	99	25344	$A3	163	41728	$E3	227	58112
$24	36	9216	$64	100	25600	$A4	164	41984	$E4	228	58368
$25	37	9472	$65	101	25856	$A5	165	42240	$E5	229	58624
$26	38	9728	$66	102	26112	$A6	166	42496	$E6	230	58880
$27	39	9984	$67	103	26368	$A7	167	42752	$E7	231	59136
$28	40	10240	$68	104	26624	$A8	168	43008	$E8	232	59392
$29	41	10496	$69	105	26880	$A9	169	43264	$E9	233	59648
$2A	42	10752	$6A	106	27136	$AA	170	43520	$EA	234	59904
$2B	43	11008	$6B	107	27392	$AB	171	43776	$EB	235	60160
$2C	44	11264	$6C	108	27648	$AC	172	44032	$EC	236	60416
$2D	45	11520	$6D	109	27904	$AD	173	44288	$ED	237	60672
$2E	46	11776	$6E	110	28160	$AE	174	44544	$EE	238	60928
$2F	47	12032	$6F	111	28416	$AF	175	44800	$EF	239	61184
$30	48	12288	$70	112	28672	$B0	176	45056	$F0	240	61440
$31	49	12544	$71	113	28928	$B1	177	45312	$F1	241	61696
$32	50	12800	$72	114	29184	$B2	178	45568	$F2	242	61952
$33	51	13056	$73	115	29440	$B3	179	45824	$F3	243	62208
$34	52	13312	$74	116	29696	$B4	180	46080	$F4	244	62464
$35	53	13568	$75	117	29952	$B5	181	46336	$F5	245	62720
$36	54	13824	$76	118	30208	$B6	182	46592	$F6	246	62976
$37	55	14080	$77	119	30464	$B7	183	46848	$F7	247	63232
$38	56	14336	$78	120	30720	$B8	184	47104	$F8	248	63488
$39	57	14592	$79	121	30976	$B9	185	47360	$F9	249	63744
$3A	58	14848	$7A	122	31232	$BA	186	47616	$FA	250	64000
$3B	59	15104	$7B	123	31488	$BB	187	47872	$FB	251	64256
$3C	60	15360	$7C	124	31744	$BC	188	48128	$FC	252	64512
$3D	61	15616	$7D	125	32000	$BD	189	48384	$FD	253	64768
$3E	62	15872	$7E	126	32256	$BE	190	48640	$FE	254	65024
$3F	63	16128	$7F	127	32512	$BF	191	48896	$FF	255	65280

Appendix N

Opcodes in Detail

Table of Opcodes and Their Functions, Hexadecimal Values, Timing, and Processor Flags

Opcode	Description	Flags						
		N	V	B	D	I	Z	C
ADC	Add memory with carry to accumulator	N	V				Z	C
AND	Logical AND memory with accumulator	N					Z	
ASL	Shift memory or accumulator one bit left	N					Z	C
BCC	Branch if carry bit clear							
BCS	Branch if carry bit set							
BEQ	Branch if zero bit set							
BIT	AND with A, storing Z and bits 6 and 7	M7	M6				Z	
BMI	Branch if N (negative) flag set							
BNE	Branch if zero bit clear							
BPL	Branch if N bit is not set							
BRK	Force break to IRQ			1		1		
BVC	Branch on internal overflow bit clear							
BVS	Branch on internal overflow bit set							
CLC	Clear the carry bit							0
CLD	Clear decimal flag (for hex arithmetic)				0			
CLI	Clear interrupt disable flag					0		
CLV	Clear internal overflow flag		0					
CMP	Compare memory to accumulator	N					Z	C
CPX	Compare memory to X register	N					Z	C
CPY	Compare memory to Y register	N					Z	C
DEC	Decrement memory location	N					Z	
DEX	Decrement X register	N					Z	
DEY	Decrement Y register	N					Z	
EOR	Logical exclusive OR memory with A	N					Z	
INC	Increment memory location	N					Z	
INX	Increment X register	N					Z	
INY	Increment Y register	N					Z	
JMP	Jump to new address							
JSR	Jump to new address, saving return							
LDA	Load accumulator from memory	N					Z	
LDX	Load X register from memory	N					Z	
LDY	Load Y register from memory	N					Z	
LSR	Shift memory or accumulator one bit right	0					Z	C
NOP	No operation							
ORA	Logical inclusive OR memory with A	N					Z	
PHA	Push accumulator onto stack							
PHP	Push processor status flags onto stack							
PLA	Pull stack into accumulator	N					Z	
PLP	Pull stack into processor status flags	N	V	B	D	I	Z	C
ROL	Rotate memory or A one bit left, inc. C	N					Z	C
ROR	Rotate memory or A one bit right, inc. C	N					Z	C
RTI	Return from interrupt	N	V	B	D	I	Z	C
RTS	Return from subroutine called by JSR							
SBC	Subtract memory and C-complement from A	N	V				Z	C
SEC	Set the carry bit							1
SED	Set the decimal flag (for BCD arithmetic)				1			
SEI	Set the interrupt disable flag					1		
STA	Store accumulator into memory							
STX	Store X into memory							
STY	Store Y into memory							
TAX	Transfer accumulator to X register	N					Z	
TAY	Transfer accumulator to Y register	N					Z	
TSX	Transfer stack pointer to X register	N					Z	
TXA	Transfer X register to A	N					Z	
TXS	Transfer X register to stack pointer							
TYA	Transfer Y register to A	N					Z	

Abs	Abs,X	Abs,Y	Zer	Zer,X	Zer,Y	Implied	Immed	Rel	Acc	(Ind,X)	(Ind),Y	Ind	
6D 4	7D*4	79*4	65 3	75 4			69 2			61 6	71*5		ADC
2D 4	3D*4	39*4	25 3	35 4			29 2			21 6	31*5		AND
0E 6	1E 7		06 5	16 6					0A 2				ASL
								90[2] 2					BCC
								B0[2] 2					BCS
								F0[2] 2					BEQ
2C 4			24 3										BI T
								30[2] 2					BMI
								D0[2] 2					BNE
								10[2] 2					BPL
						00 7							BRK
								50[2] 2					BVC
								70[2] 2					BVS
						18 2							CLC
						D8 2							CLD
						58 2							CLI
						B8 2							CLV
CD 4	DD*4	D9*4	C5 3	D5 4			C9 2			C1 6	D1*5		CMP
EC 4			E4 3				E0 2						CPX
CC 4			C4 3				C0 2						CPY
CE 6	DE 7		C6 5	D6 6									DEC
						CA 2							DEX
						88 2							DEY
4D 4	5D*4	59*4	45 3	55 4			49 2			41 6	51*5		EOR
EE 6	FE 7		E6 5	F6 6									I NC
						E8 2							I NX
						C8 2							I NY
4C 3												6C 5	JMP
20 6													JSR
AD 4	BD*4	B9*4	A5 3	B5 4			A9 2			A1 6	B1*5		LDA
AE 4		BE*4	A6 3		B6 4		A2 2						LDX
AC 4	BC*4		A4 3	B4 4			A0 2						LDY
4E 6	5E 7		46 5	56 6					4A 2				LSR
						EA 2							NOP
0D 4	1D*4	19*4	05 3	15 4			09 2			01 6	11 5		ORA
						48 3							PHA
						08 3							PHP
						68 4							PLA
						28 4							PLP
2E 6	3E 7		26 5	36 6					2A 2				ROL
6E 6	7E 7		66 5	76 6					6A 2				ROR
						40 6							RTI
						60 6							RTS
ED 4	FD*4	F9*4	E5 3	F5 4			E9 2			E1 6	F1*5		SBC
						38 2							SEC
						F8 2							SED
						78 2							SEI
8D 4	9D 5	99 5	85 3	95 4						81 6	91 6		STA
8E 4			86 3		96 4								STX
8C 4			84 3	94 4									STY
						AA 2							TAX
						A8 2							TAY
						BA 2							TSX
						8A 2							TXA
						9A 2							TXS
						98 2							TYA

* +1 if index crosses page
[2] +1 if branch is taken,
+1 more if page crossed

Appendix O

Table of 6502 Opcodes

Opcode Low Nybble

Opcode High Nybble

	0	1	2	4	5	6	8	9	A	C	D	E
0	BRK	ORA (Ind,X)			ORA Zer	ASL Zer	PHP	ORA Imm	ASL A		ORA Abs	ASL Abs
1	BPL	ORA (Ind),Y			ORA Zer,X	ASL Zer,X	CLC	ORA Abs,Y			ORA Abs,X	ASL Abs,X
2	JSR	AND (Ind,X)		BIT Zer	AND Zer	ROL Zer	PLP	AND Imm	ROL A	BIT Abs	AND Abs	ROL Abs
3	BMI	AND (ind),Y			AND Zer,X	ROL Zer,X	SEC	AND Abs,Y			AND Abs,X	ROL Abs,X
4	RTI	EOR (Ind,X)			EOR Zer	LSR Zer	PHA	EOR Imm	LSR A	JMP Abs	EOR Abs	LSR Abs
5	BVC	EOR (Ind),Y			EOR Zer,X	LSR Zer,X	CLI	EOR Abs,Y			EOR Abs,X	LSR Abs,X
6	RTS	ADC (Ind,X)			ADC Zer	ROR Zer	PLA	ADC Imm	ROR A	JMP Ind	ADC Abs	ROR Abs
7	BVS	ADC (Ind),Y			ADC Zer,X	ROR Zer,X	SEI	ADC Abs,Y			ADC Abs,X	ROR Abs,X
8		STA (Ind,X)		STY Zer	STA Zer	STX Zer	DEY		TXA	STY Abs	STA Abs	STX Abs
9	BCC	STA (Ind),Y		STY Zer,X	STA Zer,X	STX Zer,Y	TYA	STA Abs,Y	TXS		STA Abs,X	
A	LDY Imm	LDA (Ind,X)	LDX Imm	LDY Zer	LDA Zer	LDX Zer	TAY	LDA Imm	TAX	LDY Abs	LDA Abs	LDX Abs
B	BCS	LDA (Ind),Y		LDY Zer,X	LDA Zer,X	LDX Zer,Y	CLV	LDA Abs,Y	TSX	LDY Abs,X	LDA Abs,X	LDX Abs,Y
C	CPY Imm	CMP (Ind,X)		CPY Zer	CMP Zer	DEC Zer	INY	CMP Imm	DEX	CPY Abs	CMP Abs	DEC Abs
D	BNE	CMP (Ind),Y			CMP Zer,X	DEC Zer,X	CLD	CMP Abs,Y			CMP Abs,X	DEC Abs,X
E	CPX Imm	SBC (Ind,X)		CPX Zer	SBC Zer	INC Zer	INX	SBC Imm	NOP	CPX Abs	SBC Abs	INC Abs
F	BEQ	SBC (Ind),Y			SBC Zer,X	INC Zer,X	SED	SBC Abs,Y			SBC Abs,X	INC Abs,X

6502 Quasi-Opcodes

6502 Quasi-Opcodes

Instruction	Abs	Abs,X	Abs,Y	Zer	Zer,X	Zer,Y	(Ind,X)	(Ind),Y	Imm
ASO (ASL,ORA)	**0F**	**1F**	1B	**07**	**17**		03	13	**0B**
RLA (ROL,AND)	**2F**	**3F**	3B	**27**	**37**		23	33	**2B**
LSE (LSR,EOR)	**4F**	**5F**	5B	**47**	**57**		43	53	**4B**
RRA (ROR,ADC)	**6F**	**7F**	7B	**67**	**77**		63	73	**6B**
AXS (STX,STA)	**8F**			**87**		97	83		
LAX (LDX,LDA)	**AF**		BF	**A7**	**B7**		A3	B3	
DCM (DEC,CMP)	**CF**	**DF**	DB	C7	**D7**		C3	D3	
INS (INC,SBC)	**EF**	**FF**	FB	E7	**F7**		E3	F3	
ALR (LSR,EOR)									**4B**
ARR (ROR,ADC)									**6B**
XAA (TXA,AND)									8B
OAL (TAX,LDA)									**AB**
SAX (DEX,CMP)									**CB**
NOP	**1A, 3A, 5A, 7A, DA, FA**								
SKB	**80, 82, C2, E2, 04, 14, 34, 44, 54, 64, 74, D4, F4**								
SKW	**0C, 1C, 3C, 5C, 7C, DC, FC**								

ASO	ASL then ORA the result with the accumulator
RLA	ROL then AND the result with the accumulator
LSE	LSR then EOR the result with the accumulator
RRA	ROR then ADC the result from the accumulator
AXS	Store the result of A AND X
LAX	LDA and LDX with the same data
DCM	DEC memory and CMP the result with the accumulator
INS	INC memory then SBC the result with the accumulator
ALR	AND the accumulator with data and LSR the result
ARR	AND the accumulator with data and ROR the result
XAA	Store X AND data in the accumulator
OAL	ORA the accumulator with #$EE, AND the result with data, then TAX
SAX	SBC data from A AND X and store result in X
NOP	No operation
SKB	Skip byte (that is, branch of +1)
SKW	Skip word of two bytes (that is, branch of +2)

A number of bit patterns which do not appear in Appendices N and O will still be interpreted by the 6502 as opcodes. These commands are not part of the 6502's specification. Types X3, X7 XB, and XF (and most of X2) aren't defined. Generally, these quasi-opcodes arise from the processor attempting to execute two instructions simultaneously.

There are many regularities in these results. Codes ending in bits 11 execute two standard instructions ending with bits 01 and 10, simultaneously; if the addressing modes of the instructions don't match, the higher may be executed first. Those quasi-opcodes shown in the table in boldface seem likely to be more reliable than the others.

While there are no guarantees that these opcodes will continue to work with all revisions of the 6502, it is a fact that some published software containing these codes has given no problems. All 6502s seem to be produced from the same masks, as is shown by the well-known bug in indirect JMP, where JMP ($01FF) takes its two-byte address from $01FF and $0100.

Besides providing some programming shortcuts, quasi-opcodes allow some measure of concealment from disassembly, as no standard disassembler program will be able to interpret them. For example,

```
033C  ASO  $0342  ;Shift Left contents of $0324
033F  DCM  $0345  ;Decrement contents of $0345
0342  ML program
```

shows on *VICMON* as

```
033C  0F  42  03  CF  45
0341  03  XX  ??  ??  YY
```

where XX, ??, and YY are parts of the ML program. Disassembly starting at 033C will produce at least ten bytes of garbage. However, the program will run properly, but only once. You must compensate for the first two instructions, which halve the contents of $0345 and decrement the contents of $0345. If you set up a loop to change some other portion of the ML—for example, by EORing it with some set values—the whole of a large section of RAM ML can be made hard to decipher.

Interconverting VIC-20, Commodore 64, and CBM Programs

"Conversion" is a deceptively simple word, hiding the reality that one machine's programs must often be rewritten for use on another. First you'll see how to transfer programs between machines. Then you'll see how to convert them to run in their new environments. Generally, these remarks apply only to BASIC. ML programs usually have to be rewritten.

LOADing Other Programs into the VIC-20

Commodore 64 programs. Commodore 64 disk programs should load without difficulty into VIC-20. However, Commodore 64 tape programs may give problems, since recording speeds differ even though the format is the same. If loading is unsuccessful, try saving the original as a file with OPEN1,1: CMD1: LIST: PRINT#1: CLOSE1 and using MERGE from Chapter 6 to read it into VIC-20. This writes the program in smaller chunks, so loading is easier. If this fails, loading into a CBM first (see below), and then into the VIC, is likely to work. Alternately, the program could even be transferred by modem.

PET/CBM programs. PET/CBM disk programs should load into VIC-20, but only if formatted with CBM's 4040 disk drive. Tape should be trouble-free; if there are LOAD errors, try using the same recorder with both CBM and VIC to be sure the head alignment isn't a factor.

Note that the earliest (tiny keyboard) PETs don't operate in quite the same manner; they have an extra zero byte at the start which usually scrambles the first line after loading into VIC (the rest of the program is OK).

To load a program from one of these very early PETs into the VIC, add 3K expansion and force-load with LOAD "NAME",1,1. Alternately, you can add a redundant first line to the PET program and delete the meaningless line number at the start when it's loaded into VIC. You can also load the program into a newer CBM and save it, giving a VIC-loadable program.

Loading VIC Programs into Other Computers

Loading VIC-20 programs into the 64. This is no problem with disks. However, tape may be unsuccessful, because of timing differences. Use the same cures as you would when loading 64 programs into the VIC.

Loading VIC-20 programs into PET/CBMs. CBMs didn't need, and don't have, VIC's relocating LOAD feature. They cannot recognize forced-LOAD programs (that is, those saved with SAVE "NAME",1,1) as programs at all. CBMs expect all programs to start at $0400.

If the VIC-20 program can run with a 3K memory expander, and no other extra RAM (and if you have access to such an expander), save the program with the extra 3K in place. Then either tape or disk will load into a CBM and be ready to run. (The CBM must have a 4040 disk drive if disks are used.)

If 3K expansion isn't appropriate for the program, loading into CBM is still easy. First, load the program into VIC and enter PRINT PEEK(830) to find the start address as recorded on tape, and note this number. With a disk program, find the LOAD address by reading the first two bytes. The relevant byte is usually 16 or 18. If it's 4, the program will load as it stands; it was saved with 3K expansion in place. Then load the program into CBM, type the line 0 REM and enter POKE 1025,7: POKE 1026,16 (or 18, or whatever number you got from the PEEK(830)). Now delete line 0 by typing 0 and RETURN. The POKEs alter the link address after the tiny program 0 REM, and the built-in editor moves the program down to $0400. The result can be saved and reloaded without trouble with the CBM, and it will load into VIC successfully.

Another method, which keeps BASIC in the same part of RAM as in VIC, is to use:

```
POKE 256*16,0: POKE 41,16: CLR
```

before LOADing into CBM. This moves the start of BASIC to coincide with VIC's start of BASIC. (If the PEEK(830) didn't give you 16, replace the 16 in the line above with the correct value.)

These methods work with all PET/CBMs.

Program Conversion

Programs which are pure BASIC, even for non-CBM computers, can often be converted to run on the VIC-20. Difficulties are likely, though, particularly if a program is long, since the VIC's memory may not be large enough. You may also have problems if the program assumes a 40-column screen width or if much disk or tape access is needed. Sometimes a 40-column screen utility will allow the VIC to run 40-column software; alternately, the screen layout can be cut down to VIC's size.

VIC can perform any Commodore disk operation, although CBM BASIC 4.0 requires translation into the lower-level version, since it includes disk commands not available on the VIC or 64. Other computers' disk operations may well be rewritten to operate with the VIC-20.

There are often other subtle differences between computers, too. For instance, some interpret logical "true" as 1, rather than −1 as with CBM, so logical operations may work incorrectly. And of course some commands (like PRINT USING) are simply missing from CBM BASIC.

CBM BASICs are all more or less transportable between machines. However, the earliest PETs and latest CBMs are a bit different from VIC in several small ways—GO TO isn't allowed in one, for example, and DS is a reserved variable in the other. Pure BASIC (without SYS, PEEK, POKE, WAIT, or USR) is compatible to a very large extent; apart from memory size, only the 22-column screen of VIC is a big problem, with related features like the bug in INPUT "LONG PROMPT";X$, differences with POS, SPC, and TAB, and cursor movements which may scroll the screen.

You can expect that calculation programs and programs which print out results will work with little change; so will programs written without PEEKs or POKEs. With luck, programs which use the built-in graphics set may convert quite easily. A checkers program, with complicated logic and a simple board display, may need work on the display but can be expected to run properly if the graphics are right.

POKE, PEEK, SYS, WAIT, and USR. These are the problem areas when converting programs; very little ML is transportable between machines. Some ML has an exact equivalent in each CBM machine, for example, screen POKEs and POKEs into the keyboard buffer. But other ML is machine-specific. For example, sprites in the 64 have no equivalent in other CBM machines.

If you're lucky, and the BASIC program has many REMarks, conversion can be a simple matter of looking up the location in one memory map and finding the equivalent in another. Disabling the STOP key and manipulating the keyboard buffer are examples. You may be able to delete some commands; disabling STOP isn't very important. In addition, you may be able to replace some PEEKs amd POKEs. For instance, VIC's POKE 198,0 has a BASIC equivalent, FOR J=1 TO 10: GET X$: NEXT, which clears the keyboard buffer. CBM's POKE 59468,14 to switch to lowercase is replaceable by PRINT CHR$(14) on the VIC and 64.

Generally, POKEs, PEEKs, and WAITs involving locations 140–250 are likely to apply to the screen or keyboard. Low memory values often alter BASIC pointers. Most low RAM locations have the same sort of effect with VIC and the 64. CBM is rather different, though as a rule BASIC 2's usage of locations up to 120 or so are just three addresses less than VIC/64 values (a POKE to location 41 in a PET/CBM has the same effect as a POKE to location 44 on a VIC or 64).

SYS commands can be converted only if you have ML knowledge. A routine may call some Kernal addresses, and be usable unchanged; more likely, disassembly will show up a few addresses which have to be changed. Without ML knowledge you can't be sure what ML POKEd to RAM does. The only exception is published routines, as in this book, where an alternative routine can be typed in to perform the same function.

POKE commands are usually the most difficult to convert, because they can change the whole program configuration. Screen POKEs and the color RAM, graphics definitions and sound, interface chip manipulations, and uses of multicolor mode illustrate this sort of thing. PEEKs to interface chips (to read joysticks, for example) are also tricky, but they can be routinized more easily in view of the narrower purposes they serve.

The following table gives relevant POKE and PEEK locations for a variety of functions. It cannot possibly be exhaustive, but at least it will help you identify the purpose of those mysterious POKEs in other people's programs.

Equivalent Memory Locations

	VIC-20	64	CBM BASIC 2 & 4
Screen Memory	7680–8185 (unexpanded) 4096–4591 (with 8K or more expansion)	1024–2023	32768–33767 (40 column) 32768–34767 (80 column)
Color Memory	37888–38393 (unexpanded) 38400–38905 (with 8K or more expansion)	55296–56295	—
Character ROM	32768–36863	53248–57343	—
Registers to Control Character Set Location	36866, 36867, 36869	53272	—
Sound Registers	36874–36878	54272–54300	59464, 59466
Joystick Registers	37151, 37152	56320, 56321	—
Light Pen Registers	36870, 36871	53267, 53268	—
Paddle Registers	36872, 36873	54297, 54298	—
Interface Chip Registers	VIA1 37136–37151 VIA2 37152–37167	CIA1 56320–56335 CIA2 56576–56591	PIA1 59408–59411 PIA2 59424–59427 VIA 59456–59471
Start-of-BASIC Pointer	43,44	43,44	40,41
Top-of-BASIC Pointer	55,56	55,56	52,53

Index

Commodore's VIC-20 is one of the most popular computers ever produced. It has astonishing capabilities for a machine of its price and size, but there has never been a book which thoroughly explains how to get the most out of your VIC.

Until now.

Written by noted Commodore authority Raeto Collin West, *Programming the VIC* is the definitive work on the VIC-20. It's packed with information on every aspect of the VIC-20, from elementary BASIC, sound, and graphics techniques to advanced hardware applications and machine language programming. Here's just a sample of what you'll find:

- Dozens of programming techniques and tricks for you to experiment with.
- Detailed descriptions of every BASIC command in VIC's vocabulary.
- Thorough discussions, including numerous program examples, of BASIC and machine language programming techniques.
- A comprehensive guide to VIC sound and graphics.
- Complete explanations of tape and disk operation.
- An annotated list of 6502 opcodes.
- A thorough mapping of the VIC-20's ROM.
- Clear explanations of how the VIC user port works, with suggestions on how to use it in your own programming.
- A practical guide to selecting and using printers, plotters, and modems.

This book addresses virtually every programming situation that the VIC user is likely to encounter. How can you use interrupt-driven routines? To DIM, or not to DIM? Can you back up ROM cartridges or add commands to VIC BASIC? Here are answers to these questions and many, many more. Its 17 chapters contain hundreds of examples, each designed to clarify or illustrate a particular concept or technique. There are also dozens of complete programs, thoroughly tested and ready to type in and run, as well as hundreds of short routines that you can incorporate into programs of your own.

For beginning programmers, *Programming the VIC* serves as a comprehensive introduction. More advanced programmers will find it an instructive tutorial and valuable reference. But every VIC owner, regardless of experience level, will find it to be the definitive, indispensable VIC resource book—one that will be referred to again and again.

ISBN 0-942386-52-3