Programming the COMMODORE 64

The Definitive Guide

Raeto Collin West

The encyclopedic reference guide to the Commodore 64 computer.

A COMPUTE! Books Publication $19.95

Programming the COMMODORE 64

The Definitive Guide

Raeto Collin West

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

Printed in the United States of America

ISBN 0-942386-50-7

10 9 8 7 6 5 4 3 2 1

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809, is one of the ABC Publishing Companies and is not associated with any manufacturer of personal computers.

Contents

Foreword

Programming the Commodore 64 is *the* definitive guide. It covers virtually every aspect of the Commodore 64, from simple BASIC commands to complex machine language techniques. Every explanation is written in depth and with clarity. The result is a comprehensive, easy-to-understand reference that thoroughly explains the 64's capabilities.

If you program in BASIC, you'll find the detailed, annotated listings of every BASIC command a tremendous aid. And if you're writing in machine language, you'll be especially interested in the ROM maps and listings of Kernal routines. No matter what your experience with the Commodore 64, you'll find the numerous program examples both useful and informative.

Beginning with a brief introduction to Commodore BASIC and BASIC programming, this book goes on to discussions of more advanced programming, including sophisticated machine language techniques. Specialized sound and graphics applications are included as well. Complete chapters are devoted to disk and tape storage, 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 *Programming the Commodore 64*. You'll discover new information on each page.

The author has included scores of practical programs to demonstrate many of the techniques discussed. To help you enter the programs correctly, we've included "The Automatic Proofreader," an error-checking program. As a convenience, COMPUTE! Publications is also making available for purchase a disk that contains most of the significant programs from this book. To order a disk, use the coupon in the back or call 800-334-0868 (in North Carolina 919-275-9809).

This is the first book to thoroughly cover every aspect of 64 programming. It's certain to become an indispensable work for any Commodore 64 owner.

Chapter 1

About This Book

- Introduction
- Programming Your 64
- Conventional Terms
- Acknowledgments

Chapter 1

About This Book

Introduction

The two main objectives of this book are to teach competent programming on the Commodore 64, and to provide a comprehensive reference book for people wanting quick, accurate answers to questions about the 64. These two goals are difficult enough to achieve. For example, while virtually everyone begins with BASIC and progresses to machine language (ML), it is often desirable to use both ML and BASIC in examples, which means comparative newcomers to the 64 find themselves skipping sections of temporarily difficult text. It is practically impossible to arrange the material so that everything falls into a natural sequence for all readers, because many of the chapter headings themselves can't be understood properly without some knowledge of the machine's structure.

This book explains BASIC and ML in sequence from simple to complex, ending with a chapter on mixing ML with BASIC, an efficient 64 programming method. These chapters are interspersed with machine organization details and are followed by in-depth examinations of major topics—sound, graphics, tape, and so on.

For these reasons, the text contains two kinds of programs. First, there are very short routines, intended to be typed in quickly (and therefore with little chance of error). Second, there are longer, more practical programs, which use graphics, sound, tape, disks, and all the other features of the 64. The shorter, example programs cover how BASIC commands are used, how special features work (notably the VIC and SID chips), and how to use ML. Many useful routines are included, and *these can be used successfully without any knowledge of their internal operation.* Many readers will thus be able to acquire BASIC and ML experience painlessly as they use this book.

Programming the Commodore 64 is one of a set of three books, which also includes *Programming the VIC* and the earlier *Programming the PET/CBM*. The books have been written entirely independently of Commodore; they contain criticisms, comments, hints, and a great deal of otherwise unpublished information.

Programming Your 64

The 64 is one of the world's most popular microcomputers; like all big-selling computers it is both rather inexpensive and rather easy to use. But many owners have found that however easy using other people's programs may be, writing their own programs for the 64 is not so simple. Reliable information has been difficult to find—*until now.*

Programming the Commodore 64 shows you how to plan and write BASIC programs, how to move programs from tape to disk to save time, and how to play music while a program runs. It also explains how to program the function keys, round numbers to two places, use high-resolution graphics, design and store your own graphics characters, display a screenful of sprites, make the screen scroll smoothly, and save sections of memory to tape. This is only a small sample of problems which have puzzled many 64 users. All these and more are comprehensively discussed here.

The 64 is at times a difficult machine to program, in spite of what you may have heard. But programming is easier if you have a good overall understanding of the machine, and *Programming the Commodore 64* attempts to generalize results rather than give isolated facts. For example, the way the VIC chip determines what kind of graphics to display is crucially important to understanding the system, and there is a handy table to illustrate this.

This book is just above the introductory level. There's not enough room to cover the elementary topics (which are often better learned directly at the keyboard or from a friend who knows the machine) and still provide the information you need on advanced topics. But prior knowledge of Commodore systems is not essential, so anyone with intelligence and some microcomputer experience ought to be able to grasp the fundamentals of the 64 fairly easily.

Several versions of the 64 exist: the repackaged SX-64, the PET 64 (which has no sound chip and only monochrome graphics), and 64s with slightly different ROMs. As Chapter 8 explains, these computers run software similarly, but not exactly alike. Most of the book is applicable to all these machines, but emphasis is on the more common models.

The programs have been tested and will almost always work without difficulty. If there are problems, a program may have been entered incorrectly, or the *soft* nature of the 64 may be to blame—there are many special memory locations in the 64, any one of which can cause odd results if altered. The first thing to do when a program will not run properly is to check the program carefully. If that doesn't work, it's usually easiest to *save the program,* turn the 64 off, then back on, reload the program, and try again. Some of the programs in this book use the "Automatic Proofreader," Appendix C, which allows you to quickly and easily check each line you have entered for accuracy.

Information with the widest appeal—BASIC, graphics, music, and full use of the 64's RAM capacity—is documented fully. However, minority interests are not excluded. *Programming the Commodore 64* doesn't gloss over difficulties and evade important topics. Many people have gone to great lengths to check the information in this book, for it is intended to be reliable. Nevertheless, there are certain to be errors, and for any resulting inconvenience or bafflement, we apologize in advance.

Conventional Terms

The special Commodore logo key (located at the bottom left of the 64's keyboard) will be called the *Commodore key*. Program listings and short examples, like SYS 64738, will usually be in capitals to mimic their appearance on the screen and printers of the 64 in its uppercase mode. This is the mode the 64 is in when you turn the power on. Text can, of course, appear in lowercase mode, usually after pressing SHIFT–Commodore key. (Don't type the hyphen; just hold down the SHIFT key and press the Commodore key.) In either case, programs are mostly entered using unSHIFTed keys. Named keys (CLR/HOME, SHIFT, CTRL, f1, and so on) are generally referred to as they appear on the keyboard of the 64.

Some BASIC listings have been made with a program which prints the screen editing commands, colors, and other special characters, such as {HOME}, {CYN}, {F1}, and so on, in place of the less readable 64 characters. With apologies to people who can spell, the book uses the spelling *Kernal* for the ROM routines which handle the screen, keyboard, and input/output devices (this is the spelling Commodore uses). This book also uses *ML* as a handy abbreviation for *machine language*.

Hexadecimal numbers are represented here with a leading dollar sign ($). If you don't understand hexadecimal notation, read Chapter 6. In accordance with 6502 convention, the number symbol (#) indicates a value, not an address, so that LDA #$00 denotes the command to load the accumulator with the number 0, while LDA $00 loads the accumulator with the value in location 0.

Many 64 BASIC programs begin with a few commands to change color from the usual light blue characters on a dark blue background. The following line sets a green background, white border, and black characters:

```
POKE 53281,5: POKE 53280,1: POKE 646,0
```

Some of the demonstration programs include this line; others assume that CTRL-WHITE or some similar command has already been used to improve clarity.

Acknowledgments

Several chapters, notably those on sound and graphics, are partially the work of Marcus West. Additional hardware information has been provided by Rod Eva of Y2 Computers, Watford, U.K. COMPUTE! Publications, Inc. in the U.S., TPUG (Toronto PET Users Group) of Canada, and ICPUG (Independent Commodore Products Users Group) of the U.K. have provided information.

Chapter 2

Getting to Know the 64

- The 64's Connectors
- The Keyboard
- Editing BASIC on the 64

Chapter 2

Getting to Know the 64

Commodore has been making computers for a decade or so, and became a household word with the introduction of the low-priced VIC-20, followed more recently in the early 1980s by the Commodore 64. Both machines proved remarkably successful, far outselling other Commodore Business Machine (CBM) computers in volume. CBM's earlier experience enabled a range of appropriately priced peripherals (tape drives, disk drives, printers, modems) to be produced at about the same time. All CBM machines have strong resemblances, and straightforward BASIC programs which run on one CBM machine are likely to run on others, too. But programs of any complexity will generally run only on the machine for which they were written.

The 64's Connectors

The back panel of the 64 has these features (left to right viewed from the back):

The cartridge socket is where ROM cartridges are plugged in. (Only those designed for the 64 will work.) Be sure to insert cartridges correctly. Chapter 5 has further information. Some other peripherals are designed to plug into this socket, including the CP/M cartridge and the Magic Voice speech module.

The channel selector switch selects channel 3 or 4 on TVs in the U.S. (Commodore 64s for PAL-type TVs in Europe and elsewhere don't have this switch.)

The RF modulator output jack provides a combined video and audio signal that can be used directly by a standard television. The RF (radio frequency) modulator inside the 64 performs the function of a tiny TV station, converting (modulating) the computer's video and audio signals into a signal that can be received via the TV's antenna connectors.

The audio-video socket provides high-quality output from the 64's sound and video chips, for use by video monitors or even hi-fi sound equipment. This is a DIN socket. Earlier 64s have five-pin sockets; some later models may have eight-pin sockets. A cable for the five-pin socket will work with the eight-pin connector, but not vice versa.

The serial port is a modified version of the parallel IEEE input/output port of the earlier CBM computers. The signal format on the port, unique to Commodore, is designed for use with VIC and 64 devices (disk drives, printers) but is not directly compatible with much else. (This should not be confused with the user port, which provides RS-232 serial communication.) Chapters 15 (disk drives) and 17 (printers) have more information.

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

The user port is compatible in size and function with VIC equipment. It is designed to allow communication with the outside world with a Commodore modem, or for other purposes, including the control of electronic instruments. Chapter 5 explains how it's programmed.

The side panel of the 64 has these features, again from left to right:

Control ports 1 and 2 allow one or two joysticks to be connected to the 64. The joystick nine-pin D-connector is standard. Port 2 is often preferred, as it's a little easier to program. Pairs of paddles (rotary controllers) can be plugged into either port. A light pen can be plugged into port 1. Chapter 16 has full programming information on these and other controllers.

The power switch, as you might have guessed, is where you turn the power on.

The power input socket is where you plug in the connector from the power supply unit, which requires standard household current.

The Keyboard

Chapter 6 discusses the keyboard in depth. Here, we'll briefly survey the keys and their functions as they act just after the computer is turned on, before they are modified by a program (this is called their *default* arrangement). Alphabetic, numeric, and symbol keys appear as the keytop legend implies, subject to the changes mentioned below.

SHIFT selects uppercase lettering or the right-hand graphics set, depending on the character set in use.

Commodore key accesses the left-hand graphics set or, with the color keys, selects one of eight additional colors not named on the standard 64 keyboard. Where these don't apply, the Commodore key acts as an alternative SHIFT key.

SHIFT-Commodore key (SHIFT and Commodore key together) changes the display from uppercase, with the full keytop graphics set, to upper- and lowercase lettering, plus the left-hand graphics set.

CTRL (Control) acts with the color keys and reverse keys to change the color of the characters or to turn reverse video printing on and off. CTRL-A through CTRL-Z generate CHR$(1) through CHR$(26), acting as a conventional control key—see Chapter 3. CTRL also slows screen scrolling; this is useful when listing a program.

Function keys (f1 through f8) display nothing. In BASIC, GET is the easiest way to detect them. (See Chapter 6 for program information.)

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

SHIFT-RUN/STOP (SHIFT and RUN/STOP together) loads, then runs the next BASIC program on tape. Note that almost any key followed by SHIFT-RUN/STOP runs the BASIC program in memory. A normal *disk* LOAD command followed by a colon, then SHIFT-RUN/STOP will load the specified program from disk, then run it.

RUN/STOP-RESTORE acts like a panic button; the system returns to its normal state, retaining the BASIC program in memory. Chapter 6 explains both RESTORE and RUN/STOP in detail.

CLR/HOME places the cursor at the top left of the screen; **SHIFT-CLR/HOME** also erases the screen, leaving it blank, with the cursor flashing in the home position.

INST/DEL (insert/delete) is part of the screen editing system—the set of operations which allow the alteration of any part of the screen. Both keys repeat, although INST has little effect if it isn't followed by characters on its BASIC line. The screen editing is powerful and easy to use, despite having a few small quirks in quote mode. To delete characters to the left of the flashing cursor, press the key unSHIFTed; to insert characters to the right of the cursor, press SHIFT-INST/DEL.

CRSR keys (cursor keys) move the flashing cursor in the directions printed on the keytops. UnSHIFTed, the cursor will move in the direction printed below the letters CRSR on the key. SHIFTed, the cursor will move in the direction shown above CRSR. These keys automatically repeat to save time. Movement down the screen eventually causes the text on the screen to scroll up.

RETURN is the key used by the 64 to signal that information on the screen is ready for processing. For example, if you type:

```
PRINT "HELLO"
```

on the screen, pressing RETURN causes HELLO to appear. Similarly, RETURN signals that data typed at the keyboard in response to an INPUT statement is ready for the 64 to process.

SHIFT-RETURN or **Commodore key-RETURN** moves the cursor to the next BASIC line (not the same as the next screen line—see below), but *without* causing the 64 to take in the information. (For example, if you begin to correct a line of BASIC, but change your mind, SHIFT-RETURN leaves the line as it was.)

Quotation marks (SHIFT-2) are important in BASIC; quotation marks designate the start or end of a string (a group of characters) that you may want to print, assign to a variable, or manipulate in some other way. When in quote mode (more on this later), several special characters which follow double-quotes are stored as reversed characters. See SHIFT-RETURN above.

The space bar repeats if held down. Chapter 6 gives full information.

Editing BASIC on the 64

Everything entered into the 64 is treated as BASIC, unless some special language has been loaded into memory. The 64 operates in several modes, which are described below.

Direct mode. We've 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 the following line, followed by RETURN:

```
10 PRINT "HELLO"
```

Apparently, nothing happens. In fact, however, the line is stored in the 64's memory as a line of a BASIC program; any line beginning with a number (up to 63999) is interpreted as a program line and stored. This is called *program* mode or *deferred* mode. LIST displays the BASIC program in memory; RUN executes it. If you run the above example, you should see HELLO on the screen.

Quote mode. In quote mode, the 64's special characters are stored for future use. Quote mode enables you to use the 64's powerful screen control features, including cursor moves and screen clearing, from within programs.

BASIC Terms

Variables are algebraic in nature; X=10: PRINT X prints the number 10 on the screen, since the variable, X, has been given the value 10. The value can be altered, so X is referred to as a variable. The next chapter explains this more fully.

Keywords are the commands recognized by BASIC; PRINT is one. All the keywords are listed, with examples, in the next chapter. Note that most keywords can be entered in a shortened form. For example, PRINT can be abbreviated with a question mark.

10 ?

followed by LIST reads as:

10 PRINT

Device numbers allow the 64 to communicate with external hardware devices, selectively. The tape unit, for example, is device 1. Tape is the default device, which means that if no other device is specifically requested by number, the tape unit will be selected. Data can also be written to or read from other devices, but there are restrictions; you can read and write to tape, but only write to a printer, for example. The commands for reading and writing also require a reference number, which is called a *logical file number*. Sometimes these commands will need a *secondary address* as well. For details, see OPEN and CLOSE in the reference section in the next chapter.

The 64 has 25 screen lines, each with 40 characters. But BASIC can link together the information on two screen lines into one program line—hence, the distinction between screen lines and BASIC lines (sometimes called *physical* and *logical* lines, respectively). Try typing PRINT, followed by quotes and several lines of asterisks (or other characters). You'll find that the third and subsequent lines aren't included as part of the program line.

Finally, BASIC has built-in *error messages*, which are designed to help with debugging (removing mistakes from) your programs. The final section of Chapter 3 lists the error messages alphabetically with explanations.

Chapter 3

BASIC Reference Guide

- BASIC Syntax
- BASIC Keyword Dictionary
- BASIC Error Message Dictionary

Chapter 3

BASIC Reference Guide

BASIC Syntax

BASIC is now the most popular language for personal computers. It's easy to write, test, and edit BASIC, and simple programs can be written by people with very little computing experience, which is exciting and encouraging. BASIC is sometimes described as being like English, but the resemblance is tenuous.

Almost every machine has its own version of BASIC. As a result, expressions work differently on different machines. The information in this book applies to the version of BASIC contained in the 64.

Numbers and Literals

Numbers and literal character strings are constants, as opposed to variables. Examples of numbers are 0, 2.3 E−7, 1234.75, and −744; examples of literals are "HELLO", "ABC123", and "%!£/", where the quotation marks are *delimiters* (not part of the literal). The rules which determine the validity of these forms are complex; generally, numbers are valid if they contain 0–9, +, −, E, or a decimal point in legal combinations. Thus, 1.2.3 is not valid (only one decimal point may be used); nor is 2EE3 (only one E is permitted). But either 0E or a single decimal point is accepted as 0.

Exponential notation (using E) may be unfamiliar to some; the number following E is the number of positions left or right that the decimal point must be moved to produce a number in ordinary notation. (1.7E3 means 1.7 × 10 to the third power, or 1.7 × 1000, which is 1700. The form 9.45E−2 is simply another notation for the number .0945.) Be careful when typing these numbers in, because SHIFT-E is not accepted. Values outside the ranges .01 to 999,999,999 and −.01 to −999,999,999 are printed in exponential form.

Strings can contain any of the Commodore 64's ASCII characters; all characters can be accessed with the CHR$ function, including quotes, CHR$(34), and RETURN, CHR$(13). The maximum length of a string is 255 characters.

Variables

A variable is an algebraic symbol used to represent a number or string of characters. X, X%, and X$, respectively, are numeric (values between ±2.93873588E−39 and ±1.70141183E38), integer (whole numbers between −32768 and +32767), and string (up to 255 characters) variables. If the variables haven't been assigned values, numeric and integer variables default to 0, strings to the null character, a string of zero length.

A variable, as the name implies, can be changed at will. The direct mode line

```
X=1: PRINT X: X=2: PRINT X
```

illustrates this point.

Names of variables are subject to these rules:

- The first character must be alphabetic.
- The next character may be alphanumeric.

- Any further alphanumerics are allowed, but not considered part of the variable name.
- The next character may be % or $, denoting integer and string variables, respectively.
- The next character may be (to denote a subscripted variable.
- A name cannot include reserved words, since the BASIC interpreter will treat them as keywords. Note that reserved variables (TI, ST) can be *incorporated* in names (but not used by themselves as variable names), since they are not keywords.

All these rules remove ambiguity and make storage convenient and fast. If 1A were a valid variable name, for example, the line 100 1A=1 would require special treatment to distinguish it from 1001 A=1. And if symbols other than alphanumerics were permitted—so that B= were a valid name, for instance—this would cause problems.

Conversion between different types of numeric variables is automatic; however, string-to-numeric and numeric-to-string conversions require special functions. For example, L%=L/256 automatically truncates L/256 (removing the fractional portion, but *not* rounding it) and checks that the result is in the range −32768 to +32767. L$=STR$(L) and L=VAL(L$) are for converting between numbers and strings. Two other conversion functions are CHR$ and ASC, which operate on single bytes and enable expressions which would otherwise be treated as special cases to be processed.

Operators

Binary operators take two items of the same type and generate a single new item of the same type from them. Unary operators modify a single item. The numeric operators supported by BASIC on the 64 are standard, much like those supported by other computer languages, while the string and logical operators are less similar.

When a string or arithmetic expression is evaluated, the result depends on the priority assigned to each operator and the presence or absence of parentheses. In both string and arithmetic calculations, parentheses insure that the entire expression *within the parentheses* is evaluated as a unit before the other operations are performed. The rules for using parentheses dictate levels of priority, so that an expression in parentheses *within another set of parentheses* will be evaluated first. In the absence of parentheses, priority is assigned to operators in this order, starting with the highest level:

↑	Exponents
+ or −	Unary plus or minus sign—positive or negative number
* or /	Multiply or divide
+ or −	Binary plus or minus—addition or subtraction
< = or >	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 AND B, A and B are called *operands*.

Arithmetic operators work in a straightforward way, but string comparisons are not as simple. 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 up to the end of the shorter string, the shorter one is considered the lesser by string comparison logic. Characters later in the ASCII sequence are considered greater than those earlier in the series. Therefore, the string "1" is less than the string "10", but "5" is greater than "449".

Functions

Some BASIC keywords are valid only when followed by an expression in parentheses; they may be used on the right of assignment statements or within expressions. These are functions: They return a value dependent on the expression in parentheses. Numeric functions return numeric values and include SQR, LOG, and EXP; string functions, which include LEFT$, MID$, RIGHT$, and CHR$, return string values. (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. Some functions (like FRE) take a so-called *dummy argument*: an expression required by the interpreter's syntax-checking routine, but ignored by the code which evaluates the function. Typically, the dummy parameter is a 0, for convenience. The line below is an example:

```
PRINT FRE(0)
```

Expressions

A *numeric expression* is a valid arrangement of numbers, numeric functions, real and integer variables, operators and parentheses, or logical expressions. Numeric expressions can replace numbers in many BASIC constructions, for example, the right side of the assignment statement:

```
X=SQR(M)+PEEK(SCREEN + J)
```

A *string expression* is an arrangement of one or more literals, string functions, string variables, the string operator +, or parentheses. String expressions can replace literals in many BASIC constructions, like this:

```
X$=MID$("HI " + NAME$,1,L)+ CHR$(13)
```

A *logical* (or *Boolean*) *expression* evaluates as true or false (−1 or 0, respectively, in BASIC) and usually contains one or more relational operators (<, =, or >), logical operators, parentheses, numeric expressions, or string expressions. Their main use is in IF statements.

```
IF X$="Y" OR X$="N" GOTO 100
```

contains logical expressions.

BASIC doesn't distinguish sharply 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"
```

which is fairly simple, but also trickier lines like:

```
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. (These lines are examples only, not complete routines that you should type in and run.)

Another aspect of logical expressions is that logical operators can easily be wrongly programmed; because mistyping may be undetected by BASIC, the priority of logical expressions is low (they're executed last), and the meaning of expressions is easily misunderstood. For example:

```
IF PEEK(X)=0 AND PEEK(X+1)=0 THEN END
```

looks for two zero bytes, then ends, which is the desired result, but:

```
IF PEEK(X) AND PEEK(X+1)=0 THEN END
```

ends whenever PEEK(X) is nonzero and PEEK(X+1)=0.

True and false are actually two-byte expressions like integer variables; −1 (true) means all bits are 1; 0 (false) means all bits are 0. Chapter 5 explains in detail.

Every intermediate result in an expression must be valid; numerals must be in the floating-point range, strings no longer than 255 characters, and logical expressions in the 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. The different types of statements are discussed below.

- *Assignment statements.* LET *variable* = *expression.* (LET is optional; but its presence makes the intention behind arithmetically impossible statements, like X=X+1, clearer for the beginner. Languages like Pascal indicate assignments with the symbol :=, which is read as "becomes.")
- *Conditional statements.* IF *logical expression* THEN *statement.*
- *Program control statements.* For example, GOTO, GOSUB, RETURN, STOP.
- *Input statements.* Fetch data from device or from DATA statement: INPUT, GET, INPUT#, GET#, READ.
- *Looping statements.* FOR-NEXT loops, for example.
- *Output statements.* Send data to screen, disk, cassette, or other device: PRINT, PRINT#.
- *REM statements.* These 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 never run and lines that contain only colons can be included in this category.
- *Conversion statements.* These convert between string variables and literals, real variables and numbers, and integers and numerals. Functions like ASC, CHR$, STR$, and VAL are examples.

BASIC programs are made up of numbered program lines; each program line is made up of statements, separated from each other by colons where two or more statements are used on the same program line. Spaces generally are ignored outside quotation marks, as are multiple colons.

BASIC Keyword Dictionary

This section lists every BASIC keyword, with explanations and examples, in a uniform format. Details of error messages are not included here, but collected in an alphabetic list after this section.

Each command's syntax is given in a standard way. Parameters are usually numeric expressions, string expressions, or variables, and these are always carefully distinguished; for example, ABS (*numeric expression*) means that any valid numeric expression is usable with ABS, which in turn implies ABS can be used with variables, as in ABS(X).

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

Numeric functions probably cause most errors. First, there's a chance of a simple SYNTAX ERROR, perhaps an arithmetically wrong construction or omitted parenthesis. Second, number parameters have a wide assortment of range restrictions: byte values must be 0–255, memory addresses must be 0–65535, integer and logical expressions must be within −32768 and +32767, no numbers can be outside approximately −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 errors are mentioned only when, as in DATA, some noteworthy feature exists.

Chapter 11 is a guide to the Commodore 64's ROMs and includes information on the keywords. In a few cases, information is provided in this chapter, where it helps clarify some aspect of BASIC, and tokens are listed for programmers interested in 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: ABS returns the absolute value of the parenthesized numeric expression. 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 check for approximate equality; when TARGET is 6, the program ends only if X is between 5.99 and 6.01. This kind of text is typically used in iterative computations in which a calculated value is expected to converge to a given value.

2. **100 IF ABS(X1−X2)<3 AND ABS(Y1−Y2)<3 GOTO 90**

   From a game program, this recalculates starting positions on screen for two players if randomly generated starting positions are too close.

AND

Type: 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: AND applies the logical AND operator to two expressions. For the purposes of the AND comparison, numeric expressions are evaluated as 16-bit signed integers, so each operand must be in the range −32768 to 32767. Values outside this range result in an ?ILLEGAL QUANTITY ERROR. Each of the 16 bits in the first operand is ANDed with the corresponding bit in the second operand, resulting in a 16-bit, two-byte integer. The four possible combinations of *corresponding individual* bits are:

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

The result becomes 1 only if *both* bits are 1.

AND has two separate 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, as in:

```
POKE 1, PEEK(1) AND 254
```

This forces bit 1 of location 1 to 0, regardless of its previous value (which switches in RAM in place of the BASIC ROM).

Although these uses appear to be different, AND behaves identically in each. A logical expression is treated as false when 0 (all bits in the result are zero), and is considered true when −1 (all bits in the result are 1) or nonzero.

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**

   This 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 = %11111000. Therefore, X AND 248 clears the three rightmost bits to 0.

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. Use:

   ```
   IF NOT OK THEN line number
   ```

   to branch for reinput if the data was unacceptable.

ASC

Type: Numeric function

Syntax: ASC(*string expression 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 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 the Appendices for a table of ASCII values.

Note that the converse function to ASC is CHR$, so ASC(CHR$(N)) is the same as N, and CHR$(ASC("P")) is the character P. All keys except RUN/STOP, SHIFT, CTRL, the 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$. Adding CHR$(0) allows detection of the null character, which otherwise gives ?ILLEGAL QUANTITY ERROR.

2. **X = ASC(X$)−192**

   Converts uppercase (SHIFTed) A–Z to 1–26. Useful when computing checksums, where each letter has to be converted to a number.

3. **1000 IF PEEK(L)=ASC("*") THEN PRINT "FOUND AT" L**

   Shows how using ASC can make your programs more readable; 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: This is the arc tangent, or inverse tangent, function. This function returns, in radians in the range $-\pi/2$ to $+\pi/2$, the angle whose tangent is the numeric expression. The expression may take any value within the valid range for floating-point numbers, approximately ±1.7E38.

To convert radians to degrees, multiply by $180/\pi$. This changes the range of values of ATN from $-\pi/2$ through $\pi/2$ to $-90°$ through $90°$.

In some cases, ATN(X) is a useful transformation to apply, since it condenses almost the entire number range into a finite set 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 easting and northing.

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 cosine, respectively. Remember that the arc tangent can never be exactly 90 degrees; if necessary, test for this extreme value to avoid errors.

CHR$

Type: String function

Syntax: CHR$(*numeric expression*)

Modes: Direct and program modes are both valid.

Token: $C7 (199)

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

Purpose: CHR$ converts a numeric expression (which must evaluate and truncate to an integer in the range 0–255) to the corresponding ASCII character. It is useful for manipulating special characters like RETURN and quotes which are CHR$(13) and CHR$(34), respectively. Check the Appendices for a table of ASCII values. Note that ASC is the converse function of CHR$.

Examples:

1. **A$=CHR$(18)+NAME$+CHR$(146)**
   This adds {RVS} and {OFF} around NAME$, so PRINT A$ prints NAME$ in reverse video.
2. **FOR J=833 TO 848: PRINT CHR$(PEEK(J));:NEXT**
   This prints the name of the most recently loaded tape program, by reading the characters from the tape buffer, assuming the buffer hasn't been altered by a program.
3. **PRINT#4, CHR$(27)"E08"**
   The above command sends the ASCII ESC (escape) character, plus a command, to a printer. Special printer features are often controlled like this, and the codes will vary from one brand of printer to the next.
4. **OPEN 2,2,0,CHR$(38)+CHR$(60)**
   A command which opens a file to a modem. The two CHR$ parameters are required in this format by BASIC.
   CHR$(0) represents the null character, but, unlike the *null string,* "", it has a length of one, and can be added to strings. See ASC for an application. Embedded null characters, as in Y$="12"+CHR$(0)+"34" can cause strange results.

CLOSE

Type: Input/output statement

Syntax: CLOSE *numeric expression*

Modes: Direct and program modes are both valid.

Token: $A0 (160)

Abbreviated entry: CL SHIFT-O

Purpose: CLOSE completes the processing of the specified file and deletes its file number, device number, and secondary address from the file tables.

A numeric expression may be used as a logical file number; it must evaluate to a number in the range 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, but only the first is used.)

Notes:

1. Files opened for reading do not have to be closed, but files opened for saving to tape or disk should always be closed, or tape files will lose the last portion of data held in the buffer, while disks may be corrupted. Chapters 14 and 15 have details. (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.)
2. CLOSE is a straightforward command, but it is made more complicated by the behavior of CMD, which must be followed by a PRINT# command to switch output back to the TV or monitor.

Example:

```
OPEN 4,4: PRINT#4,"HELLO": CLOSE 4
```

The line above opens a file, sends data to a printer through the file, then closes the file. The second number in the OPEN command is a *device number,* which selects the printer (device 4).

CLR

Type: Statement

Syntax: CLR

Modes: Direct and program modes are both valid.

Token: $9C (156)

Abbreviated entry: C SHIFT-L

Purpose: CLR clears the memory area currently allocated to variables, leaving the BASIC program, if there is one, unchanged. Any machine language routines in RAM are left unaltered. Additional effects are noted below.

Note: CLR is actually part of NEW, and does most of the things NEW does, while keeping the current program intact. CLR operates by resetting pointers, and *doesn't actually erase variables,* so in principle these could be recovered. It has other functions, too. Following is the complete list:

- The string pointer is set to top-of-memory, and the end-of-variables and end-of-arrays pointers are set to end-of-BASIC. All variables and arrays are thus effectively lost.
- The stack pointer is reset, but the previous address is retained; therefore, all FOR-NEXT and GOSUB-RETURN references are lost, and also, if CLR executes within a program, that program continues at the same place.
- The DATA pointer is set to start.
- Input/output activity is aborted.
- Files are aborted (but not closed), and keyboard and screen become the input/output devices.

Examples:
1. **POKE 55,0: POKE 56,48: CLR**
   Sets the top of the BASIC program storage area to 48*256=$3000, typically to reserve space for graphics in VIC bank 0.
2. **1000 CLR: GOTO 10**
   This sort of operation is useful in some simulation programs; all existing variables are erased and the program continues. RUN 10 has a similar effect.

CMD

Type: Output statement

Syntax: CMD *numeric expression [, any expression]*

The numeric expression, a file number, must evaluate to a number in the range 1–255. The optional expression does not include the brackets shown above, but must follow a comma; it is printed to the specified file and can be used to put a header on a printout.

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, a CMD statement followed by a device number redirects printed output from TV to the specified device. 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 changing all PRINT statements to PRINT# statements. However, CMD has bugs; GET and sometimes GOSUB will redirect output to screen. Where this is a problem, use PRINT#.
2. CMD is necessary in order to list programs to printers.

Examples:
1. **OPEN 4,4: CMD4,"TITLE":LIST**
   This will list the current program (or disk directory file, if present in memory) to a printer. Follow this with:

   PRINT#4: CLOSE4

   to return 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: Only direct mode is available. (In program mode CONT enters an infinite loop.)
Token: $9A (154)
Abbreviated entry: C SHIFT-O
Purpose: CONT resumes execution of a BASIC program stopped by a STOP or END statement, or by the RUN/STOP key. CONT cannot be used to restart a program that has stopped due to any sort of error. Also, CONT cannot be used if you edit any program lines after the program stops.

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 continue, provided you make no error. ?CAN'T CONTINUE ERROR has several causes. In such cases, GOTO a line number serves a similar purpose as CONT.

Note: Because STOP aborts files, CONT may be accepted, but not actually continue as before; for example, output which ought to go to a printer may be displayed on the screen after CONT.

Example:

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

Run this, then press the RUN/STOP key. The BASIC command CONT will cause the program to continue. You can change J, by typing J=10000 in direct mode, for example, and CONT will resume (using 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: COS returns the cosine of the numeric expression, which is assumed to be an angle expressed in radians.

Examples:

1. **PRINT COS(45* π/180)**

   The above statement prints the cosine of 45 degrees (conversion from radians to degrees is accomplished by multiplying the value in radians by π and dividing by 180).

2. **FOR J=0 TO 1000 STEP π/10: PRINT COS(J): NEXT**

   This 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: Statement

Syntax: DATA *list of data separated by commas*

Modes: Only program mode is available.

Token: $83 (131)

Abbreviated entry: D SHIFT-A

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

Notes:

1. DATA statements to store ML programs can be generated automatically: See Chapter 9.
2. A ?SYNTAX ERROR in a valid DATA statement line means that the READ and DATA statements don't match properly.
3. Unnoticed or omitted commas can cause baffling bugs:

   DATA R,O,Y,G,,B,P,

   contains eight data items, two of them (between G and B, and following P) null characters.
4. Because DATA statements are handled in sequence (RESTORE restarts the sequence), take care when adding more data (for example, by appending a subroutine) in case data from a wrong routine is read.

Examples:

1. **100 DATA "7975, LAZY RIVER ROAD"**

   This shows that quotes enable commas, colons, leading spaces, and other special characters to be included in strings.
2. **1000 DATA CU,COPPER,136.2,FE,IRON,25.1**

   This illustrates how sets of data can be stored. Typically, a loop with READ A$,M$,W inside might be used to read each set of three items.
3. **10000 DATA SUB1 :REM MARKS START OF SUB1's DATA**

   Here's a trick that might be used to insure that the correct data is being read. Use the following line to locate SUB1:

   1000 READX$: IF X$<> "SUB1" GOTO 1000

DEF FN

Type: DEF FN, statement; FN, numeric function

Syntax: DEF FN *valid variable name* (*real variable*)=*arithmetic expression*

Modes: Only program mode is available.

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

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

Purpose: DEF FN 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 function definitions are stored along with ordinary variables. See Chapter 6 on storage. Once defined, functions can be called by FN in direct mode.
2. ?UNDEF'D FUNCTION ERROR results if DEF FN hasn't been encountered before FN is used. ?SYNTAX ERROR, when caused by a definition, refers to the line using FN, even when that line is valid.
3. After loading a new program from within BASIC, redefine any functions; otherwise, 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; IF-THEN statements aren't allowed in the function definition, so logical expressions may be necessary—see the examples. Calling another function definition is valid, however.
5. The dependent variable need not be used in the definition; if not, it's called a *dummy variable.*

Examples:

1. **100 DEF FN DEEK(X) = PEEK(X)+256*PEEK(X+1)**
   110 PRINT FN DEEK(50)
   These lines print the decimal value of the two-byte quantity stored in memory locations 50 and 51. (DEEK is a double-byte PEEK.)
2. **100 DEF FN MIN(X) = −(A>B)*B−(B>A)*A**
   This will return the smaller of A and B. Note that X is a dummy variable in this case; any other variable could be used. The awkward form of the expression is necessary to fit it into a single statement.
3. **100 DEF FN PV(I) = 100/ (1+I/100)**
   This 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
   The above lines show in outline how a very long expression can be spread over several lines of BASIC.

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: DIM is short for DIMension. It sets up space above the BASIC program in memory for variables in the order the variables are listed in the DIM statement. This command is automatically carried out when a variable is given a value (for example, a line that contains the expression X=1), so there's no need for DIM, unless arrays with dimensions greater than 10 are needed. All variables set up by DIM are set to 0 (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). Dimensions can have a maximum of 32767 elements, and not more than 255 subscripts may be used in multidimensional arrays; in practice, an ?OUT OF MEMORY ERROR will result long before you reach these limits.
2. Arrays are stored in memory above regular variables. Chapter 6 explains the consequences in detail, but here are a couple:
   - New variables introduced after an array has been set up cause a delay.
   - Arrays can be deleted with:

   POKE 49,PEEK(47): POKE 50,PEEK(48)

   So if intermediate results are computed with a large array, this can be deleted when you are finished.
3. Integer arrays are efficient, while the efficiency of string arrays depends on the lengths of the strings.

   PRINT FRE(0)

   gives a quick indication of spare RAM at any time. RAM space occupied by arrays is explained in Chapter 6.
4. Large string arrays are vulnerable to *garbage collection* delays, also explained in Chapter 6. The total number of separate strings, not their lengths, is the significant factor in garbage collection.

Examples:

1. **100 INPUT "NUMBER OF ITEMS";N: DIM IT$(N)**

   This 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**

   Ordering variables, with the most frequently used ones dimensioned first, will help increase the speed of BASIC programs.
3. **100 DIM A(20): FOR J=1 TO 20: INPUT A(J): A(0)=A(0)+A(J): NEXT**

   The above line uses the zeroth element to keep a running total.
4. **DIM X%(10,10,10)**

   This sets up an array of 1331 integers, perhaps to store the results of three 11-point questionnaires.

END

Type: Statement
Syntax: END
Modes: Direct and program modes are both valid.

Token: $80 (128)

Abbreviated entry: E SHIFT-N

Purpose: END causes a program to cease execution and exit to immediate mode. This command may be used to set breakpoints in a program; CONT causes a program to continue at the instruction after END. BASIC doesn't always need END; a program can simply run out of lines, but END is needed to finish the program in the middle. END leaves BASIC available for LISTing; you may prefer to prevent this with NEW or SYS 64738 in place of END. (*Note:* Early computers always needed END, to separate each program's punch cards. Now this isn't so.)

Examples:

1. **10000 IF ABS(BEST−V) <.001 THEN PRINT BEST: END**

   This causes the program to end when a repeating process has found a solution to a problem within a desired accuracy range.
2. **100 GOSUB 1000: END: GOSUB 2000: END: GOSUB 3000: END**

   These lines are from a program being developed; 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: EXP calculates e (2.7182818...) to any power within the range −88 to +88, approximately. The result is always positive, approaching 0 with negative arguments, becoming large with positive arguments. EXP(0) is 1.

Note: EXP is the converse of LOG. Sometimes logarithms of numbers are used in calculations; EXP transforms the results back to normal. EXP(Q) could be replaced by 2.7182818↑Q, but the shorter form is more readable.

Examples:

1. **PRINT EXP(LOG(N))**

   The above line prints N (possibly with rounding error), demonstrating that EXP and LOG are converse operations.
2. **100 P(N) = M↑N * EXP(−M)/FACT(N)**

   This is a typical statistical formula, for the probability of exactly N rare events happening when the average is M. FACT(N) holds N! for a suitable range of values. (EXP is important for its special property that it equals its own rate of growth; it tends to turn up in scientific calculations.)

FOR-TO (STEP)

Type: Statement

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

Modes: Direct and program modes are both valid.
Token: FOR: $81 (129); TO: $A4 (164); STEP: $A9 (169)
Abbreviated entry: FOR: F SHIFT-O; TO: None; STEP: ST SHIFT-E
Purpose: FOR-TO [STEP] provides a method to count the number of times a portion of BASIC is executed.

Notes:

1. *How FOR-NEXT loops work.* The syntax after FOR is checked, rejecting, for example, FOR X%=1 TO 10. 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. Now, 18 bytes are put on the stack, if there's room. Once they are placed there, they won't change, so the upper limit of the loop FOR X=1 TO N won't change after the looping starts, even if the value of N is changed within the loop.

   ```
   10 FOR X=489 TO 506: PRINT PEEK(X): NEXT
   ```

   lists 18 bytes from the stack; these are the FOR token, the two-byte address of the loop variable, the STEP size in floating-point format, the sign of the STEP, the floating-point value of the upper limit of the loop, the line number of the FOR, and the address to jump to after the loop is finished. The STEP value defaults to 1.

   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, for example, 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, if a negative STEP size was used), processing continues with the statement following 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. So:

   ```
   FOR J=1 to 10000 STEP .5
   ```

   is exact, as is the default STEP size of 1. On the other hand:

   ```
   FOR M=1 TO 1000 STEP 1/3: PRINT M: NEXT
   ```

   will produce errors. Chapter 6 explains this in more detail.

   This description should enable you to pinpoint bugs in loops, which can be difficult to locate without detailed information.

2. *Loop execution speed.* When fine-tuning a long program for speed, pay special attention to loops, because inefficiencies are magnified in proportion to the loop's size. If you dimension variables in decreasing order of importance, this can increase the speed of execution (don't dimension any variables inside the loop, though, as this will cause an error condition).

3. *Exiting from loops.* One of the best ways to exit from a loop is from the NEXT statement, and changing the loop variable is a simple way to accomplish this. For example:

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

finishes early if the A key is pressed.

4. *Other loops.* The extended command DO WHILE can be simulated with:

   FOR J=−1 TO 0: *statements you wish to execute* : J=CONDITION: NEXT

   Processing continues until J is false. Obviously, more intricate looping structures are possible.

Examples:

1. **PRINT "{CLR}": FOR J=1 TO 500: PRINT "*";: NEXT**
   The above example line prints 500 asterisks.
2. **K=0: FOR J=1024 TO 1024+255: POKE J,K: K=K+1: NEXT**
   This POKEs characters 0 to 255 sequentially into screen memory. K counts along with J.
3. **FOR J=2048 TO 9E9: IF PEEK(J)<>123 THEN NEXT:PRINT J**
   These statements search memory from 2048 upward for a byte equal to 123. When the loop ends, PRINT J gives the location.
4. **5 FOR J=1 TO 12**
   10 IF M$<>MID$("JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC", 3*J−2,3) THEN NEXT
   The above lines match a correctly entered month abbreviation, previously input as M$. When the program is finished running, J will be a number from 1 to 12, indicating the month (or 13 if no match was found).

FRE

Type: Numeric function

Syntax: FRE (*numeric 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. If there are more than 32767, the value returned is negative; adding 65536 converts this to the true figure. The Commodore 64 has 64K of RAM memory, but 24K of this is normally hidden by overlying ROM and doesn't appear in the total of free bytes. See Chapter 5. FRE is useful for removing unused dynamic strings, which take up variable space in RAM and are a potential source of ?OUT OF MEMORY ERRORs. FRE first performs a *garbage collection* (see Chapter 6) before returning its value. FRE uses a dummy expression; usually a zero is used, so the expression is FRE(0).

Examples:

1. **1000 IF FRE(0)<100 PRINT "SHORT OF RAM"**
   The above example prints a message when free memory is below 100 bytes or returns a negative value. This routine would need to be modified to be useful, because of the negative values FRE returns (see explanation above).

2. **F=FRE(0): DIM X$(50): PRINT F−FRE(0)**
   This prints the number of bytes used to dimension X$ to 50.
3. **200 F=FRE(0)−(FRE(0)<0)*65536**
   This calculates the number of free bytes under any circumstances, storing the answer in F.

GET

Type: Input statement

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

Mode: Only program mode is available.

Token: $A1 (161)

Abbreviated entry: G SHIFT-E

Purpose: GET 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 a null, and numeric variables are given a value of 0. GET (unlike INPUT, or GET on some other computers) doesn't wait for a keypress, so BASIC can test for a key and continue if there isn't one. GET X$ is more powerful than GET X, which crashes when it detects a nonnumeric key; so the string form of GET is nearly always used, and conversions are made when necessary. The string GET works with any ASCII character, but the RUN/STOP, Commodore, SHIFT, CTRL, and RESTORE keys 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 reliable INPUT-like routines.

Examples:

1. **5 GET X$: IF X$="" GOTO 5: REM AWAIT KEY**
   10 PRINT "{CLR}" X$ ASC(X$): GOTO 5
   This short program waits for a key to be pressed, then clears the screen and prints the key pressed and its ASCII value. There are a few exceptions, like quotes and the color controls. You'll see how RETURN is read, plus all the normal keys. Chapter 4 discusses this in more depth.
2. **100 DIM IN$(200): FOR J=1 TO 200: GET IN$(J): NEXT**
   This line gets 200 characters into an array, most of which will be nulls.
3. **200 GET A$,B$,C$**
   This is a syntactically valid statement, but this format is more appropriate with the GET# statement. The syntax is accepted because GET, GET#, INPUT, and READ largely use the same Kernal routines, and the 64's designers felt it was not worth removing the relatively useless alternatives.

GET#

Type: Input statement

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

The numeric expression, a file number, must evaluate and truncate to a number in the range 1–255.

Mode: Only program mode is available.

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

Abbreviated entry: G SHIFT-E #

Purpose: GET# reads a single character from the specified file, which must be open to an input device or a ?FILE NOT OPEN ERROR will result. Unlike the INPUT# statement, GET# can read characters like colons, quotes, and RETURNs. GET# can read files character by character in a way impossible with INPUT# and is not limited to 88 characters per string.

Notes:
1. GET# can read from screen or keyboard, although there's usually no real advantage in this.
2. GET# from tape sets the status variable (ST) to a value of 64 when it reaches the end-of-file, so programs can evaluate ST to 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 also sets ST=64 at end-of-file; from then on, ST is set to 66, which indicates 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 program extract reads in a string, character by character, from tape or disk, building IN$ from each character, and exiting to the next part of the program when RETURN indicates the end of a string. A routine at line 2000 would handle the string after the GET# process was complete.
2. **100 GET#8,X$: IF ST=64 GOTO 1000: REM END OF DATA**
   The above line shows how to use ST to detect that no more data is on file, and how to jump to another part of the program based on that information.
3. **100 GET#1,X$,Y$**
   This example program line GETs a pair of consecutive characters from file 1, which has already been opened to an input device.

GO

Type: Dummy statement

Syntax: Always part of GO TO

Modes: Direct and program modes are both valid.

Token: $CB (203)

Abbreviated entry: None

Purpose: The sole function of GO is to allow GO TO as a valid form of GOTO, which occasionally gives problems. For example, some renumbering programs ignore it, some early CBM machines don't have it. Chapter 8 shows how you can modify GO to suit 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 the specified BASIC line, saving the address of the original line on the stack, so that the RETURN statement in a program can transfer control back to the statement immediately following the GOSUB statement. 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. *Testing subroutines in direct mode.* It is often simple to test parts of a large program while in direct mode. For example:

   ```
   L=1234: GOSUB 500
   ```

   tests the decimal/hex converter in Chapter 6's "PRINT USING."
2. *Processing GOSUB.* Line numbers following GOSUB are scanned by a routine similar to VAL; numbers are input until a nonnumeric 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.

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

   The above program prints five bytes from the stack, which are a GOSUB token (141), GOSUB's line number, and a pointer to the GOSUB statement. 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 and cause ?OUT OF MEMORY ERROR. Type and run the following one-line program to see the effect:

   ```
   100 GOSUB 100
   ```
3. *Miscellaneous.* Chapter 6 has a computed GOSUB, and a POP to delete GOSUB statements that don't have matching RETURN statements. GOSUB 500: RETURN is identical to GOTO 500 in its effect, but uses more space on the stack.

   Structured programming makes a lot of use of subroutines; rewriting programs which use multiple IFs or other complex constructions into subroutines helps make programs clearer. See Chapter 4 for more on this.

Examples:

1.
```
100 EM$="DISK NOT IN DRIVE":GOSUB2000
110 END
2000 PRINT "{HOME} {RVS}*** ERROR " EM$ " {OFF}"
2010 FOR J=1 TO 2000: NEXT: RETURN
```

Part of a simplified error message routine, this excerpt prints an error (EM$ must be set before GOSUB 2000) in reverse at the top of the screen.

2. **500 GOSUB 510**
   510 PRINT"*"
   520 RETURN
   This shows how a subroutine can have several entry points. Here, GOSUB 510 prints an asterisk, while GOSUB 500 prints it twice.

GOTO; GO TO

Type: Statement

Syntax: GOTO *line number;* GO TO *line number*

Modes: Direct and program modes are both valid.

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

Abbreviated entry: G SHIFT-O

Purpose: GOTO jumps to the specified BASIC line. IF and ON allow conditional GOTO statements.

Notes:
1. *Using GOTO in direct mode.* Direct mode GOTO executes the program in memory *without* executing CLR, so if the program has been previously run, the variables are retained. This is similar to CONT, except that any line can be selected as the starting point. Variables can be changed, but these will be lost if any BASIC program lines are edited.
2. Line numbers are read by the same routine that handles GOSUB's line numbers, and similar restrictions apply.

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

IF-THEN

Type: Conditional statement

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: IF-THEN statements allow conditional branching to any program line or conditional execution of statements after THEN.

The expression after IF is treated as Boolean (that is, if zero it's false, if nonzero true). If the expression is true, the statement after THEN is performed; if it is 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, so IF X$ THEN may be true or false.)

Examples:

1. **1000 LC=LC+1: IF LC=60 THEN LC=0: GOSUB 5000**

   This program excerpt increments the line count (LC); if LC is 60, it resets the count value to 0 and calls a form advance subroutine at 5000 before continuing.
2. **700 IF X=1 THEN IF A=4 AND B=9 THEN PRINT "*"**

   This is a composite IF statement, identical in effect to:

   IF X=1 AND A=4 AND B=9 THEN PRINT "*"

   but probably a little faster.
3. **500 IF X THEN PRINT "NONZERO"**

   This example illustrates that IF X THEN is the same as IF X<>0 THEN.

INPUT

Type: Input statement

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

Modes: Only program mode is available.

Token: $85 (133)

Abbreviated entry: None

Purpose: INPUT is an easily programmed command which takes in data from the 64 and assigns it to a variable. INPUT echoes the data to the screen, so editing features like DEL can be used. Pressing the RETURN key signals the end of INPUT.

Notes:

1. *INPUT's prompts.* INPUT N$ and INPUT "NAME";N$ illustrate the two forms of INPUT. Both print a question mark followed by a flashing cursor, but the second version also prints NAME, giving NAME? as the prompt. When you use an INPUT statement with multiple variables—for example, INPUT X$,Y$—two consecutive question marks (??) will be displayed on the screen if only the first string is entered, followed by RETURN. In response to the double question marks, you simply enter the remaining data. Typing the two entries, separated by a comma (FIRST, SECOND) assigns both strings, with no further prompt.

   These demonstration lines show how the prompt string can be used:

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

   This clears the screen and moves the cursor (and the question mark prompt) down and to the right of the home position.

   100 INPUT "{2 SPACES}--{LEFT}{LEFT}{LEFT}{LEFT}",X$

   This illustrates the technique of indicating the expected length of the input data. There should be two spaces after the first quote, and two more cursor-lefts than hyphens.

```
100 INPUT "{RED}{RVS}NAME",N$
```

This prints the prompt in red and reverse video. Long prompt strings may cause problems. If the user's response is long enough to cause the cursor to wrap around (move to the next line), the prompt may be added to the input. To avoid this, you may want to use a combination of PRINT and INPUT statements:

```
PRINT "A LONG PROMPT MESSAGE";:INPUT X$
```

You may not have to worry about this problem, since recent 64s are free from this bug.

Another approach is to POKE the keyboard buffer; in this way the question mark can be eliminated. Type and run the following line:

```
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, allowing strings like LDA $A000,X to be input despite their containing commas or colons. Chapter 6 has more on this.

2. *How input data is handled.* When you press RETURN, the line of data is put into the input buffer to await processing. Chapter 6 has a more detailed discussion, but note that one effect of this is the inability to use INPUT in direct mode. Chapter 7 explains how ML can solve this problem. After entry, the data in the buffer is matched with the list of variables after INPUT. The message ?EXTRA IGNORED means too many separate items were entered; the prompt ?? means too few items were entered and requests more; and ?REDO FROM START means the variable types didn't match the data entered.

```
100 INPUT X
```

The above line causes the computer to expect numeric input; it will accept 123.4 and even 1E4, but not HELLO.

```
100 INPUT X$,Y$
```

In the above example, the computer expects two strings to be entered. It will accept HELLO, THERE but as two separate strings; if HELLO, THERE, 64 is entered, the computer will accept HELLO as the first string, THERE as the second, and will not accept the characters 64, but will instead respond with ?EXTRA IGNORED.

Generally, these aren't serious problems, unless a program is intended to be foolproof, in which case the GET statement is essential (see Chapter 4 for more information). Without some kind of *error trapping*, a user could type HOME, CTRL-WHT, SHIFT–RUN/STOP, or quotes and the INPUT statement would be wrecked.

Note: All prompts (?, ??, ?EXTRA IGNORED) are suppressed when INPUT# is in use. When using INPUT:

```
POKE 19,1
```

has this effect. Similarly, the following line, which opens a file to the keyboard, suppresses prompts:

```
OPEN 1,0: INPUT#1,X$
```

Examples:
1. **INPUT "ENTER NAME";N$: PRINT "HELLO, " N$**
   This is a straightforward string input of N$, followed by a personal greeting.
2. **FOR J=1 TO 10: INPUT X(J): NEXT**
   The above line inputs ten numbers into an array.
3. **100 INPUT "DEVICE{2 SPACES}3{LEFT}{LEFT}{LEFT}";D**
   This example shows one method to allow a default value for INPUT. In this case, simply pressing RETURN assigns the value 3 to D, which saves time for the user. Another method, which doesn't print the default under the cursor, is this:
4. **100 X$="YES": INPUT X$**
   If the user simply presses RETURN, X$ retains the value "YES".

INPUT#

Type: Input statement

Syntax: INPUT# *numeric expression, variable name [,variable name...]*

The numeric expression, a file number, must evaluate and truncate to a number in the range 1–255.

Mode: Only program mode is available.

Token: $84 (132)

Abbreviated entry: I SHIFT-N (This includes the #.)

Purpose: INPUT# provides an easy method to read variables from a file, usually on tape or disk. The format is the same as with PRINT#; the data consists of ASCII characters separated by RETURN characters. Provided INPUT# matches PRINT#, this command should be trouble-free.

Note: INPUT# is closely similar to INPUT. Below is a list of differences between the two:

- Since the device generally can't use it, no prompt is printed.
- Some characters are ignored (like spaces without text, and screen editing characters) unless preceded by quotes. Similarly:

  PRINT#1,"HELLO:THERE"

  is read by INPUT# as two strings, because the colons are treated as separators. Usually, PRINT# with straightforward variables will help you avoid these bugs.
- INPUT# can't take in a string longer than 88 characters, as this results in a ?STRING TOO LONG ERROR. Screen input doesn't have this problem, since part of an overlong string is simply ignored.
- ST signals the end-of-file point, as it does with the GET# statement.

Example:

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

This example 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: INT, the integer function, converts the numeric expression to the nearest integer less than or equal to the expression. Two sample results of the integer function are INT(10.4) is 10, and INT(−2.2) is −3. The expression is assumed to be in the full range for numerals, between about −1.7 E38 and +1.7 E38. So 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 WHOLE NUMBER**

   The above example rounds numbers—including negative numbers—to the nearest whole numer.
2. **100 PRICE = INT(.5 + P*(1+MARKUP/100))**

   This calculates price to the nearest cent from percentage markup and purchase price in cents.

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 the $.)

Purpose: LEFT$ returns a substring consisting of the leftmost characters of the original string expression. The numeric expression (which must evaluate to 0–255) is compared with the length of the string; the smaller of the two determines the length of the substring.

Examples:

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

   This prints 20 strings, "", "H", "HE", "HEL", and so on. As J increases, the number of characters printed does, too. However, the number of characters printed never exceeds 11, the length of the original string. Thus, the eleventh through twentieth strings printed will be identical.
2. **PRINT LEFT$(X$ + "--------------------",20)**

   This formatting trick pads X$ to exactly 20 characters with hyphens. This way the output is always 20 characters long.
3. **PRINT LEFT$("--------------------",20-LEN(X$)); X$**

   This line right justifies X$, preceding it with hyphens. (X$ is assumed not to be longer than 20. Other characters, notably spaces, are usable in the same way to format output.)

LEN

Type: Numeric function
Syntax: LEN(*string expression*)
Modes: Direct and program modes are both valid.
Token: $C3 (195)
Abbreviated entry: None
Purpose: LEN determines the length of a string expression. The result is always between 0 and 255 (see Chapter 6 for more details).
Examples:
1. **10 PRINT SPC(20−LEN(MSG$)/2);MSG$**

   This line will center a short message on the computer screen by adding leading spaces.
2. **50 IF LEN(IN$)<>L THEN PRINT "MUST BE" L "DIGITS": GOTO 40**

   The above excerpt rejects an input string of the wrong length, sending the program back to line 40, where another attempt may be tried.
3. **100 FOR J=1 TO LEN(W$): IF L$=MID$(W$,J,1) GOTO 200: NEXT**
   110 PRINT "NOT FOUND"

   This checks word W$ for the presence of letter L$. The use of LEN(W$) allows the program to set the loop counter for any length of W$.

LET

Type: Statement
Syntax: [LET] *numeric variable = numeric or logical expression*
[LET] *integer variable = numeric expression in range −32768 to +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: LET assigns a number or string to a variable. The statement LET is usually omitted, since the 64 assumes LET by default. Simple or array variables may be used, and if a simple or array variable doesn't already exist, LET makes room for it in the variable storage area in memory. LET will dimension an array to the default size of 11 (elements 0 through 10) if it has not been previously dimensioned with a DIM statement.
Notes:
1. Chapter 6 has full details on variable storage; it also has a routine, VARPTR, showing how LET can be used from ML. Since LET is rarely used, it can be modifed by the user (Chapter 7 demonstrates this).
2. Variables can be reassigned freely, so be careful not to try using a variable for two purposes simultaneously. This is often a problem when using subroutines, because it is harder to keep track of variables.

Examples:

1. **X=123456: LET X=123456**
   Both of these statements set X to 123456.
2. **Q%=Q/100**
   The above line sets Q% equal to the integer portion of Q/100, so if Q=1234, Q%=12.
3. **LET QH%=Q/256: LET QL%=Q−QH%*256**
   This sets QH% and QL% equal to the high and low bytes of the number Q.

LIST

Type: Command

Syntax: LIST *[line number] [-] [line number]*

Modes: Direct and program modes are both valid.

Token: $9B (155)

Abbreviated entry: L SHIFT-I

Purpose: LIST displays part or all of BASIC in memory to the screen, or (with CMD) to disk, tape, or other output device.

Notes:

1. Line numbers must be ASCII characters, not variables.
2. LIST uses many RAM locations; it always exits to READY mode if used within a program.
3. LOAD errors and other errors may show up in LIST. For example, if a machine language program designed for some other part of memory is loaded into the BASIC program area, an attempt to LIST will show only random characters.
4. Chapter 6 lists BASIC tokens and has examples of BASIC storage in memory. Also, a LIST reference has programs to modify LIST in useful ways. (Chapter 8 shows how it's done.) The entry for 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**
   The line above displays the BASIC lines from 2000 through 2999.
2. **LOAD "$",8** followed by **LIST**
   This displays a disk directory, which is stored in memory as though it were a BASIC program.
3. **1000 LIST -10**
   This lists all lines up to and including line 10 of a BASIC program. As shown in this example, LIST can be included within a BASIC program line. However, execution of this line will stop the program, and CONT will not restart it.
4. **LIST 1100-**
   This displays 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, if used, 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*]

A name and a numeric expression, typically 8, which is the device number, are required. The second numeric parameter has the same meaning as in tape LOAD. Chapter 15 has full details.

Modem: LOAD cannot be used with a modem. Attempting to use device number 2, therefore, will result in an error.

Modes: Direct and program modes are both valid. (See "Notes" below.)

Token: $93 (147)

Abbreviated entry: L SHIFT-O

Purpose: LOAD reads from an external device, filling RAM with a BASIC program, ML, graphics, or other data. In its simplest form, LOAD {RETURN}, then RUN {RETURN} loads BASIC from tape and runs it. (SHIFT–RUN/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 chapters on tape and disk usage. 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 tidy.
2. Tape LOAD blanks the TV screen to the border color during any tape reading. When a program or file header is found, a FOUND message is displayed on screen for about ten seconds, after which loading takes place if the program name is acceptable, and the screen temporarily blanks again. Otherwise, the process of searching goes on. Pressing the Commodore key, or one of several other keys, cuts the ten-second pause short.
3. A BASIC program LOAD 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 the 64 with any memory configuration and prepares it for RUN. In fact:

   ```
   POKE 43,LO: POKE 44,HI: POKE HI*256+LO,0: NEW
   ```

   followed by the correct disk or tape LOAD can put a BASIC program anywhere you choose, if there's room for it.
4. Loading ML, graphics definitions, and other data is generally trickier than loading BASIC programs, and needs a LOAD format like this:

   ```
   LOAD "CHARSET",1,1
   ```

to insure that the data is put back where it came from. *Supermon*'s .L load command (see Chapter 7) does this. Block LOAD in Chapter 6 explains how blocks of bytes can be loaded without disturbing BASIC.

5. Program mode LOADs generally *chain* BASIC; see CHAIN in Chapter 6, and also OLD, which explains how to chain a long program from a shorter one.

Examples:

1. **LOAD: LOAD "",1**
   These are tape LOADs and have identical effects. Either loads the first BASIC program found on tape.
2. **LOAD "PROG"**
   This loads the program with the filename PROG from tape. Actually, because of the filename checking scheme used, the first program encountered on tape having a name beginning with PROG (PROGRAM, or PROGDEMO, for example) will be loaded.
3. **LOAD "PROG",8**
   This line will load only PROG from disk. No other program with a name beginning with PROG will be loaded if PROG is not found; instead, a ?FILE NOT FOUND ERROR will be reported.
4. **LOAD "PAC*",8**
   This illustrates a typical disk pattern-matching LOAD command, which will load PACMAN, PACKER, or the first program beginning with PAC.
5. **10000 PRINT "PLEASE WAIT": LOAD "PART2"**
   The above line loads and then runs the tape program PART2 from within BASIC. If the correct key on the tape deck is pressed, no message appears on the TV.
6. **10 IF X=0 THEN X=1: LOAD "GRAPHICS",1,1**
   20 REM THE PROGRAM CONTINUES HERE AFTER THE LOAD
   This loads the graphics into a fixed area of memory. A LOAD command from within a program causes that program to be run again *from the start*. The variable, X, is used as a flag, which prevents GRAPHICS from being loaded repeatedly and allows the program to continue.

LOG

Type: Numeric function

Syntax: LOG(*numeric expression*)

Modes: Direct and program modes are both valid.

Token: $BC (188)

Abbreviated entry: None

Purpose: LOG 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 used mainly in scientific work; their susceptibility 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 uses of the LOG function.
2. **LF=(N+.5)*(LOG(N)−1) + 1.41894 + 1/(12*N)**
   This defines LF, an approximation of LOG(N!), so that EXP(LF) approximately equals N!. This illustrates how LOG helps when using 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 the $.)

Purpose: MID$ 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 to be extracted. If this isn't used, the substring extends to the end of the original string.

Examples:
1. **N$=MID$(STR$(N),2) :REM REMOVE LEADING SPACE FROM N**
   This is useful when a number's leading spaces aren't wanted. It works with any positive numbers in the correct range.
2. **10 INPUT X$: L=LEN(X$)**
   20 FOR J=1 TO L: PRINT MID$(X$,L−J+1,1);: NEXT
   This inputs a string, then prints it out backward, 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: NEW allows a new BASIC program to be entered, by ignoring any previous program. It corrects pointers after a forced (nonrelocating) LOAD like:

LOAD "ML",1,1

so BASIC can operate without an ?OUT OF MEMORY ERROR.

Notes:
1. Actually, most of BASIC and all ML routines and data are unaltered; NEW puts zero bytes at the start of BASIC, resets pointers, and performs a CLR, which

aborts files, among other things. OLD in Chapter 6 will recover BASIC after NEW (or after resetting by the method described in Chapter 5), provided new program lines haven't been entered.

2. NEW may sometimes generate ?SYNTAX ERROR. (See the error message notes.)

Examples:

1. **NEW**

   In direct mode, NEW readies the 64 for a new program. (Without NEW, the program would be simply added to the one currently in BASIC as extra or replacement lines.)

2. **20000 NEW: REM PROGRAM NO LONGER WANTED**

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

NEXT

Type: Statement

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

Modes: Direct and program modes are both valid.

Token: $82 (130)

Abbreviated entry: N SHIFT-E

Purpose: NEXT 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**

   This prints an unformatted multiplication table for values up to 10 × 10. 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 following NEXT statements 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 approximately ten seconds, unless a key is pressed; if it is, line 81 gets rid of the still active J loop.

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

   NEXT can appear anywhere, allowing clumsy constructions like the one above.

NOT

Type: Logical operator

Syntax: NOT *logical or numeric expression*

Numeric expressions must evaluate after truncating to −32768 to +32767.

Modes: Direct and program modes are both valid.

Token: $A8 (168)

Abbreviated entry: N SHIFT-O

Purpose: NOT 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 signed binary 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. So NOT of arithmetic expressions does not necessarily convert true to false.

Note: NOT has precedence over AND and OR. Thus:

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**

   This line flips a quote mode flag, denoting whether quote mode is on or off.
2. **IF NOT OK THEN GOSUB 20000: REM ERROR MESSAGE AT 20000**

   The above line 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 truncate to 0–255.

Modes: Direct and program modes are both valid.

Token: $91 (145)

Abbreviated entry: None

Purpose: ON allows a conditional branch to one of the listed line numbers, depending on the value of the expression after ON. If it is 1, the first line number is used; if it is 2, the second is used, and so on. If the value is 0 or too large for the list, the line is ignored and processing continues with the next statement. This provides a readable method of programming multiple IFs, provided a variable takes consecutive values 1, 2, 3, ...

Examples:

1. **ON SGN(X)+2 GOTO 100,200,300**

   The above line branches three ways, depending on X being negative, zero, or positive.
2. **90 ON ASC(IN$)−64 GOTO 100,200,300,400**

   This line jumps to one of the lines listed, depending on IN$ being A, B, C, or D.

3. **30 ON 6*RND(1)+1 GOSUB 100,200,300,400,500,600**
   This selects at random one of six subroutines in a game.
4. **100 ON X GOTO 400,410,420,430,440,450**
   101 ON X−6 GOTO 460,470,480
   Above is an example of how options can be spread over several program lines (provided X is not 0).

OPEN

Type: Input/output statement
Syntax:
Tape: OPEN *numeric expression [,numeric expression [,numeric expression [,string expression]]]*
The first numeric expression, the file number, must evaluate to 1–255; the second is the device number, which is 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 file number must be 1–255, the device number is usually 8, the secondary address is usually between 2 and 15, and the string expression a command like "SEQ FILE,W", which the disk drive itself, not BASIC, processes. Chapter 15 has full details.
Modems and other RS-232 devices: The same as for a disk drive, 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.
Printers and other write-only devices: These require file and device numbers. The string expression is irrelevant with these devices, while the third numeric parameter may or may not matter. See Chapter 17.
Tape and disk filenames can't exceed 16 characters.
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 external devices like tape or disk drives. For example, the statement:

```
OPEN 1,1,1,"TAPE FILE"
```

opens logical file 1, called "TAPE FILE", to the cassette. After this, the statement:

```
PRINT#1
```

followed by data will write the data to tape, and

```
CLOSE 1
```

leaves a complete new file called "TAPE FILE", which can be read back later, typically by OPEN 1 and INPUT#1,X$ or similar statements.
As many as ten files (enough for almost any purpose) can be open at once; each must have a different logical file number (the first parameter of OPEN) so that they

are distinguished. In addition, the secondary addresses of disk files must all be different. Three tables in RAM store the file numbers along with their device numbers and other information.

Note: Opening a file to tape blanks the screen during tape read/write activity. OPEN 1 in direct mode (valuable for reading a program header's information) pauses for ten seconds before returning to READY unless the Commodore key, space bar, or one of a few other keys is pressed. OPEN 1 in program mode does not result in a pause.

Examples:

1. **OPEN 2,1,0,"TAX"**

   The above example opens a file from tape called TAX (or TAXI, TAXIDERMY, etc.) for reading (since the third parameter is 0), and assigns it logical file 2, so INPUT#2 or GET#2 will fetch data from the file. 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. **OPEN 1,8,3,"ORDINARY FILE,S,R"**

   The line above opens a sequential file (specified by the ,S after the filename) on disk called ORDINARY FILE for reading (specified by the ,R after the filename) by INPUT#1 or GET#1 statements.

3. **OPEN 2,2,0,CHR$(6)**

   This is an OPEN which prepares the modem (device 2) for PRINT#2 and INPUT#2. The string expression is used to set modem parameters such as parity and data transfer rate.

4. **OPEN 4,4: REM OPENS FILE#4 TO DEVICE#4**

   This line opens a file to a printer, assuming that the printer (and interface, if one is used) operate like a standard Commodore printer.

OR

Type: Logical operator

Syntax: *Logical or numeric expression* OR *logical or numeric expression*

Numeric expressions must evaluate after truncating to −32768 to +32767.

Modes: Direct and program modes are both valid.

Token: $B0 (176)

Abbreviated entry: None

Purpose: OR calculates the logical OR of two expressions, by performing an OR on each of the bits in the first operand and the corresponding bits in the second. For the purposes of the OR comparison, numeric expressions are evaluated as 16-bit signed binary numbers. The four possible combinations of single bits are:

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. And it follows that:

380 OR 75 = 383

though verifying this by finding the binary arithmetic forms of 380 and 75 is somewhat tedious.

Examples:

1. **560 IF (A<1) OR (A>20) THEN PRINT "OUT OF RANGE"**
   This is a typical validation test; A must be a value from 1 to 20.
2. **POKE 328,PEEK(328) OR 32**
   The above POKE and PEEK combination 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 clear them to 0.

PEEK

Type: Numeric function

Syntax: PEEK(*numeric expression*)

The expression must evaluate to a number in the range 0–65535; the value returned will be in the range 0–255.

Modes: Direct and program modes are both valid.

Token: $C2 (194)

Abbreviated entry: P SHIFT-E

Purpose: PEEK returns the decimal value of the byte in a memory location. PEEK allows BASIC programs and their variables and pointers to be examined, plus other internal memory, like ML programs, the BASIC interpreter, hardware registers, and so on.

Notes:

1. PEEK (like POKE) is unusual in that it is easily replaced by ML routines. Chapter 17, for example, has ML routines to read joystick values, which are much faster than using PEEK in BASIC.
2. A number of locations are modified by hardware and therefore have values which vary: joystick, paddle, and keyboard locations are obvious examples. Some locations vary as a result of software modification; much of the memory area from locations 0–255 (called the *zero page*) is used by BASIC as it operates, and numerous locations in the range 256–1023 are used from time to time during BASIC execution. Most of the memory above 32768 can be reallocated for different uses and is largely controlled by the byte in memory location 1, as Chapter 5 explains.

Examples:

1. **PRINT CHR$(34);: FOR J=2048 TO 2147: PRINT CHR$(PEEK(J));: NEXT**
   This prints 100 characters PEEKed from the start of the 64's normal BASIC program storage area. (The quotation mark generated by CHR$(34) is an attempt to prevent spurious control characters from clearing the screen, etc.)
2. **500 IF (PEEK(653) AND 1)=1 THEN PRINT "SHIFT KEY"**
   This tests bit 0 of location 653—the byte that stores the SHIFT, Commodore key, and CTRL key flags—to determine if the SHIFT key is pressed.

POKE

Type: Statement

Syntax: POKE *numeric expression, numeric expression*

The first numeric expression is an address, and the second is a one-byte value, so the ranges must be 0–65535 and 0–255, respectively.

Modes: Direct and program modes are both valid.

Token: $97 (151)

Abbreviated entry: P SHIFT-O

Purpose: POKE stores the byte specified by the second expression into the address given by the first. POKE can store ML routines into memory from DATA statements, alter BASIC pointers, alter hardware registers, and perform other useful functions.

Notes:

1. POKE (like PEEK) can be replaced by simple ML routines; replacing a POKE to the screen with the ML equivalent is an ideal introduction to ML (see Chapter 7).
2. A careless POKE to an uninitialized variable, like:

   POKE A0,0

   in place of:

   POKE A,0

   will alter the direction register at location 0; this affects tape and the 64's RAM/ROM bank switching.

Examples:

1. **POKE 53281,9**

   This changes the screen background color by altering a VIC chip register.
2. Chapter 6 has a large number of programs which READ values from DATA statements, then POKE them into RAM.
3. **FOR J=0 TO 499: POKE 1024+J, PEEK(2048+J): POKE 55296+J,0: NEXT**

   This puts 500 bytes of BASIC program data onto the screen in black, by POKEing values to both screen and color memory.
4. **FOR J=40960 TO 49151: POKE J, PEEK(J): NEXT: POKE 1,54**

   This moves the BASIC ROM ($A000–$BFFF) into RAM, then switches that part of memory to RAM, so the BASIC interpreter is now held in RAM. Normally, POKE J, PEEK(J) has no effect, of course. The example works only because the 64 is designed so that POKEd information goes into RAM which is *underneath* ROM. (This means that either the usual ROM or alternative RAM may be used. This technique will not work with most other computers.)

POS

Type: Numeric function

Syntax: POS(*numeric expression*)

The numeric expression is a dummy expression, as with FRE.

Modes: Direct and program modes are both valid.

Token: $B9 (185)

Abbreviated entry: None

Purpose: POS returns the position of the cursor on its current logical line as seen by BASIC. Normally, POS(0) is 0–79, but some PRINT statements may return values up to 255. POS's usefulness is confined to the screen; it won't work with printers, for example.

Examples:

1.
```
90 FOR J=1 TO 100: PRINT W$(J-1)" ";
92 IF POS(0) + LEN(W$(J))>38 THEN PRINT
94 NEXT
```
The above program lines print the words in array W$ in a tidy format, without allowing wraparound to following lines. (This assumes that no string longer than 38 characters will be allowed.)

2. Chapter 9 contains a routine (to convert ML into DATA) which uses POS.

PRINT

Type: Output statement

Syntax: PRINT *[expression]*

The expression may be any type, separated by one or more of the following: SPC(*numeric expression*), TAB(*numeric expression*), space, comma, semicolon, or no separator (where this causes no ambiguity).

Modes: Direct and program modes are both valid.

Token: $99 (153)

Abbreviated entry: Question mark (?)

Purpose: PRINT evaluates and prints string, numeric, and logical expressions to an output device, usually the TV or monitor. The punctuation of the material after the PRINT statement affects the appearance of the output, which also depends on the internal character set being used. (See "PRINT USING" in Chapter 6, for information on ML formatting of numbers.)

Notes:

1. *Built-in graphics.* The entire character set can be printed, but {RVS} is necessary to complete the set using PRINT statements (POKE, of course, does not require the use of {RVS}). 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 TV into PRINT statements in a program. Homing the cursor, inserting spaces, and typing line numbers followed by ?" and RETURN will sometimes work. This method, however, won't accept reversed characters, so be careful when designing graphics directly on the screen. Chapter 12 has detailed information on graphics.

   Note that the SHIFT-Commodore key combination normally toggles between two different character sets, one with lower- and uppercase (good for text), and one with uppercase and extra graphics characters. Printing CHR$(14) selects the lowercase/uppercase set, CHR$(142) selects the uppercase/graphics set, CHR$(8) locks out SHIFT–Commodore key switching, and CHR$(9) enables SHIFT–Commodore key character set switching.

2. *User-defined graphics.* PRINT operates with ASCII characters, and their onscreen appearance is irrelevant, so user-defined characters can be handled by PRINT, too. In most cases, it's easiest to keep most characters as usual, so that program listings on the screen are readable. See Chapter 12 for full details.
3. *Punctuating PRINT:*
   - Expressions, as stated at the start of this chapter, are fairly straightforward. Numeric expressions can include numbers, TI, ST, the value π, and so on; string expressions may include TI$.
   - SPC and TAB allow the print position to be altered.
   - Commas tabulate output into the first, eleventh, twenty-first, or thirty-first columns. For example, try:

     ```
     PRINT 1,2,3,4,5
     ```
   - Semicolons prevent print position from skipping to the next line, and therefore act as neutral separators. Try this line:

     ```
     PRINT 1;2;3;: PRINT 4
     ```

     Remember that numbers are output with a leading space (in case there is a negative 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.
   - Colons end the statement, and in the absence of a semicolon move print position to the next line. The following line advances the print position two lines:

     ```
     PRINT: PRINT
     ```
   - Spaces (unless within quotation marks) are generally ignored, so:

     ```
     PRINT X Y;2 4
     ```

     does the same as PRINT XY;24.

Examples:

1. **PRINT X+Y; 124; P*(1+R%/100) :REM NUMERIC EXPRESSIONS**
   This prints three numbers on the same line. Notice, though, that if the first semicolon is left out, X + Y1 is printed.
2. **PRINT "HI " NAMES$ ", HOW ARE YOU?" :REM STRING EXPRESSION**
   The above line prints output on a single line (if there's room).
3. **FOR J=1 TO 20: PRINT J,: NEXT :REM SHOWS USE OF COMMA**
   This illustrates the tabbing effect of the comma in PRINT statements.

PRINT#

Type: Output statement

Syntax: PRINT# *numeric expression [,expression]*

There must be no space between PRINT and #; the numeric expression is a file number—the file must be open; and subsequent expressions should use format similar to PRINT.

Modes: Direct and program modes are both valid.

Token: $98 (152)

Abbreviated entry: P SHIFT-R (This includes #, and ?# *does not work.*)

Purpose: PRINT# sends data to an output device, usually a printer, tape, disk drive, or a modem.

Notes:

1. *Punctuating PRINT#.* The effect of punctuation in PRINT# statements is identical to PRINT, except for a few cases: Eleven spaces are always written after a comma, and expressions in TAB or POS will not work.

   PRINT#4,X$

   writes X$ followed by CHR$(13), but:

   PRINT#4,X$;:

   writes X$ alone.

   PRINT#128,X$:

   writes X$ followed by a carriage return *and* linefeed; this feature of files numbered 128 or more is useful with certain non-Commodore printers.
2. *PRINT# and INPUT#.* Remember that INPUT# cannot handle strings longer than 88 characters, so this limit must be observed when setting up data files with PRINT#.
3. *PRINT# and CMD.* PRINT#4,;: unlistens the device using file 4, while CMD4,;: leaves it listening, so these expressions are opposites. See Chapter 17 for full details on printers and modems.

Examples:

1. **OPEN 1,1,1,"TAPE FILE": INPUT X$: PRINT#1,X$: CLOSE 1**

   This opens TAPE FILE to tape and, after allowing the user to enter a string, writes the string to tape. Chapter 14 has full details on tape; Chapter 15 has information on disk files.
2. **100 FOR J=32768 TO 40959: PRINT #1,CHR$(PEEK(J));: NEXT**

   This prints the bytes in RAM or ROM in the plug-in area to file 1. Note the semicolon to prevent characters having a RETURN character following each one (making the file twice as long). The resulting file must be read back with GET#, since it is ML and not formatted for INPUT# to handle.

READ

Type: Statement

Syntax: READ *variable [,variable...]*

Modes: Direct and program modes are both valid.

Token: $87 (135)

Abbreviated entry: R SHIFT-E

Purpose: READ assigns data stored in DATA statements to a variable, or variables.
If the type of variable doesn't match the data (for example DATA "ABC": READ X), ?TYPE MISMATCH ERROR is printed when the program runs. ?OUT OF DATA ERROR is given when all the data has been read and the program has encountered an extra READ statement; a RESTORE statement is used to start reading data from the beginning again.

Examples:

1. **100 READ X$: IF X$<>"ML ROUTINE" GOTO 100**

   This shows how a piece, or group, of data can be found anywhere in the DATA statements. This construction (with RESTORE) allows data to be mixed fairly freely throughout BASIC program space.
2. **10 READ X: DIM N$(X): FOR J=1 TO X: READ X$(J): NEXT**

   The above line shows how string variables (for example, words for a word game) can be read into an array effectively, by putting the word count at the start, like this:

   1000 DATA 2,RED,YELLOW.

REM

Type: Statement

Syntax: REM *[anything]*

Modes: Direct and program modes are both valid.

Token: $8F (143)

Abbreviated entry: None

Purpose: REM allows documentation to be included in a program. Everything on the BASIC line after REM is ignored by the BASIC interpreter. REM statements take space in memory, and a little time to execute, so final versions of programs will generally have the REM statements removed.

Note: See the section on REM in Chapter 6 for some special effects. Chapter 7 explains how ML can be stored in REM statements.

Examples:

1. **GOSUB 51000: REM PRINT SCREEN WITH INSTRUCTIONS**

   Above is a sample REM comment.
2. **70 FOR J=1 TO 1000: REM MAIN LOOP**
   80 A(J)=J*A: NEXT

   This shows poor placing of REM, because the REM executes 1000 times. Move the REM to line 69 to increase speed.
3. **15998 REM ------------------------**
   15999 REM * SUB 16000 PRINTS TITLE *****

   These lines show how REM statements can be made easy to read in long programs.

RESTORE

Type: Statement

Syntax: RESTORE

Modes: Direct and program modes are both valid.

Token: $8C (140)

Abbreviated entry: RE SHIFT-S

Purpose: RESTORE resets the data pointer, so that subsequent READs retrieve data starting from the first DATA statement. NEW, RUN, and CLR all perform RESTORE as part of their functions.

Note: This command has no connection with the RESTORE key.

Examples:

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

   This example is the first line of a loop to read the same block of DATA continuously—perhaps the notes for a tune to be repeated. When ** is encountered as the last element of the data, the RESTORE resets the data pointer, and the line executes again.

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 27, 32, SUB, MLROUTINE1, 169, 0, 141, 202, 3, 162, 1, 160, 20

   This shows how data can be labeled to insure that the correct section is read; line 130 reads the data until it reaches the label MLROUTINE1, which is followed by the desired data.

3. **RESTORE: GOTO 100**

   This is a direct mode command of the sort helpful in testing programs which contain DATA statements, since after the RESTORE statement, all the data will be 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: RETURN transfers program control to the statement immediately after the most recent GOSUB statement. GOSUB and RETURN therefore permit subroutines to be automatically processed without the need to store return addresses in programs.

Notes:
1. See GOSUB for a full account of subroutine processing.
2. This command has no connection with the 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 the $.)

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

Examples:

1. **FOR J=1 TO 7: PRINT SPC(8−J) RIGHT$("AMAZING",J): NEXT**
   prints seven substrings of "AMAZING", aligned using SPC.
2. **100 PRINT RIGHT$("   "+STR$(N),10)**
   is another method for right justification; each string is padded with leading spaces, making a total length of ten.

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: RND 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. *RND's argument.* The argument used in the parentheses as part of the RND statement affects the way the number will be generated:
   - *Positive.* The value of the number is irrelevant. RND(1) and RND (1234) behave identically. The sequence of numbers generated is always the same, starting with .185564016 immediately after the computer is turned on.
   - *Zero.* This causes RND to take values from CIA timers; the result is more truly random, although short ML loops, for example, may show repetitiveness.

- *Negative.* The random number is reseeded with a value dependent on the argument. A negative argument always returns the same value; RND (−1) is always 2.99 E−8, for example. Chapter 8 has information on RND and explains why negative integers give very small seed values.
- *Programming with RND.* During program development with random numbers, start with, say, X=RND(−1.23) to seed a fixed value, then use RND(1) while testing the program, which will always follow the same sequence. The final version of the program might use X=RND(0) to start seeding with a random value.

2. To obtain a random number between A and B (but excluding the exact values of A and B), use:

   A + RND(1)*(B−A)

   For example:

   −1 + RND(1)*2

   generates numbers between −1 and +1.

   Integers are equally simple. To obtain an integer value between A and B (*including* the exact values A and B), use:

   A + INT(RND(1)*(B−A+1))

   For example:

   1 + INT(RND(1)*10)

   generates integers from 1 to 10 with equal probability.

Examples:

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

   The above line causes a random delay of up to roughly three seconds.
2. **100 RESTORE: FOR J=0 TO 100*RND(1): READ X$: NEXT**

   This example reads a random number of items from DATA statements (between 1 and 100); thus, the last item read into X$ will be retained as a randomly selected string. This could be used perhaps to choose a word for a language test from a list of 100 data items.
3. **1000 IF RND(1)<.1 THEN PRINT "A VERY GOOD DAY TO YOU"**

   This line 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**

   This technique is useful in generating test data. The construction generates an array holding N strings, of random lengths between 1 and 10 characters.
5. **ON RND(1)*4+1 GOSUB 200,300,400,500**

   The above line selects one of the four subroutines at random.

RUN

Type: Command

Syntax: RUN *[line number]*

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: RUN executes a BASIC program in memory, either from its beginning or from a line number. In effect, RUN starts by executing a CLR, so variable values are lost; GOTO *[line number]* retains variable values.

Notes:

1. RUN doesn't execute a LOAD. The program must be read into memory beforehand.
2. ?SYNTAX ERROR as the result of a RUN means the start-of-BASIC pointers have been altered. See the information about CLR.
3. Chapter 8 shows how to run BASIC with ML.

Examples:

1. **RUN**
   RUN 1000
   These are two straightforward direct mode examples of the RUN command.
2. **IF LEFT$(YN$,1)="Y" THEN RUN**
   The above example is for use after:
   INPUT "ANOTHER RUN";YN$
   This starts the program from scratch after Y, or YES, is typed in.

SAVE

Type: Command

Syntax: SAVE *[string expression [,numeric expression[, numeric expression]]]*

Identical to that for LOAD. The interpretation of the final parameter is different, however, when using a tape drive: 0 allows a relocating LOAD, so a BASIC program can work whatever its starting 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 full details on saving to tape, 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 the BASIC program in memory to tape or disk, so the program is stored for future use. Programs must be saved to disk by name, but tape programs need not have names (although names can be useful in identifying tape contents).

ML, graphics characters, and other continuous blocks of RAM can be saved, too. Only two pointers have to be changed (in four memory locations), effectively redefining the position of BASIC's program area. The pointers are locations 43 and 44 (start), as well as 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 data in a very compact form, and 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 like this, too—see Chapter 12. (In each case only the pointer in 45 and 46 need be altered before saving.)

Note: 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
   Above are two BASIC program SAVE commands for use with tape. (SAVE with forced LOAD address is generally used only with ML, where the correct location of the program in memory is essential.)
2. **SAVE "PROGRAM"+TI$,8**
   Here is a sample disk SAVE command. This adds a clock value to keep track of several versions of a given program. The third parameter is ignored when saving to disk; there is no SAVE with forced LOAD address for disk.

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: SGN computes the sign of a numeric expression; it yields −1 if the expression is negative, 0 if it has a value of zero, and +1 if the expression is 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**
   This statement causes program execution to branch to 400 if X is negative, to 600 if X is 0, and to 800 if X is positive.
2. **FOR J=−5 TO 5: PRINT J;SGN(J);SGN(J)*J;SGN(J)*INT(ABS(J)): NEXT**
   This example prints several results; the last 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: SIN returns the sine of the numeric expression, which is assumed to be an angle in radians. (Multiply degrees by π/180 to convert to radians.)
See ATN for the converse function.

Examples:

1. **FOR J=0 TO 90: PRINT J SIN(J* π/180): NEXT**
   The above line 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 calculates the x and y coordinates of a geometrical shape.

SPC(

Type: Special output function

Syntax: SPC(*numeric expression*)
SPC appears only in PRINT and PRINT# statements. The numeric expression must evaluate to a value in the range 0–255.

Modes: Direct and program modes are both valid.

Token: $A6 (166)

Abbreviated entry: S SHIFT-P (This includes the open parenthesis mark.)

Purpose: SPC helps format screen or printer output. The name is misleading: With a TV or monitor, 0 to 255 cursor-rights can be printed, but these aren't spaces. Try the following:

```
PRINT SPC(200)"HI!"
```

However, with any other device, spaces are output, since cursor-rights are not in the ASCII system. TAB is exactly the same except that it moves from the leftmost column, rather than moving from the current position.

Examples:

1. **100 PRINT "{HOME}";: FOR J=0 TO 21: PRINT "X" SPC(38) "X";: NEXT**
   This prints a border down each side of the screen, without disturbing the screen characters between the borders.
2. **90 OPEN 1,3: CMD 1**
   Add this line to the previous; note how an open file causes spaces, not cursor-rights, to be output.

SQR

Type: Numeric function

Syntax: SQR(*numeric expression*)
The numeric expression must be positive, or an ?ILLEGAL QUANTITY ERROR will result.

Modes: Direct and program modes are both valid.

Token: $BA (186)

Abbreviated entry: S SHIFT-Q

Purpose: SQR calculates the square root of a positive argument. This is a special case of the power (up-arrow) function. SQR actually works faster than X↑.5, though, and is also more familiar to many people.

Examples:

1. **PRINT SQR(2) :REM PRINTS 1.41412356**
   This prints the square root of 2.
2. **X1=(−B + SQR(B*B − 4*A*C)) / (2*A)**
   X2=(−B − SQR(B*B − 4*A*C)) / (2*A)
   These are *both* solutions of the equation $AX^2 + BX + C = 0$.

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: ST indicates the status of the system after any input or output operation to tape, disk, or other peripheral. ST is set to 0 before GET, INPUT, and PRINT as well as CMD, GET#, INPUT# and PRINT#, so ST is rather ephemeral; where it is used it should be used after every command.

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, so ST=66 combines the conditions indicated by 64 and 2.) Chapters 14, 15, and 17 provide details on ST with tape units, disk drives, and modems, respectively.

Table 3-1. Status Variable (ST) Values

	Tape		Modem	Serial Bus (e.g., Disk)	
ST	Read	Write		Read	Write
1			Parity error		Print time-out
2			Framing error	Input time-out	
4	Short block on input		Rx buffer full		
8	Long block on input		Rx buffer empty		
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 disks is stored in location 144; ST for RS-232 devices is held in location 663. ST (like TI and TI$) is checked when a variable is set up; normally, no ST variable exists in RAM, and ST is processed by special routines. ST isn't a tokenized keyword or even a normal variable; this is why BEST=2 is accepted, and means BE=2, despite the apparent presence of ST.

ST (like TI and TI$) can be POKEd to any value in the legal range. ST can be used from ML. See Chapter 8, which deals with Kernal routines for information on this and on methods for reading errors from the disk drive.

Examples:

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

   The above line opens a file to a nonexistent device: This sets ST=−128.
2. **150 INPUT#8,X$: IF ST=64 GOTO 1000**

   This is a typical end-of-file check, for use 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 RUN/STOP key, the STOP statement returns the program to READY mode and prints a BREAK message showing the line number at which the program stopped. Like END, STOP can set breakpoints in BASIC, but it's better 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
```

This is typical of a test for keypress, which allows a program to be stopped at a particular point.

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 the $.)

Purpose: STR$ converts any floating-point number into a string, so that the number can be edited. 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**

   This line prints 1 as 1.0, 2 as 2.0, and so forth.
2. **PRINT "0" + MID$(STR$(X),2)**

outputs X as "0.57", etc., where X is between 0.01 and 1.0; MID$ and STR$ together remove the leading space. (Remember that numbers from 0 to 0.01 are output in exponential notation.)

SYS

Type: Statement

Syntax: SYS *numeric expression*

The expression must evaluate to a number between 0 and 65535.

Modes: Direct and program modes are both valid.

Token: $9E (158)

Abbreviated entry: S SHIFT-Y

Purpose: SYS transfers control to ML at the address following SYS. The ML is executed and will return to BASIC and execute the statement after SYS if the machine language routine ends with an RTS instruction (or the equivalent). The registers A, X, Y, and SR are loaded with values from locations 780–783 by SYS, and their values after the subroutine call are replaced in 780–783. This offers a useful way to check short ML routines for errors.

Notes:

1. Chapter 7 introduces ML programming on the 64; Chapter 6 has many examples which use SYS. Many of them end with a DATA value of 96, which is the decimal value of RTS. A jump to a ROM subroutine ending with RTS (DATA 76,XX,XX) also works, and RTI is sometimes used, too (a decimal value of 64).
2. Careless SYS calls may crash or corrupt BASIC, and perhaps cause odd results sometime later in the same programming session. Try:

   SYS 47175

   as an example. (This sets the decimal flag in the chip.)
3. ROM occupies locations 40960–49151 and 57344–65535 in the Commodore 64, unless one or both ROMs have been switched out (Chapter 5 explains this). Usually, SYS calls into these regions have repeatable and predictable effects with the 64, as explained in Chapter 11. Such calls will not work with other computers, which means they are *machine-specific*.

Examples:

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

   The above SYS calls ML stored *within* BASIC; this form works regardless of BASIC's start address. Chapter 9 explains these techniques in depth.
2. **SYS 64738**

   This well-known ROM routine call resets the 64 as though it were turned off, then on again. Chapter 11 lists ROM call addresses, and some (such as the screen routines in Chapter 6) are listed elsewhere.
3. **POKE 780,ASC("$"): SYS 65490: REM KERNAL ROUTINE**

   This puts the dollar character in the storage location for A, then calls a Kernal output routine at $FFD2. (Kernal routines are explained in depth in Chapter 8.) The effect is to print a dollar sign.

TAB(

Type: Special output function

Syntax: TAB(*numeric expression*)

TAB appears only in PRINT or PRINT# statements. There must be no space between B and (and the expression must evaluate to a number in the range 0–255.

Modes: Direct and program modes are both valid.

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

Purpose: TAB prints an expression at the desired position on the line (values between 0 and 255) specified by the parameter, unless this position is to the left of an earlier TAB in the same PRINT statement (like typewriter TABs, TAB doesn't work from right to left).

Note: TAB is nearly identical to SPC; the difference is that TAB subtracts its current position on the line from the TAB value, then issues that number of moves right. TAB's use of cursor-rights and spaces is the same as SPC's. In reverse mode, cursor-rights show as square brackets.

Example:

```
FOR J=1 TO 10: PRINT J TAB(4)J*J TAB(10)J*J*J: NEXT
```

The above example produces a tabbed 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: TAN 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 errors 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
```

This converts any radian measurement into its equivalent 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, except that no value may be assigned to TI (TI=X is never allowed). TI$ = *string expression of length 6* is allowed, but expressions with more or fewer than six characters cause an error.

Modes: Direct and program modes are valid.

Token: Not applicable
Abbreviated entry: Not applicable
Purpose: TI and TI$ access the internal software clock. It's kept running by BASIC as a normal part of its operation. A feature known as an *interrupt* operates this (and allows access to the keyboard); about every 1/60 second, locations 160–162 are incremented, and their collective value is used to obtain TI and TI$. See Chapter 5 for information on the hardware side of this and the end of Chapter 8 for programming information. (The interrupt rate can be changed by POKEing 56324 and 56325 with different values.) The clock isn't all that reliable; tape operation makes it run much faster than usual, and any programs which disable (turn off) interrupts stop it. The 64 does include two time-of-day clocks which are completely accurate to 1/10 second. Chapter 5 explains their use.

The maximum value for TI is 5184000, the number of 1/60 second intervals in a day. TI is equal to:

```
65536*PEEK(160) + 256*PEEK(161) + PEEK(162)
```

where the last of the three bytes changes fastest. Try writing a loop that repeatedly PEEKs location 162 for a demonstration. The easiest way to change the clock setting is with the statement:

```
TI$="101500"
```

The above line would set the clock to a quarter after ten. Note that ML can be used to set and read TI$; Chapter 8's section on Kernal ROM routines gives full information.
Note: Like ST, these variables are intercepted by BASIC, not set up in the normal variables section of memory located above BASIC program space. TIME and TIME$ are treated like TI and TI$, but ANTIC is treated as AN, and the characters TI are ignored.
Examples:
1. **50 TI$=HH$ + MM$ + SS$**
   This program line combines three previously entered two-digit strings into TI$.
2. **T$=TI$: PRINT MID$(T$,1,2) + ":" + MID$(T$,3,2) ":" + MID$(T$,5,2)**
   The above line prints TI$ in the format 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).

3.
   ```
   10 DIM T1,T2,J: T1=TI
   20 FOR J= 1 TO 1000:NEXT
   30 T2=T1-58 :REM FIGURE MAY VARY A BIT
   40 PRINT T2/60 "THOUSANDTHS OF SEC"
   ```
   Shows how individual BASIC operations can be timed. The first line insures that T1, T2, and J are placed at the start of variables; the number in line 30 must be set so that the program as it stands prints a value of 0. This means that any *additional* commands put in the loop in line 20 are timed exactly. You'll find that:

   ```
   POKE 7680,123
   ```

   takes 8/1000 second, a colon takes about 1/10,000 second, and so on. See Chapter 6 for more on this topic.

USR

Type: Numeric function
Syntax: USR(*numeric expression*)
Modes: Direct and program modes are both valid.
Token: $B7 (183)
Abbreviated entry: U SHIFT-S
Purpose: USR allows you to define a function in machine language. This requires a thorough understanding of ML; in BASIC, it's nearly always easier to use a DEF FN expression, and not much slower. In Chapter 8, the section on calculations has a complete explanation of this function with examples.

VAL

Type: Numeric function
Syntax: VAL(*string expression*)
Modes: Direct and program modes are both valid.
Token: $C5 (197)
Abbreviated entry: V SHIFT-A
Purpose: VAL converts a string into a number, so calculations can be performed on it. If the string is not a valid number representation, as much as possible is converted, and the remainder ignored with no error message. Valid characters are spaces, signs, numerals, unSHIFTed E, and periods in certain combinations. VAL is the converse of STR$.
Example:

```
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: VERIFY *[string expression [,numeric expression[, numeric expression]]]*
Identical syntax as LOAD. Syntax should match that of the LOAD or SAVE statement preceding VERIFY.
Modes: Direct and program modes are both valid.
Token: $95 (149)
Abbreviated entry: V SHIFT-E

Purpose: VERIFY reads and compares a BASIC or ML program from disk or tape with the program already in memory. If they aren't identical, ?VERIFY ERROR is reported. When VERIFY is used at all (it often isn't), it's generally to verify that a program has saved correctly. (It can be used in program mode, so a program can 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 most data files, since these cannot be loaded like programs.

Examples:

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

   The above commands save a program to disk, then verify that it has been saved correctly.
2. **10 PRINT "REWIND TO VERIFY"**
   20 GET X$: IF X$="" GOTO 20: REM WAIT FOR KEY PRESS TO VERIFY
   30 VERIFY

   At the start of a tape program, this verifies in program mode.
3. **LOAD "VERSION6",8**
   VERIFY "VER 6",8

   If you have two programs which you believe *may* be identical, VERIFY will compare them. OK means that your two programs are indeed identical. This is often useful when a disk contains lots of versions of a program, including security duplicates, saved during program development.

WAIT

Type: Statement

Syntax: WAIT *numeric expression, numeric expression [,numeric expression]*

The first parameter is an address (in the range 0–65535); the others are in the range 0–255 and will be converted to integers. The optional third parameter defaults to 0.

Modes: Direct and program modes are both valid.

Token: $92 (146)

Abbreviated entry: W SHIFT-A

Purpose: This statement waits until one or more bits of the memory location are cleared (given a value of 0) or set (given a value of 1) 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 parameter. If all bits are still 0, 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 work on other computers. In fact, they're often 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 (EOR is the 6502 mnemonic, so we'll use it as an abbreviation). Its truth table is:

   ```
   0 EOR 0 = 0
   0 EOR 1 = 1
   1 EOR 0 = 1
   1 EOR 1 = 0
   ```

   If both bits tested are the same, the result of an EOR is 0. If the bits are different, the result is 1. To put it another way, EOR is true if *either but not both bits are set.* The statement:

   WAIT address,a,b

   first EORs the byte in *address* with *b*. This allows any bit, or bits, to be flipped (set bits will be cleared and clear bits will be set). The result is ANDed with *a*, which allows any bit to be turned off (in this case, ignored). Since a zero result makes WAIT loop again (continue waiting), we can select *a* and *b* to respond so that any single bit changing either to on or off, can cause the program to exit from WAIT and execute the next statement. In the special case:

   WAIT address,a

   there is no EOR parameter. (Actually, the value in *address* is EORed with a zero byte, so the result is always the same as the original value in *address.)* Thus:

   WAIT address,16

   waits until bit 4 is set. 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**

   This line causes the computer to wait until the jiffy clock TI counts to 16 (about 1/4 second).
2. **100 POKE 198,0: WAIT 198,1**

   This example waits for a keypress (until one character is in the keyboard buffer).
3. **WAIT 56321,32,32**

   This waits until bit 5 of location 56321 is off. This happens when the Commodore key is pressed.
4. **10 WAIT 53265,128: POKE 53281,RND(1)*16: GOTO 10**

   The above line waits until bit 8 of the raster line becomes 1, indicating bottom of screen is reached, before changing screen color. Chapter 12 has more examples of these techniques.

BASIC Error Message Dictionary

(Disk error messages are handled separately from BASIC: see Chapter 15.)

?BAD SUBSCRIPT

The value given an array subscript is negative, or larger than that in the DIM statement (larger than 10 if the array has not been explicitly dimensioned). This message is also given if the wrong number of subscripts is used.

?BREAK

The RUN/STOP key was pressed before LOAD or SAVE was complete.

?CAN'T CONTINUE

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

- The program halted due to a SYNTAX ERROR, instead of the RUN/STOP key, STOP, or END;
- CLR has erased its variables;
- the program was edited after it stopped, effectively erasing variables;
- a direct mode error occurred, which the system can't distinguish from a program error; or
- the program has not been run.

?DEVICE NOT PRESENT

This means the printer, disk drive, or other device does not respond, typically on GET# or INPUT#, because it is unplugged, off, addressed by a wrong device number, or is nonstandard and unresponsive. Also, this error occurs when an end-of-tape marker is 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. 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 this is caused by the inclusion of commas or colons in an input string; avoid this with leading quotes.

?FILE DATA

The type of data in a file doesn't match the variables to which it is assigned by GET# or INPUT#. This happens when INPUT#X tries to read a string.

?FILE NOT FOUND
A disk file or program is not present on current disk, or the name is misspelled. (Tape gives ?DEVICE NOT PRESENT if the end of the tape is reached before the specified file is found.)

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

?FILE OPEN
This 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 three or more parenthesized subexpressions. String storage (the string descriptor stack at $19–$21) is exhausted. For example, PRINT "A"+("A"+("A"+"A")) will give this error.

?ILLEGAL DEVICE NUMBER
This means either that a command has been issued to an unacceptable device, like saving to the keyboard or loading from the screen, for example, or that the tape buffer has been moved below $0200.

?ILLEGAL DIRECT
This indicates that a statement requiring the input buffer has been entered in direct mode, typically GET or INPUT, or that 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. Attempting a POKE with either parameter negative, using a logical file number greater than 255, and asking for the square root of a negative number are examples.

I/O ERROR (1–9)
These are Kernal error messages, only visible in BASIC after executing POKE 157,64. See Kernal notes in Chapter 8.

?LOAD
A tape or disk program was not loaded successfully. See Chapter 14 (tape) or Chapter 15 (disk) for information on how to read the status byte to determine the cause of the error.

?MISSING FILE NAME
LOAD and SAVE must include a program name when using the disk drive.

?NEXT WITHOUT FOR

This message is given when the interpreter cannot find a FOR entry on the stack corresponding to the NEXT it has just encountered. This may happen if one of the following conditions exists:

- The stack has no FOR entries on it at all, because more NEXTs than FORs have been encountered;
- the variable in the NEXT statement is misspelled and doesn't match any FOR entries on the stack;
- the required FOR entry has been flushed from the stack by an incorrectly ordered NEXT in a nested loop; or
- an active GOSUB exists, as in:

```
10 FOR J=1 TO 20: GOSUB 100
100 NEXT
```

?NOT INPUT FILE

Given in response to an attempt to INPUT# or GET# from a file opened to be written to. For example, a tape data file opened in write mode cannot be read.

?NOT OUTPUT FILE

An attempt has been made to PRINT to an input file. A disk file opened in read mode 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 cannot act as an output device.

?OUT OF DATA

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

?OUT OF MEMORY

This message indicates one of the following:

- The 64 does not have enough RAM for the program and its variables (especially if dimensioning large arrays or inputting long strings);
- temporary storage on the stack has run out, having been filled with GOSUBs (about 24 maximum), FOR-NEXT loops (about 10 maximum), and intermediate calculation results:

```
PRINT (1+(2+(3+(4+(5+(6+(7+(8+(9+(10+(11+(12))))))))))))
```

- the end-of-program pointer in locations 45 and 46 has been set (perhaps by a LOAD into high memory) greater than the end-of-BASIC-storage pointer in 55 and 56. (Reset the pointer or use NEW to correct this.)

?OVERFLOW

The value of a calculation is outside the valid range for floating-point numbers, approximately −1.7E38 to +1.7E38. If a result is within the valid range, this error may be avoidable by restructuring the computation using, for example:

```
PRINT (5/4)↑100
```

instead of:

```
5↑100/4↑100
```

?REDIM'D ARRAY

An attempt has been made to dimension an array that has already been dimensioned. It may have been dimensioned automatically. A reference to X(8), for example, implicitly performs DIM X(10) if the array doesn't yet exist in memory.

?REDO FROM START

This message is 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 0 to 255 characters; this error is given if a string expression evaluates to a string longer than this. This message is output when an attempt has been made to read a string 89 or more characters long into the input buffer, typically by INPUT#.

?SYNTAX

This indicates that a BASIC statement is unacceptable. There are many causes. The 64 anticipates a sequence of statements; if a statement doesn't 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 often results. This message is also given after NEW if the first byte of BASIC is nonzero.

```
POKE PEEK(44)*256,0: NEW
```

typically avoids this error.

?TOO MANY FILES

This is given in response to an OPEN statement if ten logical files, the maximum number, have already been opened.

?TYPE MISMATCH
This message is output 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
The program in memory isn't identical to the disk or tape file it is being compared with by VERIFY. Spurious VERIFY errors occur if BASIC programs are loaded into 64 memory at different addresses from where they were saved; the link pointers between lines are different, but the BASIC statements may be the same.

Chapter 4

Effective Programming in BASIC

- How to Become Fluent in BASIC
- Programs, Systems, and People
- Program Design
- System Design
- Serious and Less Serious Programming
- Debugging BASIC Programs
- Examples in BASIC
- Making BASIC Run Faster

Chapter 4

Effective Programming in BASIC

How to Become Fluent in BASIC

BASIC is a language, with its own vocabulary and syntax, which requires a certain amount of creativity on the programmer's part to get good results, just as the ability to write novels or articles requires more than only a knowledge of words and syntax. The challenge for a novice programmer is to develop style and fluency.

Perhaps the best way to learn to write is to read *and write* a lot; similarly, the best way to learn BASIC, once you know the vocabulary, is to examine other people's programs and adopt good techniques while developing your own style. At first, use only a few BASIC statements in your programs, and limit your attempts to small tasks. As you gain more experience, you can add new BASIC words to your vocabulary. There are, however, some BASIC statements that you may never need to use, just as there are probably words in your native language that you have never spoken.

When writing programs, you have an advantage over writers who don't use BASIC; you can experiment freely and find out immediately whether your way of expressing your intentions is acceptable or not. If you fail, no harm is done, provided the experiments are kept away from your working programs and important data. With this in mind, it makes good sense to try several approaches to any programming task.

No matter what kind of programming you would like to do in BASIC, it is important that you have an appreciation of the capabilities and limitations of your computer, BASIC, and the intended user. This is largely a matter of experience. The remainder of this chapter consists of advice and information that may help you write reliable and easy-to-use programs. The final programming decisions, of course, are always yours.

Programs, Systems, and People

Before considering program design, we'll overview the software market and the attitudes of software producers and users. First, there are three main program types:

Autonomous, stand-alone programs. These don't depend on other programs, but stand alone. These programs contain all graphics and data necessary to function properly without support. Most games are like this. "Diet Calculator" (later in this chapter) is an example of an autonomous program.

Program systems (groups of programs). These generally have many programs and 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" (below) is a simple system which the 64 can run.

Pseudosystems (programs resembling systems). Single programs with a family resemblance to each other might be classified as midway between autonomous programs and systems. For example, multiple-choice and other educational programs collectively can be regarded as systems.

The concepts are important here, not the names. Systems are likely to be more difficult to program than autonomous programs, needing validation and checks unnecessary in the other types. Programs that resemble systems are likely to be easy to program, provided standardized methods have been developed.

Second, there are different types of users. Microcomputer owners can generally be classified as business, scientific, educational, or personal users.

Business. The 64, with disk drives and a printer, is capable of handling data in moderate quantities, where speed isn't crucial in day-to-day organization. Mailing lists, telephone lists, customer information, billing notices, small financial calculations using spreadsheets, and letter writing on word processors are examples of the uses for the 64 in small businesses.

A number of problems exist, though, such as unacceptable slowness. People who purchase a system program may not be able to describe accurately the features they want, or understand that new features, like fast searches and sorts, may be impossible or may require the entire system to be completely rewritten. The office staff may not be willing to key in data, particularly if instructions aren't provided in nontechnical language. And there will be problems if programs don't have thorough error trapping, or if correction and updating of information is difficult. There may be security problems in addition to these concerns, since most businesses keep records (even interoffice memos) that are not intended for all to see. This is perhaps a rather negative picture, but programmers should keep these possible difficulties in mind when designing systems.

Commercially sold software packages may be inadequate for several reasons. First, published reviews are unlikely to be of much value, because comprehensive testing takes months of work. Second, software packages are continually under development, and a purchaser may be unable to establish how recent and reliable the version being shown is. Third, there may not be a commercial program system that will do what the particular business needs. Still, for many small businesses, commercial software will be adequate, and programmers should try to include similar features to those of successful programs in their own works, improving them as much as possible.

Scientific. Controlling external hardware for monitoring experiments or controlling equipment (like large computer-operated telescopes) is a specialized area (see Chapter 5 for more information on software). Calculations and simulations are the other uses for which micros are suitable. Desk-top computers are often used to solve complex equations; anything with a definite formula is a potential task for a computer program, from architectural stress calculations to zoo nutrition, provided the computer's memory is large enough to hold the data.

Educational. The continued drop in the price of computing (apparent rather than real in many cases, after making allowance for separate disk units, printers, RAM expanders, and so on), plus skillful marketing (playing on parents' fears that their children might miss out on the computer revolution), have created a boom in computer education. Nevertheless, in spite of the huge sums paid for education, not that much is spent on computers. And it is unreasonable to expect great computer expertise from teachers who haven't themselves been trained.

A common situation has evolved where classes have one model of an expensive computer, while the students who have one at home own a less expensive or different computer. There are two different attitudes toward microcomputer education, broadly dividing people who do not have programming skills and those who do. The first group sees, with some relief, a roomful of unruly children settle down to play a number game, and thinks, "It is remarkable to see them working in an orderly way for hours. Increased equality of education is possible. Unmotivated students find their interest reawakened, and their confidence grows."

The other group's argument is, "Computers are unparalleled at teaching logical thought. They provide great opportunities for students to display their creativity, and this may be the most important part of their schooling." When the experts disagree, it is hard to know what makes good educational software.

Multiple-choice tests, with question-and-answer programs, graded 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, without the need to interpret a verbal answer. To discourage guessing, wrong answers could score −1/4 point, so completely random answers would score around zero.

Single-concept programs, like children's counting programs and alphabetic-recognition programs, are becoming available commercially. Good graphics can add a lot of appeal and help to hold the user's attention longer. 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 concepts like graph plotting, limits, sums of series, calculus, and simulations of randomness with coins, roulette, and so on. (See "Dice" page 97.)

Personal. This rapidly expanding area of the market includes games and educational and home business programs. Several magazines are currently being published which include programs in the magazine that can be typed in at home, and therefore are practically free.

Program Design

We've distinguished programs from systems, and noted that systems require more planning and knowledge; in the commercial world this is reflected in the job separation between analysts and programmers. On the relatively modest scale of the 64, it's equally true; experienced programmers can almost unconsciously plan ambitious projects out of reach of beginners. This section covers the sort of thought processes necessary in programming and in design, with a concrete example of each to give substance to the generalities. Bear in mind that many programmers write in an unorganized, ad hoc fashion and don't always worry about tidy, theoretical schemes. If your programs are messy and patched together, don't worry too much—many other people's programs are, too.

Program Example: Number Guessing Game

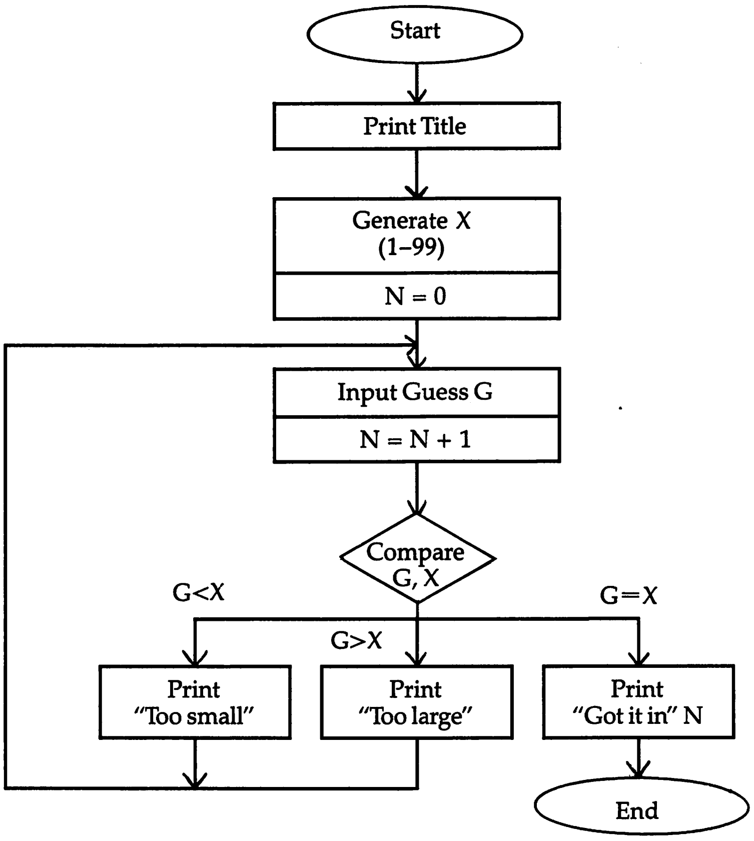
We'll write a program which thinks of a number from 1 to 99, then accepts guesses typed into the keyboard. Where the guess is wrong, it prints TOO LARGE or TOO SMALL, as the case may be. Correct input is rewarded by an encouraging message plus the total number of guesses. Putting this into BASIC requires four steps, which may be formally written down or simply carried out mentally, but always take the form outlined below.

Understand the problem. The example is quite simple: Many computer problems are not.

Express it in a computerizable way. 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. If you haven't understood that the computer has to count lines of print to know where it is on a page, you won't be able to print titles on page tops. Knowledge of the logic of programming equips you with methods and tricks to process data, but experience is probably the best way to learn the physical limitations and capabilities of a particular computer.

This flow chart expresses an approach to our game in a form that can be written as BASIC. Entries in the boxes are shorter than usual to avoid clutter. You should be able to trace how the variable, N, records the number of guesses, and how all three possible outcomes of the comparison between the correct number, X, and the current guess are processed.

Figure 4-1. Number Guessing Game Flow Chart



Flow charts generally use diamonds to indicate options and rectangles for operations; direction of flow is usually down, with loops and branches generated by the options arranged clockwise, as in this diagram. Many other, less common symbols are also used. Virtually all programs have loops and decision points, and flow charts show these clearly. However, they are hard to modify and they take up space, so many people prefer to make outlines and notes—stylized lines of English resembling programs. There's no correct notation; and the sad fact is that any complex program remains complex in whatever way it is written down.

Write it in BASIC. If it's a complex program, write parts of it and test them individually as subroutines. 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.

Algorithms are rules with explicit instructions and no exceptions, which generate correct results. Math algorithms can be used by anyone, without understanding any of the underlying theory. For example, linear programming (solving such problems as finding the least expensive combination of foods which supply all known nutrients) involves long calculations, which give the right answer. At a simpler level, arranging dates in the format YYMMDD makes them sort numerically into chronological sequence, while MMDDYY requires more work to sort properly. A version of 3-D tic-tac-toe requires the winner to avoid a line; the algorithm *start in the center and make opposite moves* always wins for the first player.

Algorithms can be used to deal with very complex situations: often the rule is found to give good results and is therefore used in lieu of anything better. Warnsdorf's rule in chess, to generate knight's tours around the whole board, illustrates this. The rule is: Move the knight to the square with fewest exit squares. This often (but not always) gives a solution. Many games—bridge, for example—are in effect often played algorithmically, as the players follow rules that sum up the experience of good players. Chess openings can be generated with simple algorithms as well; a common example is moving to maximize the area under attack by your pieces, while minimizing the opponent's range of replies. Reversi (or Othello™) played on an 8 × 8 board, with pieces white on one side and black on the other, can be played by the following simple algorithm: For about 10 moves, occupy central squares, reversing as few of the opponent's pieces as possible; for another 10 moves, keep the total number of your pieces to about ten; after this, go for maximum points.

Our number game, Program 4-1, is too simple to require such intricate algorithms, though.

Program 4-1. Number Guessing Game

```
10 PRINT "{CLR}GUESS MY NUMBER (1-99)": PRINT
20 X=INT(RND(1)*99)+1: N=0
30 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"
```

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.

Test and improve. Our example could include:

```
70 FOR J=1 TO 3000: NEXT: GOTO 10
```

effectively replacing the box END with a delay loop and a branch back to the PRINT TITLE box. Values could be checked to insure that they are integers in the correct range, and you could add color.

Testing is difficult, and many commercial programmers spend most of their time removing bugs from programs. Ideally, with good planning, bugs would not appear, but in practice it's seldom possible to foresee every potential problem.

System Example: Wordscore Analysis

"Wordscore" is a demonstration program (see the end of this chapter under string handling) which evaluates five-letter words on a score-per-letter basis. In other words, each letter is assigned a value, and the value of a word is the sum of the values of all the letters. The letters B-I-N-G-O must be included vertically, horizontally, or diagonally to form an acceptable combination of words.

The first step toward finding an algorithm is to consider the potential data base of words, and several assumptions will provide the basis for this algorithm. A typical dictionary lists about 2000 five-letter words (based on a sampling of pages), so only about 400 can be expected to exist which contain one or more letters of BINGO. The Commodore 64, even without disk or tape, easily has enough memory to store 400 short words. The demonstration shows how values can be assigned to letters A to Z before checking the words, stored on a tape or disk file. Using BASIC to select the highest-scoring words in the form Bxxxx, Ixxxx, and so on (there are 29 relevant formats) isn't difficult. The conclusion is that the 64 could be valuable for this application. (Other factors—acceptable dictionaries, competition rules—are likely to complicate matters.)

System Design

In addition to the normal programming concerns, system planning takes three major steps, which are discussed below.

Ask if the project is feasible. Time may be a problem; sorts, searches, graphics, and tape processing may be too slow; the program's response time may be inadequate; the data may take too long to key in. You may want to write a test program to check the feasibility of the task. Machine language always outperforms BASIC, but is often more difficult to program.

Generally, if much data is to be processed, estimate the total storage needed in bytes and estimate whether it can coexist with BASIC, or whether stored files would help. Perhaps splitting a program into smaller subprograms would be advisable. Less tangible problems might be user attitudes, reliability, and recovery of lost data if problems should occur. A little time spent in advance on all these questions is usually worthwhile. Even so, there will be cases where a feasibility study requires a lot of work.

Write a solution. Often it is helpful to do some prewriting, to organize your ideas before putting them into program form.

Write the programs. These should preferably be structured so that they are easy to understand later. Programs written in modules, each having only one entry and one exit point, are generally easier to modify and debug. Figure 4-2 and Table 4-1 show two ways of analyzing system programming. The first shows a file's structure, and a diagram of a modular program structured to read it, which has a left-right flow. Table 4-1 is a *condition table,* which lists alternative actions in tabular form, which may allow complex decisions to be checked more easily than long sections of IF statements would permit.

Figure 4-2. File Structure and Related Program

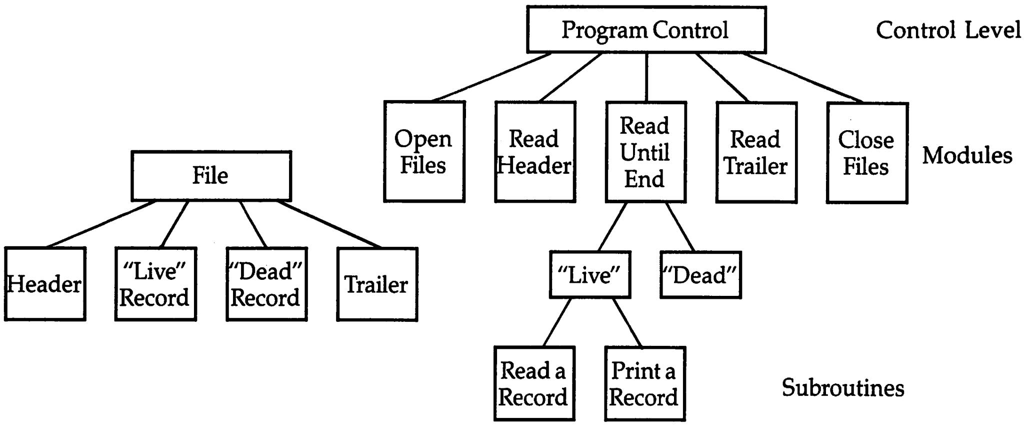


Table 4-1. Condition Table

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	-

Serious and Less Serious Programming

There's no single correct way to program. If it's your computer, you can do what you like; otherwise, you may need to conform to some set style, either of programming or of finished appearance. This section lists a number of considerations which are relevant when deciding on a program's readability, ease of maintenance, use, modification, and so on.

Conventions for line numbers, variable names, and remarks. You may want to avoid putting REM statements on lines that are the target of either GOTO or GOSUB statements, so deleting them will have no effect on the program. If standard subroutines are used, consider retaining the same line numbers in different programs. As for BASIC variables, to be sure that variables can't be accidentally changed, either list every variable as it's used (and make the names meaningful) or establish a convention. For example, local variables could end in 9 (A9, B9, ...), or I, J, K, and so on could be for local use only—initialized, then used anywhere in the program. It is still a good idea to keep track of the other variables.

REM statements make a program more readable, but take up space and slow execution speed. You may find it worthwhile to document standard routines for future reference and delete the REM statements when the routines are used in larger programs. Program 4-2 converts a four-digit hex number (see Chapter 5) to decimal and prints it. The subroutine does not perform error checking. This is a feature you may want to add later. Notice the remarks which tell what the routine does, how to use it, and which variables are used.

Program 4-2. Hex-to-Decimal Conversion Subroutine

```
560 REM{2 SPACES}SHORT SUBROUTINE TO CONVERT A STR
    ING OF
565 REM 4 HEX DIGITS TO DECIMAL AND PRINT RESULT
575 REM EXAMPLE OF USE:
580 REM L$="ABCD": GOSUB 600{4 SPACES}PRINTS 43981
590 REM USES VARIABLES H,J,L, AND L$
600 L=0:FORJ=1TO4:L%=ASC(MID$(L$,J)):L%=L%-48+(L%>
    64)*7:L=16*L+L%:NEXT:PRINTL
610 RETURN
```

A similar decimal-to-hexadecimal conversion subroutine follows; it uses the same four variables, except L is used instead of L$.

```
1000 L=L/4096:FORJ=1TO4:L%=L:L$=CHR$(48+L%-(L%>9)*7)
1010 PRINTL$;:L=16*(L-L%):NEXT:RETURN
```

Documentation. Program documentation could include an operator manual (explaining how to use the computer, handle and copy disks, and so on), a user's manual (explaining file structure, validation methods, the correct sequence of programs), and a system manual (providing a complete reference to the system programs and files).

Ease of modification. The term *hard coding* means that significant parts of a program use constants; *soft coding* means variables are used. Soft coding is easier to modify, but as a rule more tedious to write. See the program "Payroll Analyzer" below; the first line can be altered to change the program. The BASIC line:

```
OPEN N,N:PRINT#N
```

illustrates soft coding as well. When N=3, output is to the TV, and when N=4, output is to a printer. A program may include a menu of parameters at the start, so the user's own requirements can be keyed in. For example, modem programs often begin with a menu for setting baud rate, parity, and stop bits.

Error messages. These signal that a mistake has been made and should indicate the error. Program 4-3 is a subroutine that handles error messages. Before sending the program to the subroutine, place the error message text in the variable EM$:

EM$="TOO LONG": GOSUB 10000

This will print the message on the screen in reverse video to attract attention. You could use this in a large program before resetting cursor position and returning for reinput.

Program 4-3. Error Message Subroutine

```
10000 PRINT "{HOME}": FOR J=1 TO 23: PRINT "{DOWN}
      ";: NEXT
10010 PRINT "{RVS}"EM$;: FOR J=1 TO 2500: NEXT
10020 FOR J=1 TO LEN(EM$): PRINT "{LEFT} {LEFT}";:
       NEXT
10030 RETURN
```

Easy data input. The BASIC INPUT statement is fine in many cases, but doesn't give the programmer full control. To make a program as easy to use as possible, undesirable keys should be blocked out or ignored. Integer input (see the section below on string and integer input) will accept only numbers, not cursor keys, color keys, or alphabetic characters. RUN/STOP and RESTORE may need to be disabled (see Chapter 6), and the length of the integer checked if there's a maximum value. None of this is very difficult, but it takes time and memory.

How easy a program is to use is important. Prompts, telling the user what to type, and instructions, providing an overview, are helpful. The programmer must always balance the program's features with memory usage and execution speed. The lines below illustrate how to combine PRINT and INPUT into a relatively friendly input routine and show that this requires extra memory.

100 PRINT "ENTER THE DISCOUNT"
110 PRINT "PERCENTAGE (E.G., 13.25)"
120 INPUT "AND PRESS RETURN"; PC

Menus. These are elaborate prompts, which help the user select his or her own path through a program; often a help option is available from the menu. Data entry can be simplified by presenting a summary of input at appropriate places in the program, allowing for easy corrections.

The best menu design allows the user to indicate the desired option by pressing a single number or letter. If a menu program stands alone, it has to load and run a new program (see "Chain" in Chapter 6). Of course, all or most of the options may exist as one program in memory, if there is room. Tape units don't have the flexibility of disk drives when it comes to loading one program of several, of course, because tape stores programs in sequence, rather than allowing equally rapid access to each one.

Program 4-4 is a menu which calls one of three routines based on user input.

Program 4-4. Simple Menu

```
100 PRINT "1. INTEREST RATE
110 PRINT "2. TIME PERIOD
120 PRINT "3. MORTGAGE
200 GET X$: IF X$<"1" OR X$>"3" GOTO 200
210 ON VAL(X$) GOTO 1000, 2000, 3000
```

Formatting output. Tidy output, particularly of numbers, requires some work. See "Rounding" (later) and "PRINT USING" (Chapter 6), which show how to print numbers using a standard format.

Subroutines. Standard subroutines allow programs to be developed and tested as modules; it's easier to check isolated parts of a program than entire programs, and it's also possible for several people to work simultaneously, provided the variables and line numbers are determined beforehand (see the section on conventions, above) so no conflicts arise. Subroutines often save space and improve clarity.

Testing. Thorough testing ideally requires every possible combination of data to be tried. Generally, this is impossible. In practice, depending on the program or subroutine, you can use a loop to generate ascending values and check the effect, or use RND to make up strings or numbers of the right size. Rounding includes a loop demonstration; the sort routines in Chapter 6 use random data to test sorting.

In practice, there are complications. 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 quotation mark key, are all likely to crash INPUT subroutines unless they're tested for. Second, programming errors may show up only when several events occur at once, making bugs hard to trace because of their apparent random appearance. Third, unconscious bias may influence the choice of test data, so that tricky areas may be avoided. For this reason, commercial systems are tested with data supplied by the user, who also checks that the output is what it should be. This, of course, is rather unfair, since the user may not appreciate the importance of testing with obviously wrong data which the system ought to reject. In any case, it is best to have someone else test your programs, instead of simply relying on your own testing.

Validation. This is the process of checking to make sure that data is the correct type, without checking the actual values. For example, a date entered as 19/19/86 is invalid and should be rejected; 9/9/86 would be valid, but may be incorrect. In its simplest form, validation simply causes the program to wait for data, as in the menu example above. More sophisticated checking routines include error messages.

Checksums provide additional validation and are easy to implement with computers. Typically, a single letter or number is added to the end of a reference number (or even a program line listed in a book). The suffix is calculated from the data, using an algorithm, so the composite data is internally consistent. For example, International Standard Book Numbers (ISBNs) have nine digits, plus an extra checkdigit. This final digit is computed by multiplying the first number by 1, the second by 2, and so forth, up to the ninth, then adding the results together, dividing by 11 and using the remainder, 0–9, or X, (to represent a remainder of 10). The system is not foolproof, but it is simple, and the most common errors (entry of one wrong digit or transposition of adjacent digits) are trapped.

Debugging BASIC Programs

This section lists common faults in BASIC programs. While such a listing cannot be exhaustive, it should help pinpoint errors. The BASIC STOP statement allows you to set breakpoints at which you can check the values of important variables, and PRINT allows you to check key variables while the BASIC program runs.

SYNTAX ERRORs. These occur when the 64 finds something which isn't BASIC. Generally, it's up to you to find the mistake.

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

Errors of program logic. The program may run without errors, but still do the wrong thing. These are often caused by the following:

- There may be a keyword misunderstanding, so that the statement does something unexpected. This is common with logical expressions where parentheses have been omitted.
- A variable's value may be 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. In fact, the same variable may be repeated by mistake—you may forget that D already means decimal position and use it for dollars. Also, the variable may be misspelled.
- Subroutines may be poorly structured, so that program flow drops through to the following lines. This occurs when the RETURN statement is omitted in one of many subroutines.
- The BASIC pointers may be wrong: graphics definitions and ML at the top of BASIC memory need to be protected from being overwritten by BASIC strings. Chaining (see Chapter 6) may be difficult. BASIC may assume a hardware or software arrangement which is incorrect.
- Omitting FN will cause a function to be read as an array; PRINT FN HYPTN(5) is not the same as PRINT HYPTN(5). The latter will print the value of the array element HY(5).
- System errors are usually caused by errors in loops, particularly the zeroth and final elements in buffers. Loops are often used to POKE data into memory, and these are prime sources of errors.
- DATA statements may have been put in the wrong order by typing an incorrect line number.

Unusual characteristics of BASIC itself. BASIC has a number of small peculiarities, some of which are:

- ASC of a null character crashes.
- CLOSE to printer or disk file should be preceded by a PRINT#.
- FOR-NEXT and GOSUB-RETURN require caution.
- FRE is slow if there are very many strings.
- INPUT# has no error message if it finds extra data.
- PRINT attempts to print anything; for instance, a stray decimal point can appear as a zero or can cause a number to be split into two numbers.
- Numbers are not held with infinite accuracy, as Chapter 6 explains in detail.

Examples in BASIC

The following sections illustrate some of the fundamentals of BASIC programming. One of the best ways to learn BASIC is by looking at program examples and modifying them to suit your needs. Most of your programming will involve the elementary skills discussed here, in one way or another.

Input

Programs 4-5 and 4-6 use GET to build an input string, IN$. In Program 4-4, the cursor flash POKEs in lines 110 and 130 simulate the way BASIC's INPUT looks to the user. The program gets individual characters into X$ in line 120. Line 140 allows the INST/DEL key to operate. All other special keys are disallowed, except RUN/STOP and RUN/STOP–RESTORE, which can be disabled if you wish (see Chapter 6). Line 150 defines the range of acceptable characters, so for integer input the line should be changed (by placing a 0 inside the first pair of quotation marks and a 9 inside the second pair).

Program 4-5. String and Integer Input

```
10 GOSUB 100: PRINT: PRINT IN$: GOTO 10
100 IN$=""
110 POKE 204,0: POKE 207,0
120 GET X$: IF X$="" GOTO 120
130 IF X$=CHR$(13) THEN PRINT " ";: POKE 204,1: RE
    TURN
140 IF ASC(X$)=20 AND LEN(IN$)>0 THEN IN$=LEFT$(IN
    $,LEN(IN$)-1):GOTO 170
150 IF NOT (X$>=" " AND X$<="Z") GOTO 110
160 IN$=IN$+X$
170 PRINT X$;: GOTO 110
```

Decimal input is a bit more complicated, as Program 4-6 illustrates, and extra programming is needed to insure that only one decimal point can be entered. This version allows only two digits after the decimal (this can be modified at line 152). All these features can, of course, be changed, but be sure to test the results.

Program 4-6. Decimal Input

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 GOSUB 100: PRINT: PRINT D$: GOTO 10:     :rem 118
100 D$="": 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: GOTO 170           :rem 145
142 IF ASC(X$)=20 GOTO 110                   :rem 52
144 IFX$="." THEN FOR J=0 TO LEN(D$):IF MID$(D$,J+
    1,1)<>"." THEN NEXT                     :rem 100
```

```
146 IF X$="." AND (J=LEN(D$)+1) THEN D=Ø: GOTO 16Ø
                                             :rem 222
15Ø IF NOT (X$>="Ø" AND X$<="9") GOTO 11Ø :rem 226
152 IF D>=2 GOTO 11Ø                      :rem 227
154 IF D>-1 THEN D=D+1                     :rem 89
16Ø D$=D$+X$                               :rem 75
17Ø PRINT X$;: GOTO 11Ø                   :rem 225
```

GET can build strings in any format. Machine part numbers might be of the form ###XXX (that is, three digits followed by three letters), and a routine to input these should test for the correct input and ignore anything else. Where an INST/DEL key is allowed, this is a little more difficult. An input string might be accepted, then tested for correct format; if an error were found, the program would loop back for data reentry, perhaps after displaying an error message.

The discussion of INPUT in Chapter 3 explains some tricks, like forcing quotes after the prompt.

Pressing RETURN on INPUT leaves everything unchanged; so a line like:

100 X=50: INPUT "NEW X (OR RETURN=50)"; X

allows easy data entry with automatic default values.

Output

Many times you will want to print information in some special format. This is especially important in financial calculations. Program 4-7 is an error-trapping and output-formatting routine for use with numeric data.

Program 4-7. Rounding

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
2Ø FOR V=-2Ø TO 2ØØ STEP 12.7: GOSUB 1ØØ: PRINT V;
   V$: NEXT: END                            :rem 75
1ØØ T9$=STR$(V)                             :rem 67
1Ø5 E9=Ø: FOR J9=1 TO LEN(T9$): IF MID$(T9$,J9,1)=
        "E" THEN E9=J9                      :rem 31
11Ø NEXT: IF E9>Ø AND MID$(T9$,E9+1,1)="-" THEN T9
    $="Ø.ØØ": GOTO 15Ø                     :rem 146
115 IF E9>Ø AND MID$(T9$,E9+1,1)="+" THEN T9$="***
     OVERFLOW": GOTO 15Ø                    :rem 8Ø
12Ø IF MID$(T9$,2,1)="." THEN T9$=LEFT$(T9$,1)+"Ø"
    +MID$(T9$,2)                           :rem 127
125 D9=Ø: FOR J9=1 TO LEN(T9$): IF MID$(T9$,J9,1)=
    "." THEN D9=J9                           :rem 8
13Ø NEXT                                   :rem 211
135 IF D9=Ø THEN D9=LEN(T9$)+1: T9$=T9$+"."
                                           :rem 174
14Ø T9$=T9$+"ØØ"                             :rem 3
145 T9$=LEFT$(T9$,D9+2)                    :rem 223
15Ø V$= RIGHT$("{12 SPACES}"+T9$,12)       :rem 239
155 RETURN                                 :rem 123
```

Line 20 demonstrates the routine by producing test values and going to the subroutine at line 100, where the number is converted to a string. Lines 105–115 test for an E in the string equivalent of the value, V, and check for under- or overflow. Line 120 retains the minus sign, where applicable, so every possibility is tested for. Lines 125–150 handle the decimal point and trailing zeros, and control the length of the string, V$, in its processed form.

Routines like this are valuable for such purposes as printing invoices, receipts, and reports. Chapter 6 (see "PRINT USING") contains a machine language implementation of the same idea. The following line of BASIC is a simple method for rounding a number to two decimal places:

```
X=INT(100*X + .5)/100
```

Calculations

Below are some examples of calculation and report programs. The first of these predicts weight change based on information entered by the user (weight, sex, calorie intake, and level of activity).

Program 4-8. Diet Calculator

```
100 PRINT "{CLR}"
110 INPUT "WEIGHT (POUNDS)";P
120 INPUT "INTENDED DAILY CALORIE INTAKE";C
130 INPUT "INACTIVE, FAIRLY, OR VERY ACTIVE (0-2)"
    ;A
140 INPUT "MALE, FEMALE (M OR F)";S$: S$=LEFT$(S$,
    1)
150 S=1: IF S$="F" THEN S=.9
200 PRINT "{CLR}" S$ "{2 SPACES}WEIGHT NOW:" P
210 PRINT "CALORIE INTAKE:" C
220 PRINT
300 FOR W=0 TO 16
310 PRINT "WEEK" W INT(P*10)/10
400 FOR J=1 TO 7
410 M=P*(14.3+A)*S + C/10
420 D=M-C
430 DW=D/3500
440 P=P-DW
450 NEXT J
500 NEXT W
```

Lines 400–450 calculate weight change per week. Line 410 contains the formula to determine the number of calories needed to maintain the same weight; lines 420 and 430 calculate DW, the change in weight for one day. The results of 16 weeks 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 4-9 works out the smallest bill and coin combinations to pay the separate amounts of a payroll. Line 60 is a DATA line, which can be changed, for example, to eliminate hundred dollar bills, add twenty dollar bills, or convert to other currencies (the first value on this line is the number of different denominations used). Line 130 adds a small correction to each figure so there is no chance of rounding errors.

Program 4-9. Payroll Analyzer

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
60 DATA 11,100,50,10,5,2,1,.5,.25,.1,.05,.01
                                              :rem 46
70 READ NUMBER OF DENOMS: DIM NC(NU),QU(NU) :rem 6
80 FOR J=1 TO NU: READ NC(J): NEXT          :rem 72
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 6
140 NEXT                                   :rem 212
210 FOR J=1 TO EMPLOYEES                   :rem 136
220 FOR K=1 TO NUMBER                      :rem 160
230 X=INT(SAL(J)/NC(K)): SAL(J)=SAL(J)-X*NC(K): QU
    (K)=QU(K)+X                              :rem 4
240 NEXT K                                  :rem 32
250 NEXT J                                  :rem 32
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
```

Program 4-10 employs a math technique to find solutions to equations; lines 2 and 3 are examples, and the program is set up to perform an interest calculation. It will tell you, for example, that if ten payments of \$135 will cancel a \$1,000 loan, there was a 5.865 percent interest rate per payment period. This calculation is ordinarily difficult, because the formula assumes that the interest rate is known.

The program allows for guesses to be entered as well, which is sometimes important if a problem has more than one solution. Line 60 controls the precision of the answer—greater precision takes longer. The program is not foolproof and could be improved by asking for a guess and by adding error checking.

Program 4-10. Equation Solver

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
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
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                              :rem 155
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
```

Program 4-11 calculates fractional approximations for decimal values. It tells you, for example, that π is about 22/7, and that 355/113 is much closer; it approximates any constant and may provide an easily remembered fraction to use in converting currency or measuring systems.

Program 4-11. Fraction Maker

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
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
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                          :rem 78
310 T(1)=A(1): B(1)=1                         :rem 214
320 T(2)=A(1)*A(2)+1: B(2)=A(2)               :rem 182
330 FOR J=3 TO X                               :rem 50
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
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$: PRINT "HELLO, " N$

This feature is useful for games, especially text adventures, and can be used to personalize the replies the computer makes to the user. Typing trainer programs use the same principle. At a more involved level, any individual characters can be selected at will, using MID$, LEFT$, or RIGHT$ (actually, MID$ is enough), and any combination of characters can be generated with the aid of the string concatenation operator (+). Program 4-7, "Rounding," showed how to scan a string for the character E. Program 4-12 shows how a number can be scanned (in its string form) to replace the number *0* with the letter *O*, which many people prefer.

Program 4-12. Oh, Zeros

```
10 INPUT "WHAT NUMBER";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 necessary, and Program 4-13 illustrates how long numbers can be packed into about half their normal length, using string handling:

Program 4-13. Packing Numbers

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 INPUT "NUMBER"; NS$                        :rem 254
99 REM PACK NUMBER STRING NS$ INTO NP$        :rem 214
100 IF LEN(NS$) <> INT(LEN(NS$)/2)*2 THEN NS$="0"
    {SPACE}+ NS$                               :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$ :rem 170
200 NS$="": FOR J=1 TO LEN(NP$): NI$=STR$(ASC(MID$
    (NP$,J))-33)                               :rem 40
210 NI$=RIGHT$(NI$,LEN(NI$)-1):REMOVE LEADING SPAC
    E                                          :rem 22
220 NI$=RIGHT$("00"+NI$,2): NS$=NS$+NI$: NEXT:REM
    {SPACE}ADD LEADING ZEROS                   :rem 53
300 PRINT NS$ " " NP$: GOTO 10                :rem 191
```

Analogous tricks include collecting similar 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 a character string holding eight of them, and PRINT MID$(C$,J,1) prints the Jth color, where J is 1–8.

Program 4-14, a simple version of the word game Bingo, illustrates a small-system program that evaluates five-letter words by giving each letter a point value (which you may vary between runs).

Program 4-14. Wordscore

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 REM 'RUN' BUILDS FILE OF WORDS;          :rem 202
11 REM 'RUN 200' ASSESSES WORDS.            :rem 66
13 REM TAPE USE{2 SPACES}101 OPEN 1,1,1,"5-LETTER
   {SPACE}WORDS"                            :rem 63
14 REM AND{7 SPACES}301 OPEN 1,1,0,"5-LETTER WORDS
   "                                        :rem 253
98 REM BUILD FILE OF WORDS ON DISK          :rem 209
101 OPEN 1,8,2,"5-LETTER WORDS,S,W"         :rem 234
110 INPUT W$                                :rem 157
120 IF LEN(W$)<>5{2 SPACES}GOTO 110: REM 5 LETTERS
     ONLY                                   :rem 255
130 PRINT#1,W$                              :rem 28
140 IF W$<>"END**" GOTO 110: REM SIGNALS END
                                            :rem 89
150 CLOSE 1: END                            :rem 78
198 REM PUT IN LETTER VALUES                :rem 182
200 DIM V(26)                               :rem 123
210 FOR J=1 TO 26                           :rem 61
220 PRINT CHR$(64+J);                       :rem 141
230 INPUT " VALUE"; V(J)                    :rem 18
240 NEXT                                    :rem 213
298 REM READ DISK & PRINT WORD VALUES       :rem 141
301 OPEN 1,8,2,"5-LETTER WORDS,S,R"         :rem 231
310 INPUT#1,W$                              :rem 31
320 IF W$="END**" THEN CLOSE 1: END         :rem 50
330 PRINT W$;                               :rem 217
340 S=0: FOR J=1 TO 5:REM COMPUTE SCORE BY EVALUAT
    ING EACH LETTER                         :rem 43
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
```

The first part of the program accepts five-letter words (note the check in line 120) and writes them to disk, stopping when the end-of-file indicator (END**) is typed in. RUN 200 runs the second phase: 26 values corresponding to *A*–*Z* are entered by the user, and the Commodore 64 reads back all the words on file and prints word values. Line 360 converts each letter into a number from 1 (for *A*) to 26 (for *Z*). The variable, S, in line 370 is the total for the word which has been read from file; this shows how MID$ can analyze the individual letters in a word.

The program can be refined by categorizing the words, for example, into those beginning *B*, *I*, and so on, and rejecting words whose total value is less than the highest so far, or by using a different file for each type of word.

Sorting, Searching, Shuffling, and Randomizing

Sorting is commercially important in applications like bill processing, book cataloguing, and mail distribution. Chapter 6 has examples of sorts for the 64, written in both BASIC and ML.

Searching is necessary whenever you have a lot of data in memory or on a file, but have no index to directly locate a record. For example, a corporate mailing data base might store names and addresses, surname first, so that a printout of all names and addresses is easy. Often, however, it is necessary to access a given name quickly. Rather than read through all the names, a typical search method (the *binary search*) is fast and effective.

The binary search technique assumes the data is already sorted, hence its inclusion here. The idea is simple; it's the method you probably use everyday to find a name in a phone book or a word in the dictionary. You open the book exactly midway, comparing the name you want with the names at the top of the open page, and repeat the process with earlier or later halves, depending on whether the target name is before or after the current position.

A binary search works like this (don't type this in, since it is *not* a program):

```
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): Calculate new midpoint.
   Read the appropriate field of record number R, for instance R$.
   IF R$=NA$ THEN Z: Go to line Z if item is found.
   IF N1>=N2 THEN PRINT "RECORD NOT ON FILE":GOTO X: This handles
   failed searches.
   IF R$>NA$ THEN N2=R-1: GOTO Y: This revises the upper limit downward.
   N1=R+1: GOTO Y: This revises the lower limit upward.
Z  Continue processing once record is located.
```

N1 and N2 converge, sandwiching the correct record number, R, between them. The binary search is easy to program and converges quite rapidly. Even the first and last items, which take the most tests to find, can be located quickly. Table 4-2 shows the approximate *average* number of searches to find an item.

Table 4-2. Average Binary Searches Required to Locate an 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 4-15 prints a randomly dealt whole deck of electronic cards:

Program 4-15. Shuffler

```
10 POKE 53280,1: POKE 53281,1
100 DIM S(52): FOR J=1 TO 52: S(J)=J: NEXT
```

```
110 FOR J=1 TO 51
120 J% = J + INT(RND(1)*(53-J))
130 TEMP=S(J): S(J)=S(J%): S(J%)=TEMP
140 NEXT
300 FOR J=1 TO 52
310 N=S(J)-1
400 S=INT(N/13)
410 PRINT MID$("{BLK}A{RED}S{RED}Z{BLK}X",S*2+1,2)
    ;
500 V=N - INT(N/13)*13
510 IF V=1{2 SPACES}THEN PRINT "A, ";: GOTO 600
520 IF V=11 THEN PRINT "J, ";: GOTO 600
530 IF V=12 THEN PRINT "Q, ";: GOTO 600
540 IF V=0{2 SPACES}THEN PRINT "K, ";: GOTO 600
550 PRINT MID$(STR$(V),2,LEN(STR$(V))-1)"{BLK}, ";
600 NEXT
```

Lines 100–140 of the above program generate numbers from 1 to 52, without producing the same number twice. Lines 300–410 print the suit and set the correct color, while lines 500–600 convert the number to the card's value. Although Program 4-15 is fast, it is not as easy to understand as Program 4-16, a simpler method of shuffling the cards.

Program 4-16. Simpler Shuffler

```
100 DIM S(52): FOR J=1 TO 52
110 C=INT(52*RND(1)) + 1: REM RANDOM NUMBER 1-52
120 IF S(C)>0 THEN 110:REM IF USED RETRY
130 S(C)=J: PRINTC: NEXT: REM SET J: GO BACK
```

"Simpler Shuffler" puts each of the 52 numbers into an array element. If the random array position is already occupied, it tries again with another random number. This way, every possible number is used and none is repeated.

Randomizing is using somewhat unpredictable *(pseudorandom)* numbers for games, simulations, problem solving, and so forth. Program 4-17 uses random numbers to find the chessboard positions of queens, such that no queens attack each other. Rather than testing completely random boards, the program retains most of an unsuccessful test, moving an attacking queen at random, and producing results quite rapidly. The speed decreases greatly with larger boards; analysis of a 20 × 20 chessboard could take hours.

Lines 10–50 generate the starting positions of the queens, and lines 100–150 test the board to see if any queens are attacking each other. Lines 200–300 generate the printout of the positions.

Program 4-17. Queens

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 INPUT "SIZE OF BOARD";N                   :rem 246
20 DIM Q(N)                                   :rem 44
```

```
30 FOR J=1 TO N: Q(J)=0: NEXT: FOR J=1 TO N
                                             :rem 201
40 R=1 + INT(RND(1)*N): IF Q(R)>0 GOTO 40    :rem 49
50 Q(R)=J: NEXT                              :rem 89
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
200 FOR J=1 TO N: FOR K=1 TO N               :rem 237
210 IF K<>Q(J) THEN PRINT "{RVS}{WHT}W";     :rem 252
220 IF K= Q(J) THEN PRINT "{RVS}{BLK}¯";     :rem 116
230 NEXT: PRINT: NEXT: PRINT                 :rem 219
300 GOTO 30                                  :rem 47
```

Random numbers can be used to solve simulation problems of many kinds. Program 4-18 prints the sum of the values of two dice and also keeps a running score. It keeps track of the average number of throws required to produce a 7 (the answer is it takes an average of six throws to score a 7).

Program 4-18. Dice

```
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 GOTO 10
60 N=N+1: T=T+S: S=0:{2 SPACES}REM N=NO OF 7S; T=T
   HROWS
70 PRINT "SEVEN!"{2 SPACES}T/N: GOTO 10
```

Data Structures

BASIC's data structures are 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 in handling large amounts of data. Disk files have more scope than tape, since several files can be accessed at once and search time is greatly reduced. Chapters 14 and 15 give full details of tape and disk programming, respectively. "Wordscore" (above) is an introductory example of file handling.

We've seen examples using DATA statements as well. Obviously, data cannot be changed in the same way variables can, and we needn't say more about it here.

Simple variables are used often 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 related strings or numbers, using one or more subscripts to distinguish the separate items or *elements.*

One-dimensional arrays have a single subscript, which may take any value from 0 to the value used in the DIM statement that defined the array (or 10 if DIM wasn't used). The line:

DIM A$(50), N%(100), SX(12)

defines three arrays: string, integer, and real number, respectively. Space is allocated in memory for them, except for the string arrays. Arrays can be visualized as a set of consecutively numbered pigeon holes, each capable of storing one value, and initialized with contents 0. A typical application is the *lookup table.* A string array might hold values like this:

A$(0)="ZERO", A$(1)="ONE",

and so forth, so that this line:

PRINT A$(J)

would print the value of J as a word, provided J fell into the correct range. It might hold a list of names, ready for sorting, so that A$(0), A$(1), and so on, would be arranged alphabetically after sorting. Numeric arrays can be used to store the results of calculations. Many of the examples in this section use such arrays. For example, numbers ranging from 1 to 52 can represent playing card values; numbers from 1 to 8 can represent the position of queens on a chessboard, indicating the row on which that column's queen is placed. It's often worthwhile to set up tables of the results of calculations, which can be looked up rather than recalculated. Chapter 13's sound calculations illustrate this technique.

Array variables are slower than simple variables, because of the processing required, but they are versatile. DIM A$(50) makes room for 51 variables (remember the zero element) and assigns each a unique name. Without this facility you'd have to define 51 individual names, and the resulting slowing effect would be considerable.

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, pieces being represented by positive or negative numbers, with sign representing color, and magnitude, the type of piece. Two-dimensional arrays are valuable for storing data for business reports. (Three dimensions or more are conceptually a bit more difficult, but are occasionally useful.) For example, sales figures may be available for ten items in 12 different outlets. An array can keep the sets of data distinct. Subtotals and overall totals can be conveniently stored in the often neglected zeroth elements.

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

Multi-dimensional arrays—at least those with more than two or three dimensions—aren't used much, probably because of the difficulty of visualizing the data's storage pattern. Three-dimensional arrays can be pictured as rooms in a building, having height, width, and depth. After this, depiction becomes progressively more complicated. In practice, large numbers of dimensions soon exhaust the 64's memory.

RAM storage: Data may be poked into RAM for future use, or loaded from disk or tape, although this is not strictly BASIC. Chapter 6 discusses this technique and gives many examples.

BASIC data can be treated in the same way, although generally this is worth doing only when integer arrays store data 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) to PEEK(49)+256*PEEK(50).

Control Structures

Some BASIC enhancement utilities offer structures like REPEAT-UNTIL and DO-WHILE. It's possible to simulate these forms with BASIC; see FOR in Chapter 3, and this example:

```
100 FOR J=0 TO -1 STEP 0
110 REM INSERT UNTIL COMMANDS HERE
120 J= (A=B)
130 NEXT
```

This has 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 the 64's BASIC. ON-GOTO or ON-GOSUB is the nearest approach. 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 sometimes difficult to handle; this section has 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, of course, allowed for, but the years 2000 and 1600, which don't count as leap years, have not been corrected for.)

Program 4-19 is a date validation routine, which checks to make sure the day, month, and year combination is valid. D, M, and Y should be input as one- or two-digit numbers.

If OK is true, D, M, and Y are acceptable. Line 1005 expects M to be from 1 to 12, and 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 its month and year.

Program 4-19. Date Validator

```
1000 INPUT "D,M,Y";D,M,Y
1005 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)
     {SPACE}+ (M=2)*(3+(INT(Y/4)*4=Y))
1020 IF OK THEN PRINT"DATE IS ACCEPTABLE.":END
1030 IF NOT OK THEN PRINT"DATE IS UNACCEPTABLE."
```

Program 4-20 calculates the day of the week from the date. The weekday is found by an algorithm usually called Zeller's Congruence.

Program 4-20. Day of the Week Calculator

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
3 REM ENTER 3,12,86 FOR MARCH 12 1986       :rem 126
4 REM C=19 FOR 1900S; C=18 FOR 1800S         :rem 45
20 DATA SUN,MON,TUE,WED,THU,FRI,SAT :C=19    :rem 16
30 FOR J=0 TO 6: READ D$(J): NEXT           :rem 172
40 INPUT "MONTH,DAY,YEAR"; M,D,Y            :rem 162
50 M = M-2: IF M<1 THEN M=M+12: Y=Y-1        :rem 57
60 J = INT(2.6*M - .19) + D + Y + INT(Y/4) + INT(C
   /4) - 2*C                                :rem 234
70 J = J - INT(J/7)*7                       :rem 178
80 PRINT D$(J)                              :rem 248
```

Program 4-21 calculates the number of days between two dates by taking the difference between days elapsed from an arbitrary early date to the two requested dates.

Program 4-21. Days Between Two Dates

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
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 " FIRST DATE (M,D,Y)"; M,D,Y       :rem 27
110 GOSUB 1000: DX=DE                       :rem 111
120 INPUT "SECOND DATE (M,D,Y)"; M,D,Y       :rem 81
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
```

Making BASIC Run Faster

Compiling BASIC converts it into ML, which is from 3 to 20 times faster than BASIC. See Chapter 6 ("Compile") for details.

Fine-Tuning BASIC

The following methods individually have little effect, but collectively can be useful. They're arranged in approximate order of ease of implementation:

- Turning off the VIC chip takes about 5-1/2 percent off running time (providing you don't need to watch the TV or monitor). POKE 53265,0 turns off the chip; POKE 53265,27 is the usual value to reenable the screen. See Chapter 5.

- Reducing the interrupt rate by POKE 56325,255 gives the 6510 more time to process BASIC, less to scan the keyboard. Speed is increased by a couple of percentage points.
- You should DIM variables in their order of importance at the start of the program. This has some effect on speed, depending on the number of variables in the program; Chapter 6 explains why. New variables defined after arrays have been set up cause a one-time delay, too.
- Using large numbers of strings causes garbage collection delays; these can be cut down only by reducing the number of strings or by using fewer string operations. See Chapter 6.
- *Crunching* (removing spaces from BASIC programs, except within quotes, and collecting the program into as few lines as possible) improves speed by an amount which depends on the waste in the original program.
- Altering BASIC to run efficiently comes with practice, but a typical example is moving REM statements outside loops; if they're inside, the REM statements are executed many times. Streamlining unnecessarily repetitive programs will generally provide significant speed improvement.
- The discussion in Chapter 8 about moving BASIC into RAM includes a few simple commands to alter the operation of RUN. As a result, BASIC programs run about 3-1/2 percent faster.

Chapter 5

Commodore 64 Architecture

- Introductory Hardware Topics
- The 64's Memory Configurations
- Commodore 64 Ports
- Programming the CIAs
- Program Recovery and Resetting
- Commercial Software and Hardware

Chapter 5

Commodore 64 Architecture

Introductory Hardware Topics

This chapter takes you a step beyond general knowledge of the BASIC programming language, into the specifics of the Commodore 64 as a machine. The information here can be used with BASIC or machine language (ML) programming on the 64, but much of it will not apply to other computers because it is so detailed. This introductory section discusses several topics that will help you understand the rest of the book: binary and hexadecimal numbers, microchips, memory maps, the use of computers with televisions, and other hardware (machine rather than program-oriented) subjects.

Binary Numbers

A *bit*, or *bi*nary digi*t*, is a single, tiny electronic switch, which can be either on or off. It can be pictured as an ordinary switch, which either passes or blocks current. It is actually a tiny area of a computer chip that either holds an electrical charge or doesn't. Hundreds of thousands of these switches are contained in a Commodore 64, inside the black rectangular integrated circuit chips.

Each bit has a choice of only two values: on or off. According to convention, the binary voltage values are assigned numbers (no voltage is represented by 0, and voltage high is represented by 1). The systems are then structured so that binary arithmetic works correctly. The values aren't actually 0 or 1, but this provides a convenient way of talking about them.

In principle, three values could be used, making trinary arithmetic possible, but binary hardware is by now so firmly established that this is not likely to become important. As several computer manufacturers have found already, there is often little economic sense in introducing hardware based on such novel processes.

All Commodore 64 operations are binary. In hardware terms this is reflected in the large number of electronic lines needed to carry data within the Commodore 64. The expansion port, for example, has 44 separate lines. Every line, apart from those which supply power or are grounded, is treated by the system as carrying either high or low voltage values. Each additional line roughly doubles the system's potential for information handling.

A full understanding of programming requires a grasp of the relation between binary numbers and ordinary numbers. Fortunately, this is not difficult, although it can look forbidding at first. 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 MCMLIX is a different way of writing 1959.

A digit's position within a decimal number determines the magnitude of that digit; thus, 123 and 1230 mean different things. In the same way, the positions of 0's and 1's within a binary number determine the value of that number. Binary 10101100 is different from 00101011, with the leftmost number being the most *significant*. And just as decimal digit positions increase in value by powers of ten (1, 10, 100, 1000, 10000, . . .), from right to left, binary digits increase in value by powers of two (1, 2, 4, 8, 16, . . .).

To avoid confusion, a binary number 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 the decimal value 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 8 bits which are dealt with by the system as though they were a single unit. This is purely a hardware matter: IBM invented 8-bit bytes, but other numbers such as 4 or 6 bits are in use, too. A 4-bit binary number is called a *nybble.*

The 6502-based (6510) microprocessor which operates your Commodore 64 is called an 8-bit chip because it is designed to deal with data 8 bits at a time. Its addressing uses 16 bits, but only by dividing up each address into two sets of 8 bits each, called the *low byte* (the less significant) and the *high byte* (the more significant).

Since each of the 8 bits can be either on or off, you have $2^8 = 2*2*2*2*2*2*2*2 = 256$ potential combinations for each byte. That is the reason that PRINT PEEK(*address*) always yields a value between 0 and 255. It also explains why a 16-bit address cannot exceed 65535, which is $2^{16} - 1$ (since 0 is one of the combinations). There is a total of 65536 addressable memory locations, with addresses ranging from 0 to 65535.

Binary and Decimal Arithmetic

Since binary and decimal are only different notations for the same thing, it is possible to translate binary numbers into decimal. Consider only 8-bit numbers now, since they are common in programming the 64. The convention is to number the bits by their corresponding power of two, reading from the left, as 7, 6, 5, 4, 3, 2, 1, 0. Table 5-1 shows how this works.

Table 5-1. 8-Bit Binary and Decimal Conversion

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

Table 5-1 shows some binary numbers and their decimal equivalents and 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 as a whole to the 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 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 analogous to ordinary base 10 arithmetic, are crucial to the understanding of ML programs.

Hexadecimal Numbers

Hexadecimal (base 16, called *hex* for short) is another notation that is useful when programming. BASIC programmers usually ignore hex, but ML programmers find it useful. It relates directly to the internal design of the computer, and this relation helps ML programmers and hardware designers.

Hex uses 16 different numeral symbols for its arithmetic—the ordinary numeral symbols 0–9 and the letters A–F. To avoid confusion with ordinary numbers, hex numbers are preceded by the dollar sign ($). Thus, $1 is a valid hex number, as are $A000, $1234, $21, $ACE, and $BEEF. The decimal equivalents of these numbers are 40960, 4660, 33, 2766, and 48879. They can be looked up in conversion tables or converted with a programmer's calculator.

It is important to notice the relationship between binary and hex numbers, which results from the fact that 16 (hex's base) is a power of 2 (binary's base). It turns out that any 16-digit binary number (2 bytes) can be represented in only 4 hex digits. This saves space and is easier to remember.

Because each memory location in the 64 can hold one 8-bit byte (having a decimal value 0–255), it is common to write single bytes in hex using two digits, even if the leading digit is 0. Thus, the range is $00–$FF. That makes for neater programs, because the numbers line up evenly. Similarly, since memory addresses can only range from 0 to 65535, they are written as 4-digit hex numbers. It is not *necessary* that leading zeros be included ($033C and $33C are the same number), but many ML programs are written to expect such an arrangement.

Hex and Decimal Arithmetic

While programming in BASIC, you are likely to use decimal rather than hex. But there will be times when you wish to convert between the two bases (for example, to find a SYS address). It is usually easiest to convert using a program, a calculator, or a set of conversion tables. With practice, though, translation between decimal and hex as well as addition and subtraction in hex become easier.

Like binary and decimal numbers, hex values use a position convention; the further left the digit, the greater its value. The numeral 1 can mean 1, 10, 100, or 1000 (or any power of 10), depending on where it is located within a number. Similarly, a 1 in hex can represent a decimal value of 1, 16, 256, or 4096 (or any power of 16).

Remember that hex arithmetic uses all 16 digits (0–F). Thus, $09 plus $01 is $0A, not $10, and $0F plus $01 is $10. Similarly, $1234 plus $0F is $1243. Table 5-2 illustrates some hexadecimal numbers and their decimal equivalents.

Table 5-2 Hexadecimal-to-Decimal Number Conversion

Decimal Value:	4096	256	16	1	
Sample Hexadecimal Numbers:					
$A0	0	0	10*16	0	= Decimal 160
$11	0	0	1*16	1	= Decimal 17
$1000	1*4096	0		0	= Decimal 4096
$FFFF	15*4096	15*256	15*16	15	= Decimal 65535

To convert $1234 into decimal, multiply 1 by 16^3 (or 4096), adding 2 times 16^2 (or 256), adding 3 times 16^1 (or 16), and finally adding 4 times 16^0 (or 1). Conversion from decimal to hex can be done by dividing by 4096 first, then repeatedly multiplying remainders by 16 to find the next digit.

Computer memory is measured in *kilobytes* (K), where 1K is 1024 (2^{10}) bytes. Note that $1000 is 4096, exactly 4K, and that there are exactly sixteen 4K blocks of memory in the 64, $0000–0FFF, $1000–$1FFF, and so on.

Hardware, Chips, and Addressing

The 64 contains a printed circuit board where the integrated circuit chips are mounted, a power supply, and ports for input and output. A quartz crystal clock, generating several million pulses per second, controls overall timing. The 64's central processing unit (CPU) is a 6510 chip, which is an updated version of the 6502 microprocessor. It handles most of the 64's work. The *address bus* consists of 16 lines to the 6510 that allow a choice of 2^{16} (65536, or 64K) different memory locations to be written to or read from. The other lines determine which chips are active at any time, and carry signals between chips; most lines carry either about 5 volts or 0 volts.

The 6510 can address 64K because, although it deals in 8-bit bytes, it can treat 2 bytes as one address, by using the first byte as the low byte and the second byte as the high byte. (The second byte is multiplied by 256, then added to the first byte. The result is the address, which can range from 0 to 65535, since 255 added to the quantity, 255 × 256, equals 65535.) The 6510, which controls all of this data and addressing, has no position (no address) in memory.

Chips often start at addresses like $8000 because it's easier to wire them that way. This is one reason why hex is often more meaningful than decimal when discussing computers. BASIC expressions which convert hex, like MEM=13*4096 or SYS 8*16↑3, make it clear that $D000 or $8000 (in these two examples) is the relevant address.

Random Access Memory (RAM) is one kind of memory in the 64. It is *volatile,* which simply means that the information is lost when you turn the computer off. RAM can be changed and examined—written to and read from—and can hold programs or data. When RAM is changed (overwritten) accidentally, the data is described as being *corrupt*. Meaningless data left over from earlier programs is called *garbage*. The Commodore 64 can be set, or *configured,* to have 64K of RAM.

Read Only Memory (ROM) cannot be overwritten, modified, or corrupted. The 64's built-in ROMs hold BASIC and the Kernal, among other things, but they can be *switched out*—replaced by RAM or external cartridge ROM.

Briefly, here is how the 64's memory works. POKEing or PEEKing a value means the hardware has to set 16 address lines high or low in the correct bit pattern, and simultaneously set the 8 data bus lines to the correct value to be POKEd. The circuit must be designed so that only the correct location is addressed. The chip select line of the appropriate chip must be turned on, and the write line enabled. During read or write, the power consumption of the chip rises considerably. Data is transferred only when voltages have settled, at one specific stage in the clock cycle.

ROMs are made with their programs permanently *burned in*. They are mass-produced in long production runs and are subject to risks of inventory holding and manufacturing problems, and are therefore not designed for direct use of the average consumer.

On the other hand, an **EPROM (Erasable Programmable Read Only Memory)** chip resembles a ROM chip, but is made in much smaller quantities. Any program can be burned in with inexpensive, widely available equipment. EPROMs have a window in the top, usually covered by a label. If this label is removed, exposure to strong ultraviolet light will erase the EPROM for reprogramming. This is how bugs in an EPROM program can be corrected. A **PROM (Programmable Read Only Memory)** is similar to an EPROM, but not erasable. Many cartridge products, printer interfaces, and other products under development use EPROMs or PROMs.

There is an intermediate form of memory in which data stored in RAM can be in effect converted to ROM, by disabling the RAM chip's write-enable line. Battery backup allows such a package to store data even when disconnected from the usual house current. This capability could become important, in view of the relatively low price of RAM.

The 64's **Input/Output (I/O) chips** are the VIC-II (Video Interface Chip), which generates the signal for the color TV; SID (Sound Interface Device), which controls the 64's sound output; and two CIAs (Complex Interface Adapters), which are explained later in this chapter. They control most timing and input/output (keyboard, tape, and disk). All these chips have special addresses in the memory map.

Like the 6510, the PLA (Programmed Logic Array) is invisible, or *transparent*, to the programmer. It supervises hardware operations within the 64. For example, it turns off RAM when a ROM cartridge is plugged in (so the cartridge can be read) and turns off the 6510 at intervals to allow the VIC-II to generate the TV picture.

Most computer hardware has these features. For example, a typical printer has an I/O chip to read input, a CPU with a program in ROM to process and decide what to do with its input, RAM for temporary storage of text, and character ROM to select dot-matrix characters. Printers will behave differently according to the ROM fitted inside. Another example which illustrates how simple hardware fixes work is Chapter 15's modification to change disk drive device numbers.

Another interesting if not useful feature of the 64 is incomplete address decoding, which happens when some address lines are left unconnected. Since the VIC, SID, and CIA chips all have repeating images in memory, there's actually a choice of many equivalent addresses for the same functions on these chips. For example, POKE 53600,0 has the same effect as POKE 53280,0—it changes the border color to black.

Finding Your Way with a Memory Map

A memory map is a list of a computer's addresses and functions. The 64 is designed to allow different features to be *bank-selected*, like choosing cartridge ROM rather than RAM. It therefore has several different memory maps. Moreover, every chip has its own memory map—usually a small one. The VIC-II chip's map is significant, and crucial to understanding Commodore 64 graphics (see Chapter 12). Most of this book uses the term *memory map* to mean the memory locations that are connected to the 6510 chip.

The ROM in the 64 contains several different kinds of information, all of which contribute to its overall operation. They are discussed below (see Chapter 11 for a detailed look at Commodore 64 ROM).

Tables. These contain data, not programs, and have innumerable uses. The screen link table and ROM keywords are typical examples. 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, which makes the character appear reversed on the screen when using Program 5-1 below. 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 in the 64 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 (memory locations 0–255) contains many pairs of adjacent bytes that are pointers to special locations. Information about the top and bottom of BASIC text, arrays, and strings is held in this manner, for example. The pair of bytes forms an address in the low-byte/high-byte format described above. For example, locations 43 and 44 are the pointer to the beginning of BASIC program storage. On the 64, the normal values held in these locations are 1 ($01) and 8 ($08), indicating that program storage starts at location 1+(8*256)=2049 ($0801).

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 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 the 64 to be modified.

The memory examination program described below changes the vector to the *interrupt routine,* which looks at the keyboard every 1/60 second. Sometimes ROM contains vectors; the Kernal jump table is a good illustration. It is different, though, in that each address is preceded by a 6502/6510 JMP instruction and therefore occupies three bytes instead of two.

Flags. These keep track of a wide variety of events while a program is 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 related machine language routines. The only substantial program outside ROM is CHRGET, a routine at locations $73–$8A (115–138) that 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 faster than using a ROM routine, and it permits new BASIC keywords to be added 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–$70), and the realtime clock ($A0–$A2). You can use Program 5-1 to view the three bytes of the clock changing.

The stack. The stack is discussed in more detail in later chapters. Essentially, it is 256 bytes of RAM from $100 to $1FF (256–511) that are used by the 6510 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 drive is in use, a safe starting location is $013F.

Looking Inside Your 64

Program 5-1 displays the contents of any section of the 64's memory, up to 255 bytes long, at the top of the screen. Characters are POKEd into screen RAM 60 times a second, so you can watch as the bytes in some registers change. (Because the characters are POKEd into the screen memory section instead of being printed, the @ represents a zero byte, and setting lowercase mode with SHIFT-Commodore key makes alphabetic characters more readable).

To use the "MicroScope" program, type it in and run it. The PRINT statements suggest a few interesting areas of memory. Press RUN/STOP–RESTORE to turn the display off, and SYS49152 to turn it on again.

Program 5-1. MicroScope

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
4 DATA 32,115,0,240,27,32,138,173,32,247   :rem 163
5 DATA 183,132,252,133,253,32,155,183,134 :rem 223
6 DATA 254,120,169,44,141,20,3,169,192,141 :rem 15
7 DATA 21,3,96,120,169,49,141,20,3,169,234 :rem 18
8 DATA 141,21,3,96,164,254,136,177,252,153 :rem 26
9 DATA 0,4,169,1,153,0,216,192,0,208,241,76,49,234
                                          :rem 150
10 FOR J=49152 TO 49215: READ X: POKE J,X: NEXT
                                          :rem 218
20 PRINT "{CLR}{YEL}{6 DOWN}SYS 49152, START, NUMB
   ER OF LOCATIONS                        :rem 156
40 PRINT "{DOWN}SYS 49152,512,80 IS INPUT BUFFER,
                                          :rem 146
50 PRINT "SYS 49152,217,24=SCREEN LINK TABLE,
                                           :rem 47
60 PRINT "SYS 49152,255,18=NUMBER OUTPUT BUFFER,
                                           :rem 83
70 PRINT "SYS 49152,41110,255=SOME ROM KEYWORDS.
                                           :rem 10
100 PRINT "{BLK}{DOWN}SYS 49152, TURNS ROUTINE OFF
                                          :rem 174
```

You can see how 40-column lines are linked by a table into 80-character lines or watch the activity in zero page. PRINT USING (Chapter 6) relies on the number output routine. Watch ten characters (normally) line up in the keyboard queue (which starts at 631), as you press keys quickly. For more interesting places to peek around, check a memory map.

The 64 and Your TV

Commodore's designers had the problem of interfacing the 64 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.

Computers generate three types of video signals: RGB, which directly controls the voltages applied to red, green, and blue circuits, and isn't available on the 64; composite video, a mixture of these signals, available from the audio-video socket; and modulated, where the audio and video signals are superimposed on a high-frequency carrier to mimic ordinary TV signals. This is available from the 64's phono plug, but some loss of quality occurs in the process.

Some TVs do not work well with personal computers. In general, newer TVs are better than old ones 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 a good idea to keep the brightness level below its maximum.

Monitors are modified TVs designed to give good-quality monochrome or color pictures from computer output. Commodore's 1700 series, for example, works with the 64's composite signals. The 64's audio-video socket has luminance (brightness) and composite video, and the later 8-pin sockets have a chroma pin, which is pure color information. A monochrome monitor should be connected to use luminance alone, since color information will degrade the picture. The situation is more complex with color monitors. With 5-pin DINs, the luminance output (pin 1) should be wired to the luminance, or luma, input and the composite video output (pin 4), to the chroma input. With 8-pin DINs, the chroma signal should be connected to the chroma input and the luminance output to the monitor's luma input. The cable provided may not make these connections; you may have to make up leads yourself.

VCR recording requires a TV/computer switch and a band separator. This allows switching between recording regular TV programs and recording the 64's output.

If you are planning to take pictures of the screen, use an exposure not faster than about 1/30 second. Faster shutter speeds will capture the image of a dark band on the screen, because all TVs use interlace, tracing only half the picture in each scan. This creates the illusion of movement without flicker.

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.

64 Hardware Tidbits

The Schematic. This is a useful diagram in the *Commodore 64 Programmer's Reference Guide.* It corresponds, more or less, to the internal arrangement of your 64. The power input is defined in terms of a 5-volt DC line, two 9-volt AC lines, and ground. Since these are supplied outside the 64, the same 64 can be used in countries with different house current types, provided a local Commodore adapter is used, and the crystal, jumper, and VIC-II chip match the TV. The VIC-II's power supply is separate from the SID's.

A system reset connects all the major chips' RESET pins and is triggered by a timer. The address bus (labeled A0–A15) and the data bus (D0–D7) appear as solid lines. The game control ports have power supplied to them, and joysticks are wired with the keyboard, which has two pairs of eight wires, plus a RESTORE key. Note how CIA 1's interrupt request line connects to IRQ on the 6510, while CIA 2's connects to NMI. Port A of CIA 2 controls the serial bus and also selects the VIC-II's bank; port B is connected to the user port and therefore handles RS-232 communications.

The PLA. Inputs are drawn on the left, outputs on the right. The 16 input lines allow 65,536 combinations, all of which are processed within the chip to give 8 outputs, though, as we've seen, only 5 input lines determine most of the memory configuration, selecting BASIC or Kernal ROMs, the character generator, or some other area of memory. The PLA controls which banks of memory are active and distinguishes between reading from and writing to chips, allowing otherwise meaningless BASIC statements like POKE J, PEEK(J) which PEEK from ROM but POKE to underlying RAM at the same address.

Different 64s. All computers are subject to redesign as improved layouts and technology are introduced and errors removed. This process has helped Commodore reduce prices while generally improving the hardware.

At present there are three main types of 64s. All have Microsoft BASIC 2.0 (which CBM owns the rights to). The BASIC statement PRINT PEEK(65408) returns the Kernal release number.

The earlier 64s (5-pin audio-video socket) have a 1982 printed circuit board with two CIAs, BASIC ROM 01, Kernal ROM 01 (release 0), character-generator ROM (socketed), 6510, and PLA. The SID and VIC are enclosed in a box. RAM chips occupy the bottom left, with the fuse at the right. BASIC had a few bugs, notably an occasional lockup on screen editing, and an INPUT bug.

The 1983 revision (8-pin audio-video socket) has Kernal ROM 03 (release 2), which removed most bugs (and also made screen POKEs invisible). The design is somewhat different, with improved video layout (under a perforated screen) and a better TV modulator and power supply. Most chips, except VIC-II and SID, are soldered, not socketed.

The SX-64 (release 96) has a restyled casing and small monitor, but is very similar to the 64. Its power-up message is different and it has a white background default color, since light blue on blue is unsuitable for the small monitor. Tape software, although mostly still in ROM, is branched over, so device 1 no longer exists, since there is no tape port. SHIFT–RUN/STOP puts LOAD "*",8 (with a RETURN character), then RUN (with another RETURN) into the keyboard buffer, so it will load and run the first program from disk. The rest of BASIC is unchanged.

MAX. The ULTIMAX was to have been a cartridge-compatible games machine. It never appeared. It was to have no keyboard, BASIC or Kernal ROM, or character generator, a different VIC-II memory map, and only 2K RAM. Some 64 hardware features were designed with it in mind.

VIC-II and SID Chips. Chapter 12 explains how to program the VIC-II chip. The SID chip is explained in Chapter 13.

From the hardware viewpoint the VIC-II chip is important because it sometimes disturbs the timing of other chips, since the calculations it has to do are so complex, particularly with sprites enabled. This can affect the use of tape, disk, and RS-232, though not SID's sound output. VIC-II has two on/off bits, one (DEN) in register 17, which turns off screen processing (but not sprites), and another (RESET) in register 22, which in later VIC chips has no effect. To cut out this effect, sprites must be off and DEN set. The BA (Bus Acknowledge) line on the cartridge port allows for this effect when, for example, an external Z80 microprocessor controls the 64. The fact that VIC-II doesn't work instantaneously, but is continually calculating which dots to put on the screen, can be important to grasp. For example, sprite collision registers cannot be instantly updated after they've been read; some time has to pass before the next screen scan.

The 64's Memory Configurations

Five of the PLA's input lines are vital to the 64's memory management. These lines are CHAREN, HIRAM, and LORAM (which are controlled by RAM locations 0 and 1), as well as EXROM and GAME, which are controlled by hardware plugged into the cartridge socket.

CHAREN, HIRAM, and LORAM

Location 1 has six active bits, three controlling tape. Only bits 2, 1, and 0 involve memory allocation.

Location 0 is the *data direction register*. As long as its bits 2, 1, and 0 are set, location 1 controls the lines. POKE 0,0 disables the control.

EXROM and GAME are pins 9 and 8 of the cartridge socket. When a cartridge is connected, it may ground one or both of these lines, causing the PLA to reinterpret the memory map.

Figures 5-1 through 5-4 show all possible memory configurations:

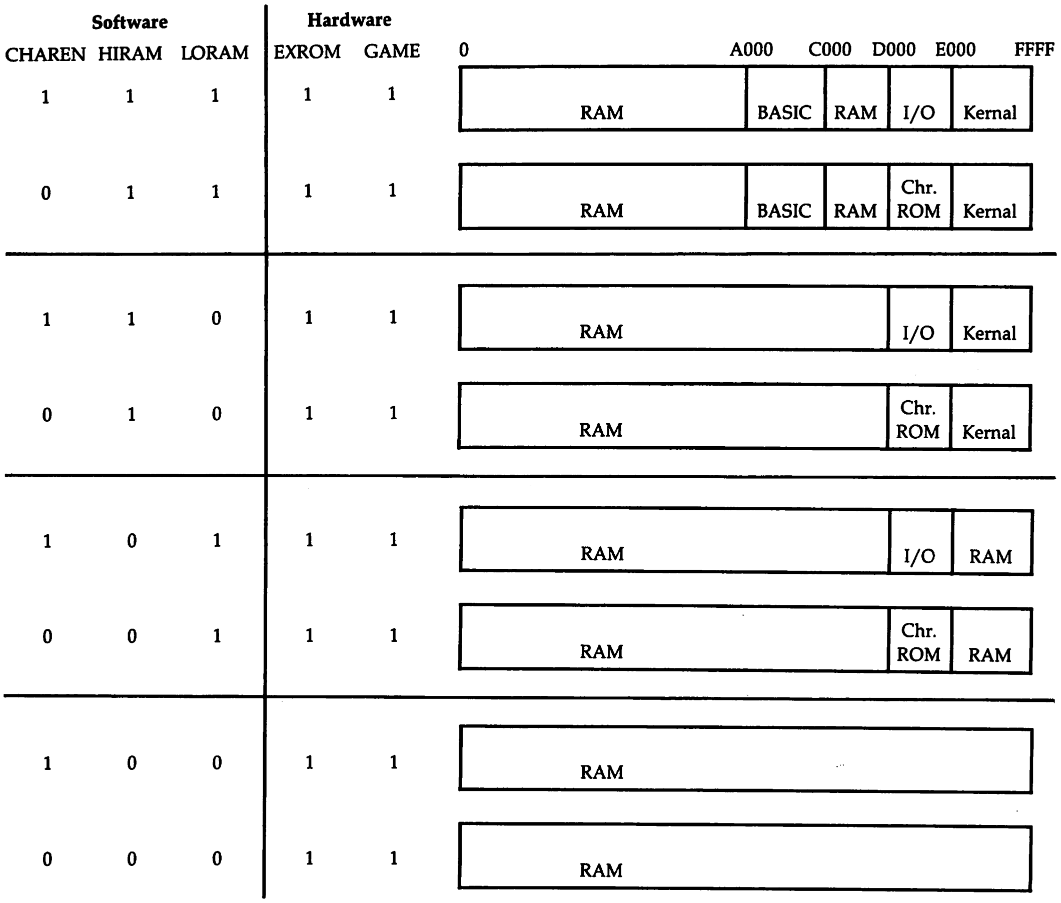
No Cartridge Connected

When you turn on the 64, you have 38K BASIC RAM plus 4K free RAM at $C000–$CFFF. BASIC ROM can be switched out, giving 52K RAM plus the Kernal, or both BASIC and Kernal can be switched out. Chapter 11 deals thoroughly with the techniques for modifying BASIC in RAM.

Memory with Cartridge and BASIC

At power up, this arrangement has an 8K cartridge, usually designed to autostart, 4K free RAM, and BASIC with 30K RAM, which you may or may not be returned to. Most cartridges using 8K or less—even pure ML—use this arrangement, since they can also borrow BASIC subroutines. Some utilities coexist with BASIC or intercept BASIC in order to add their own commands. Some utilities relocate their cartridge ML to RAM at $C000, altering BASIC vectors, then switch themselves out by resetting EXROM high. This allows another cartridge to operate, but means that RAM from $C000 must remain untouched.

Figure 5-1. Memory with No Cartridge Connected



Software			Hardware						
CHAREN	HIRAM	LORAM	EXROM	GAME	0	A000	C000	D000	E000–FFFF
1	1	1	1	1	RAM	BASIC	RAM	I/O	Kernal
0	1	1	1	1	RAM	BASIC	RAM	Chr. ROM	Kernal
1	1	0	1	1	RAM			I/O	Kernal
0	1	0	1	1	RAM			Chr. ROM	Kernal
1	0	1	1	1	RAM			I/O	RAM
0	0	1	1	1	RAM			Chr. ROM	RAM
1	0	0	1	1	RAM				
0	0	0	1	1	RAM				

If EXROM is grounded with no cartridge present, the 64 will print 30719 bytes free when turned on; it loses 8K of ROM, so $8000–$9FFF is read as garbage, but written as RAM. If GAME alone is grounded when the computer is turned on, the 64 crashes, since the Kernal is deactivated. The examples in Figure 5-2 show how the PLA detects cartridges.

Figure 5-2. Memory with Cartridge and BASIC

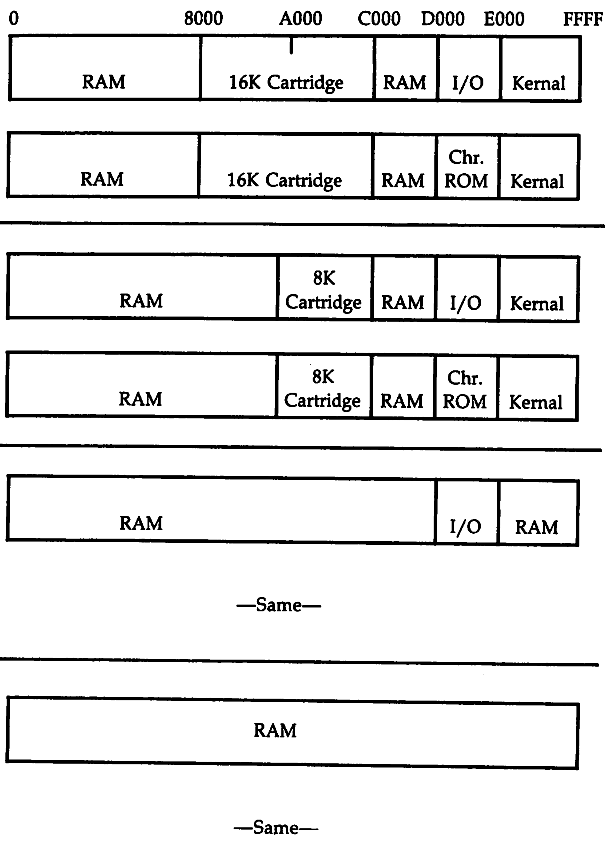
Software			Hardware							
CHAREN	HIRAM	LORAM	EXROM	GAME	0	8000	A000	C000	D000	E000 FFFF
1	1	1	0	1	RAM	8K Cartridge	BASIC	RAM	I/O	Kernal
0	1	1	0	1	RAM	8K Cartridge	BASIC	RAM	Chr. ROM	Kernal
1	1	0	0	1	Identical to EXROM high (Figure 5-1)					
0	1	0	0	1						
1	0	1	0	1						
0	0	1	0	1						
1	0	0	0	1						
0	0	0	0	1						

Memory with Cartridge but Without BASIC

This allows a 16K ML autostart cartridge to use Kernal and I/O. It's often called the application configuration, based on the theory that 16K will hold a serious program. However, it's often not enough and it's common to find cartridges using bank switching themselves. COMAL (a structured programming language) has four banks here, using 64K of ROM.

Figure 5-3. Memory with Cartridge but Without BASIC

Software			Hardware	
CHAREN	HIRAM	LORAM	EXROM	GAME
1	1	1	0	0
0	1	1	0	0
1	1	0	0	0
0	1	0	0	0
1	0	1	0	0
0	0	1	0	0
1	0	0	0	0
0	0	0	0	0

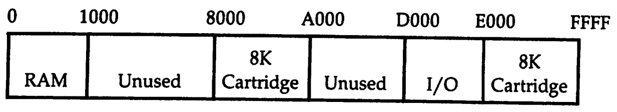

0
8000
A000
C000
D000
E000
FFFF
RAM | 16K Cartridge | RAM | I/O | Kernal
RAM | 16K Cartridge | RAM | Chr. ROM | Kernal
RAM | 8K Cartridge | RAM | I/O | Kernal
RAM | 8K Cartridge | RAM | Chr. ROM | Kernal
RAM | I/O | RAM
—Same—
RAM
—Same—

MAX

Intended to allow a 16K autostart cartridge, including its own I/O routines with 4K of RAM.

Figure 5-4. Max Memory

Software			Hardware	
CHAREN	HIRAM	LORAM	EXROM	GAME
— ANY —			1	0



Overview of 64 Memory Maps

64K of RAM is available *under* ROM. This is because the PLA insures that whenever ROM coexists with RAM, reading comes from ROM, but writing goes to the hidden RAM (or the I/O chips). You'll need to alter the LORAM or HIRAM bits to read the RAM back, as explained in Chapter 8. However, the VIC-II chip is wired to read RAM—except where it sees character ROM. Also, of course, the PLA has to switch in external ROM cartridges when detected, giving them priority over internal RAM and ROM. Note that, when the 64 is turned on, CHAREN, HIRAM, and LORAM are all set to 1, so the maps with lots of RAM must be switched in using software. They aren't necessarily easy to use; the Kernal and I/O are important if you wish to use the keyboard and screen, for example.

There are severe limitations on the amount of external ROM which the 64 can take. No external ROM can be added below $8000 without external decoding (so you must use RAM below $8000), and ROM above $8000 is confined to several blocks, arranged around the BASIC, Kernal, and character ROMs. Paradoxically, the system is in some ways less flexible than the VIC-20, where several chunks of empty memory can be filled with ROM or RAM packs.

Turnkey (ready to go) systems use the $8000 autostart feature. A cartridge can be mimicked in RAM by POKIng five bytes into $8004–$8008 to defeat a reset switch (unless EXROM is grounded). However, a cartridge which uses its own area of underlying RAM won't work if it's simply copied in RAM, and an external RAM pack, which would mimic ROM, can't be written to. So, from the software security point of view, this design is good.

Commodore 64 Ports

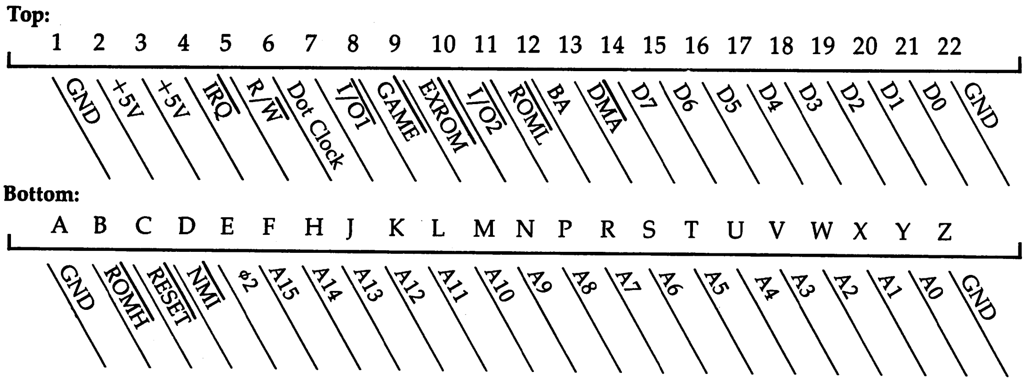
CBM computers are designed so that similar ports are compatible and dissimilar ports, incompatible. For example, the 64's cartridge port is a different size from the VIC-20's and the Plus/4's, since ML programs can't usually run on more than one of these computers. Tape ports on the 64 and VIC-20, but not the Plus/4, are compatible. And the 64 and VIC-20 user ports are similar enough for VICModem to operate correctly with either. Commodore reversed the VIC's pin numbering on the cartridge socket of the 64.

Audio/video ports. All 64s have TV-modulated output through a phono-plug. Early 64s have five-pin audio/video sockets, with luminance, audio in, audio out, composite video, and ground; later models have eight pins, partly to avoid confusion with VIC-20's five-pin socket (6V, ground, audio, two videos) which has to go through a modulator. Connecting the audio signal and ground to a hi-fi is quite easy, but putting signals into the port requires some electronics experience.

Cartridge port. This is the port at the left of the 64, looking from the rear. It has 44 connections, 22 on each side, all of which are connected. Two tracks, usually wired together, carry the +5-volt power supply to the cartridge; these are pins 2 and 3, near the top right, from the back of the 64. (As mentioned, the VIC-20 has a reversed numbering convention.) The tracks are rather close, and the possibilities of a short-circuit or arcing make it inadvisable to insert or remove cartridges when the power is on, though with care it is generally safe. Note that edge connectors are designed for the replacement of faulty computer parts during maintenance; they aren't really ideal for cartridges.

The pinouts on the *cartridge circuit board* (not viewing the computer, but the cartridge) are as follows:

Figure 5-5. 64 Cartridge Pinout Diagram



The pins function as described below:

Top

1	Ground. All four ground lines are usually tied together.
2,3	5-volt power supply to the cartridge.
4	IRQ. As long as this is low, it requests an interrupt.
5	Read/write line. Reads when low, writes when high.
6	8 MHz dot clock input, for your own video control.
7	I/O 1 goes low when 64 detects use of $DE00–$DEFF; can be used with CP/M.
8	GAME replaces BASIC ROM with external cartridge ROM when grounded.
9	EXROM replaces RAM from $8000 to $9FFF with cartridge ROM when grounded.
10	I/O 2 goes low when 64 detects use of $DF00–$DFFF.
11	ROML chip enable selects ROM $8000–$9FFF when EXROM is low; needs address bits A0–A12.
12	BA (Bus Acknowledge). To use, pull DMA low. An external device can control the 64 while BA is high.
13	DMA (Direct Memory Access). See BA.
14–21	D7 through D0. The data bus carries eight bits of data.
22	Ground.

Bottom

A	Ground.
B	ROMH selects external ROM at $A000–$BFFF (or $E000–$FFFF, for MAX) when GAME or EXROM is low; needs address bits A0–A12.
C	RESET detects a positive voltage, resetting when rising from ground to +5 volts.

D NMI connects to 6510 Non-Maskable Interrupt line. It is spike sensitive—needs a pulse in either direction. Normally high, so many devices can signal NMI.
E ϕ2 system clock. Essential for I/O timing, but not necessary for external ROM.
F,H,J,K,L,M,N,P,R,S,T,U,V,W,X,Y, Address bus (A15–A0). The full 16 address lines are necessary for DMA.
Z Ground.

A typical 16K game or word processor on cartridge uses the ground and power lines, GAME and EXROM, ROML and ROMH (for access to cartridge ROM at \$8000–\$BFFF), and the data bus, plus address lines A0–A12. All ROM addresses from 0XXX through 1XXX, plus ROML or ROMH, are therefore accessible. An 8K cartridge doesn't need GAME or ROMH.

Interfaces typically use I/O 1 and I/O 2 to control two storage buffers and R/W and ϕ2 to control timing.

Cassette port. Most CBM machines (excluding the SX-64, C16, and Plus/4) have identical tape ports. See Chapter 14. Sometimes this port's 5-volt power supply is used to drive other hardware.

Controller and game ports. See Chapter 16 for full details.

Serial port. This operates disk drives, printers, and plotters, allowing daisychaining (stringing devices together in series, each separately addressable by device number). Chapter 17 has information. It is a slow, nonstandard modification of the IEEE system found in PET/CBM machines (see *Programming the PET/CBM* for a description). CBM IEEE devices can operate with the 64, but require interfacing.

User port. This is a 24-pin port, mostly connected to CIA 2. The name is intended to suggest that users (with hardware expertise) can interface gadgetry to the 64. All CBM machines (except the Plus/4 series) have a similar user port, though the 64's has (for example) CNT (CIA counter) lines which the VIC-20's doesn't, and the VIC-20 port has cassette, joystick, and light pen lines missing from the 64 port. As a result, not all hardware items can be expected to work on both machines.

This is sometimes called the parallel user port to emphasize its potential for simultaneous eight-bit transmission, but this is a misnomer, since it can handle serial data transmission just as well.

Figure 5-6 shows the pinout as it appears from the back of the 64:

Figure 5-6. 64 User Port Pinout

Top	GND	+5V	$\overline{\text{RESET}}$	CNT1	SP1	CNT2	SP2	$\overline{\text{PC2}}$	ATN	+9VAC	+9VAC	GND
	1	2	3	4	5	6	7	8	9	10	11	12

Top	A	B	C	D	E	F	H	J	K	L	M	N
	GND	$\overline{\text{FLAG2}}$	PB0	PB1	PB2	PB3	PB4	PB5	PB6	PB7	PA2	GND

Programming the CIAs

The Complex Interface Adapter (CIA, or 6526) is a 40-pin chip, in the same series as the 6510, designed to handle interfacing. It's a descendant of the PIA and VIA chips in other Commodore machines, but it's easier to program. CIAs are versatile, but much of their capability isn't needed by the 64: the user port makes it possible for hardware specialists to use it fully. Some knowledge of CIAs is necessary to understand input/output routines, setting and reading the internal timers, and changing the interrupt rate.

Understanding the CIA

The 64 has two CIAs, numbered U1 and U2 on the schematic in the *Programmer's Reference Guide*. They are connected to the keyboard (though not the RUN/STOP–RESTORE key), joysticks, the cassette read line, the 64's serial port, the PLA lines controlling VIC-II's bank switching, both interrupt pins (IRQ and NMI) of the 6510 processor, and to many pins of the user port. As we've seen, RS-232 input and output (usually with a modem), which is serial, goes via the user port. CIAs also help control light pen and potentiometer (paddle) readings.

CIA 1, labeled U1 on the schematic, appears in the 64's memory at $DC00–$DC0F (56320–56335), occupying 16 bytes. CIA 2, labeled U2, appears at $DD00–$DD0F (56576–56591). Both chips have repeated images.

Both chips have an eight-bit data bus, usually labeled DB0–DB7. There are four address, or register select, bits, RS0–RS3, allowing 16 addresses to be distinguished, plus a chip select line, which activates the chip when low. There are also reset, clock, IRQ, read/write, +5V power, and ground lines, which have the normal functions of setting the chip when the computer is turned on, synchronizing timing, sending interrupt request signals when the IRQ line is held low, deciding whether to read the chip or write to it, and so on. In addition, the TOD (Time Of Day) pin has a 50 or 60 Hz AC input, which is used for accurate timing to 1/10 second.

The other 20 lines actually used in CIA interfacing are the 8 lines making up port A, the 8 lines of port B, and the 4 control lines, one of which is connected to the eight-bit serial register (not used by the 64). All these locations operate in the normal way of being either high (about +5 volts) or low (about 0 volts). Port A, port B, and the serial register each occupy one byte and take up 3 of the 16 locations of each CIA's memory map. Their individual bits are referred to as PA0–PA7, PB0–PB7, and SP0–SP7. The other 13 registers all control the CIA. Note that the control lines don't actually show up on the memory map; their effects have to be inferred. And remember that both CIAs have the same structure: each has a port A, for example, so be sure you've got the correct chip in mind when programming.

Figure 5-7 is a general memory map of the CIA, showing the locations and their functions.

Figure 5-7. CIA Memory Map

0 Port A	8-bit parallel port A							
1 Port B	8-bit parallel port B							
2 DDRA	Controls data directions of port A							
3 DDRB	Controls data directions of port B							
4 Timer A-Lo 5 Timer A-Hi	16-bit timer A							
6 Timer B-Lo 7 Timer B-Hi	16-bit timer B							
8 TOD-Tenths	0	0	0	0	This nybble 0–9			
9 TOD-Sec	0	This nybble 0–5			This nybble 0–9			
10 TOD-Min	0	This nybble 0–5			This nybble 0–9			
11 TOD-Hour	A.M. = 0 P.M. = 1	0	0	0 or 1	This nybble 0–9 or 0–2			
12 Serial Register	8-bit serial data + shift register							
13 ICR	Write: 0=clear, 1=set Read: 0=no int, 1=int	not used		Flag interrupt	SP line interrupt	Alarm	Timer B interrupt	Timer A interrupt
14 CRA	TOD AC Freq. 0=60 Hz 1=50 Hz	0=Serial Reg. In 1=Serial Reg. Out	0=TA counts $\phi 2$ 1=TA counts CNT	0=no effect 1=load latch in TA	0=TA continuous 1=TA one-shot	0=PB6 pulse 1=PB6 toggle	0=PB6 normal 1=PB6 outputs TA	0=Stop TA 1=Start TA
15 CRB	0=Normal TOD 1=Latch Alarm	00 TB counts $\phi 2$ / 01 TB counts CNT 10 TB counts TA / 11 TB counts TA when CNT high		0=no effect 1=load latch in TB	0=TB continuous 1=TB one-shot	0=PB7 pulse 1=PB7 toggle	0=PB7 normal 1=PB7 outputs TB	0=Stop TB 1=Start TB

General Diagram of CIA

A discussion of the CIA memory map follows:

CIA bit conventions. Fortunately, the CIA registers are accessed in the usual way, for the most part. Many registers, including the clock, timers, and ports, are simply POKEd in the usual way with an eight-bit value (or PEEKed for reading). Single bits are also used as flags. A single bit value can configure a line for output, set or read an output line, enable an interrupt to occur (or show that an interrupt has been detected), or set other features of the CIA.

If you're unacquainted with I/O chips, you may be surprised to find that many registers don't behave like RAM. For example, reading the interrupt control register clears the interrupt flags automatically.

Ports. Ports A and B have individual lines labeled PA0–PA7 in port A and PB0–PB7 in port B. Each line (controlled by a single bit) can be configured for either input or output; very often all eight are configured identically (for instance, to scan the keyboard). When configured for input, PEEKing shows whether bits are high or low. When configured for output, POKEs of 1 or 0 set high or low voltages. Hardware expertise is needed to insure that too much power isn't drawn from the chip or put into it.

PB6 and PB7 in port B can be programmed to override their normal function and act like control lines, carrying timer information, from timer A or B, respectively, either as a pulse or alternating like a square wave (toggling). We'll see an example of this feature later.

The CIA serial register (not used by the 64, and not to be confused with the more complex serial bus) is connected to a shift register, which allows user-written serial-parallel conversion. The direction and timing are controlled by control register A.

Data direction registers. DDRA and DDRB are the data direction registers for ports A and B, which set each bit of these ports for input or output. POKEing $00 into a data direction register configures the entire port for input, while $FF sets the entire port for output, and so on. Note that when the computer is turned on, the reset line to the CIA causes all the internal registers to be set to 0, so the data direction is set for input.

Control lines. Each CIA has four control lines, each connected to its own pin on the chip. Only one of the eight is used by the unmodified 64, while many go to the user port. Control lines are necessary when transmitting or reading data between devices, which otherwise would hardly ever synchronize. For example, they signal when data is ready to be transmitted. The control lines are discussed below:

CNT is an event counter for counting inputs, or a signal to line SP when the serial register generates output. CNT is not used by the 64, but is available on the user port.

SP is the shift-register pin, which is connected to the user port for RS-232.

FLAG sets the FLAG interrupt when low. This can be used with another CIA's PC line.

PC is used for handshaking. Since PC goes low for one cycle after a port B read or write, use port A first when transferring 16 bits. PC1 is not connected. PC2 goes to the user port.

→ **Timers.** Each CIA has two 16-bit timers, TA and TB, occupying two registers apiece. Both always count down. They always generate an interrupt request upon *underflow* (when they decrement below 0), but the interrupt must be enabled to actually occur.

Timers have two functions. One is timing in the usual sense; the other is counting. The first provides a regular measure of time, for instance, when sending out bits at accurate intervals; the other can perform such tasks as counting eight incoming bits, even when these are received irregularly.

Each timer has its own 8-bit control register. This includes a bit to start the time (if the bit is set to 1) or stop it (if it is cleared to 0). Any POKE into a timer is *latched* (the value is kept): Once the timer underflows or is started, it reloads with the POKEd value, so the timer's interrupt rate can be altered. However, a strobe bit allows a new value to be loaded instantly, without waiting for the timer to pass 0.

A timer can run in continuous mode (generating regular interrupts) or one-time mode (counting only once instead of continually repeating). The first mode is used to scan the keyboard; the second is used for such purposes as tracing ML commands one at a time, or detecting the presence of hardware by timing its response.

If a timer counts down with the 64's clock, the maximum time interval between interrupts is about $FFFF millionths (roughly 1/15) of a second. Timer B can count timer A, though, extending this interval considerably. Another feature allows timer B to count timer A, but only when CNT is held high.

The serial register. This 8-bit register is connected to the line SP. On command, it performs eight shifts, either moving the byte in the serial register out onto SP, as eight single bits, high bit first, or (if configured for input) reading eight bits from SP one at a time into the serial register. A shift register within the CIA does the work. When a whole byte has been shifted, an interrupt flag is set, so serial-parallel conversion can be made automatic.

The shift register is controlled by TA. It can be timed by the 64's clock or by the CNT line, which therefore can be used in handshaking.

Control registers of the CIA. Three bytes control the configuration of everything about the CIA. Register 13, called the interrupt control register (ICR), controls the five sources of CIA interrupts.

Writing to the ICR with bit 7 low clears sources of interrupts whose bits are set; this is why POKEing the register with 127 disables all interrupts. Writing with bit 7 high enables interrupts whose bits are POKEd high, so POKEing with 129 enables timer A's interrupts, but no others.

After reading the register, if the high bit is set, an interrupt has been triggered by that CIA. The bit pattern will indicate the cause. If the high bit is not set, no interrupt took place, but it's still possible for one or more of the five bits to be high, since they register even if their interrupt isn't enabled. There are examples in Chapter 8, Chapter 12 (on VIC-II), and later in this section.

CIAs in the 64

CIA 1

Port A is normally set for output and port B defaults to input. Timer A generates IRQ interrupts at regular intervals to service the keyboard, which shares lines with the joysticks and paddles. CIA 1 reads tape, using the FLAG line.

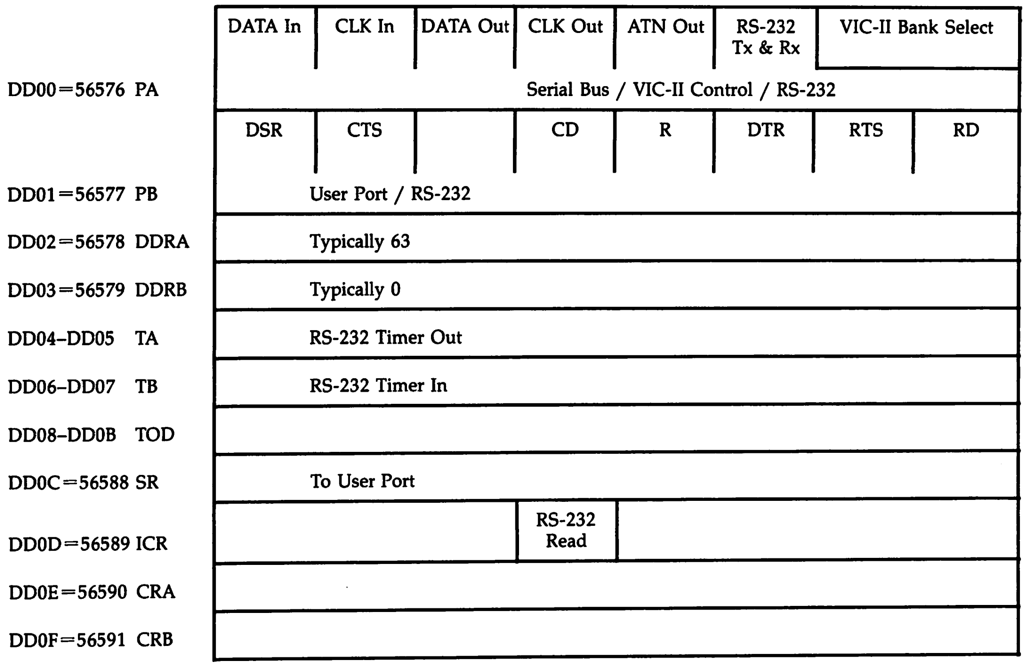
CIA 2

This chip controls NMIs, the serial bus, and RS-232 processing.

Figure 5-8. Diagram of CIA 1

DC00=56320 PA	10 Port 1 Paddles 01 Port 2 Paddles			BTN 2	Joy 2/E	Joy 2/W	Joy 2/S	Joy 2/N
					Paddle BTNS Port 2			
	Keyboard Columns (Write)							
DC01=56321 PB				BTN1	JOY 1/E	Joy 1/W	Joy 1/S	Joy 1/N
				Light Pen	Paddle BTNS Port 1			
	Keyboard Rows (Read Back)							
DC02=56322 DDRA	Usually 255							
DC03=56323 DDRB	Usually 0							
DC04–DC05 TA	IRQ Rate for Keyboard. Tape, Serial Timing							
DC06–DC07 TB	Tape, Serial Timing							
DC08–DC0B TOD	Not Used							
DC0C=56332 SR	To User Port Only							
DC0D=56333 ICR				SRQ, Tape Read			Timer B Int.	Timer A Int.
DC0E=56334 CRA								TA Start
DC0F=56335 CRB								TB Start

Figure 5-9. Diagram of CIA 2



CIA Programming Methods

The same programming principles apply in both BASIC and ML, apart from speed-sensitive processing; so the user port can often be controlled from BASIC with POKEs and PEEKs.

Program 5-2 lets you watch the CIA bit patterns change. To change a register, press any key, then type in the register address and its bit pattern. For example, you can start the internal time-of-day clock by using the register address 56328 and the bit pattern 00000000.

Program 5-2. Investigating the CIA

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
2 REM FOR CIA#2, USE FORJ=56576TO56591 IN LINE 3Ø
                                            :rem 13Ø
4 REM LINES 5Ø & 12Ø HANDLE BIT PATTERN      :rem 9Ø
1Ø PRINT "{CLR}"                            :rem 197
2Ø PRINT "{HOME}"                            :rem 7Ø
3Ø FOR J=5632Ø TO 56335                     :rem 122
4Ø PRINT J;: P=PEEK(J)                      :rem 253
```

```
50 Q=P/256: FOR K=1 TO 8: Q=2*Q: PRINT INT (Q)"
   {LEFT}";: Q=Q-INT(Q): NEXT                   :rem 245
60 PRINT " " P "{LEFT}{3 SPACES}"               :rem 104
70 NEXT                                         :rem 166
80 GET X$: IF X$="" GOTO 20                      :rem 37
100 INPUT "WHICH REGISTER";R                    :rem 202
110 INPUT "BIT PATTERN";B$                        :rem 4
120 X=0: FOR K=1 TO 8 :X=X+2↑(8-K)*VAL(MID$(B$,K,1
    )): NEXT                                     :rem 80
130 POKE R,X: GOTO 20                           :rem 110
```

CIA 1 and the keyboard. Chapter 6 explains how CIA 1 reads and decodes information from the keyboard. Setting PB6 or PB7 for output makes the keyboard unreadable and prevents the use of RUN/STOP–RESTORE to reset the machine.

Joysticks, light pens, and potentiometers. All these are shared with the keyboard, under the control of CIA 1. See Chapter 16 for full details.

Interrupt handling. Each CIA has an IRQ (Interrupt ReQuest) line, which is normally set, but can be cleared. CIA 1 connects to the 6510's IRQ line, and CIA 2 to the NMI line; each chip generates a different type of interrupt.

You should temporarily turn off IRQs to alter the IRQ vector to insert your own ML program or to read the character-generator ROM. POKE $DC0D (56333) with 127 to turn off all CIA 1 interrupts. Later, POKE with 129 (%1000 0001) to turn on timer A interrupts, which normally control the IRQ rate. Chapter 8 shows how NMI interrupts can be used with BASIC.

Tape drives. Only tape reading is controlled by a CIA (see Chapter 14 on writing tape, checking the cassette keys, and controlling the motor). CIA 1's FLAG line reads tape. Tape input is signaled on FLAG, and any negative transition sets an interrupt. Reading the interrupt register clears the interrupt flag, and as long as it's reset a low condition exists.

→ **Timers.** Program 5-2 allows you to see the timers updating on the screen. Timer A normally controls the IRQ rate, which is why POKEs to $DC04 and $DC05 (56324 and 56325) alter the cursor flash rate. TA counts down with the system clock about 60 times per second, too fast for Program 5-2 to show a pattern. Note, though, that register 5 ($DC05 or 56335) never exceeds 66; this latched value was selected because 65*256/1,000,000 is about 1/60 second. When the computer is turned on, $DC0D (56333) has bit 0 set high, enabling an IRQ interrupt to the 6510, although PEEKing won't show this.

To link timers, set $DC0F (56335) to 65 (%0100 0001). This sets timer B to follow timer A (decrementing once each time A underflows) and starts timer B. Now timer B counts down from $FFFF relatively slowly—about 60 times per second—taking about four seconds for the low register to reach 0, and therefore about 15 minutes for the whole register B to count down.

If a timer isn't started, latching has no visible effect. Set $DC06 (56326) to 0, $DC07 (56327) to 1, so CIA 1 timer B's latched value is $0100. Now set $DC0F (56335) to 81 (%0101 0001), which follows timer A, forces in timer B's value, and starts timer B. Now timer B still counts down with timer A, but is reloaded with $100.

Enter direct mode, and type the following line to set $DC0D to %1000 0010:

POKE 56333,127: POKE 56333,130

This turns off timer A and enables timer B as an interrupt source. You'll see that the cursor is flashing much slower than it normally does. The first POKE is necessary, since without it both interrupt sources are enabled.

The CNT line. Line 4 of the user port connects to CIA 1's CNT line. Latch timer B with $FFFF by POKEing 255 ($FF) into $DC06 and $DC07 (56326 and 56327). Now POKE $DC0F (56335) with 49 (%0011 0001), which sets timer B to count CNT, forces $FFFF into timer B, and starts timer B. Now, timer B remains at $FFFF until pin 4 is grounded. Note that proper debounce circuitry must be used or the count will decrement very rapidly.

Time-of-day (TOD) clocks. Every CIA has a TOD clock, but we'll use CIA 1's. Four registers hold hours (1–12, high bit for p.m.), minutes, seconds, and tenths of seconds in binary coded decimal (BCD) format. The clock starts only when the tenths register is POKEd or PEEKed, and it stops whenever the hours register is POKEd or PEEKed. This prevents errors when reading the time. For example, if the time in the clock registers happened to be 10:59:59.9, the clock could advance between the time you read the hours and the time you read minutes. This means your reading could be wrong by an hour. With CIAs, read the hours register first and the tenths register last.

You'll need BCD conversion equations to use the TOD with BASIC. This function converts a number from 0 to 59 into its BCD form:

DEF FN BCD(T) = INT(T/10)*16 + T − INT(T/10)*10

The following function converts a BCD value into the corresponding number in the range 0–59:

DEF FN TD(P) = INT(P/16)*10 + (P AND 15)

Program 5-3 is a machine language program, POKEd into memory with a BASIC loader, which reads clock 1 and prints the time. Type it in and save it, being sure to use the "Automatic Proofreader." Run the program, and then SYS 49152 each time you want to see the time. ML programmers may want to add special features to this simple clock routine.

Program 5-3. ML Clock

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 REM FOR 12-HR CLOCK, REPLACE 18 IN LINE 501 WIT
   H 0                                        :rem 20
100 FOR J=49152 TO 49220:READ X:POKE J,X:NEXT
                                              :rem 6
500 DATA 169,5,32,210,255,24,173,11,220,16,6,248
                                              :rem 46
501 DATA 41,127,105,18,216,32,46,192,173,10,220,32
                                              :rem 139
502 DATA 46,192,173,9,220,32,46,192,173,8,220,9,48
                                              :rem 166
```

```
503 DATA 32,210,255,169,154,32,210,255,96,72,74,74
                                            :rem 166
504 DATA 74,74,9,48,32,210,255,104,41,15,9,48,32
                                            :rem 60
505 DATA 210,255,169,58,76,210,255         :rem 148
510 POKE56328,0                             :rem 41
```

To use the alarm feature, POKE the registers from BASIC to start the clock, have interrupt processing ready, and set register 15's alarm bit high before POKEing in the alarm value.

POKE 56333,127: POKE 788,0: POKE 789,192: POKE 56333,129: POKE 56335,PEEK(56335) OR 128

directs interrupts to $C000 (49152), with this ML outline:

```
C000  LDA  $DC0D ; READ AND CLEAR INTERRUPT FLAGS
      AND  #04
      BEQ  EXIT
       PROCESS ALARM ...
EXIT   JMP  $EA31
```

This checks the alarm interrupt flag. In this way, lapse of a measured amount of time can end a game, display a score, or whatever. TOD alarms have a minor bug: the alarm is often triggered a second or third time, so it makes sense to turn off the alarm bit in register 15 when it's not needed.

Users of 50-cycle house current should note that the TOD is timed by house alternating current. Its accuracy relies on the electricity supplied to it, rather than interrupts. In register 14, 50 Hz or 60 Hz is selectable. But all 64s set 60 Hz even with PAL TVs. Press RUN/STOP–RESTORE or execute a SYS 64738 to return to 60 Hz. To insure 50 Hz:

POKE 56334, PEEK(56334) OR 128

or, if the Kernal is in RAM, POKE 64943,136 to alter the reset.

Program Recovery and Resetting

No matter how good you are at programming the 64, occasionally a program will crash. Suddenly, you have no control and no backup copy. There are several ways to recover your program, and they are discussed below.

The RUN/STOP key. This key works with BASIC, unless it has been disabled (see Chapter 6). It can be checked for in an ML routine, using the Kernal STOP routine (Chapter 8).

The RUN/STOP–RESTORE key combination. This is a panic button, designed to return BASIC and ML to the READY state. Hold down the RUN/STOP key and tap the RESTORE key sharply. The restore sequence has some bugs. Since it doesn't initialize the SID chip, sound may still leak out. It doesn't alter location 648; if you've moved the screen you'll need POKE 648,4, and the keys can be disabled (see Chapter 6).

RUN/STOP–RESTORE doesn't work in the following cases:

Serial bus problems can cause a lockup. For example, the printer may be in a loop or disabled; turn it off, then on.

In early 64s (release 0), the screen editor can corrupt a CIA, locking the keyboard. (Enter a long BASIC line starting at the screen bottom. Now type a line number at the bottom left and delete back. The keyscan sequence may not work now. Press #, then PLAY on tape, then RUN/STOP after a few seconds to recover.)

When you have an X2 crash caused by an internal loop in the 6510, only a reset switch (see below), followed by OLD can recover BASIC. SYS 40965 is an example. (See Chapter 6.) A so-called X2 crash occurs when the microprocessor tries to execute any opcode (other than $A2) that ends with 2 (see Appendices).

When the Computer Is Reset

First, the 6510 and all special chips are reset—they're wired to the common RESET line. Next, the reset routine at $FCE2 (64738) is entered, provided the Kernal isn't switched out by GAME. After this, locations $8003–$8007 are tested. If a specific sequence of bytes (see below) is found, JMP ($8000) takes place. This allows a cartridge to take over the 64.

Otherwise, four subroutines are called. These subroutines set CIA registers, clear pages 0, 2, and 3, then test RAM nondestructively by POKEing $55's into RAM, then $AA's, and replacing the original bytes if RAM is indicated. A few other things are set, including BASIC memory and the screen. CHAREN, HIRAM, and LORAM are all set to 1. Because BASIC RAM isn't changed, it follows that the startup routine leaves RAM from 2051 containing garbage.

JMP ($A000) is next. This is an indirect jump, which usually sends execution to the BASIC cold start routine at $E394. But if both EXROM and GAME are grounded, and a cartridge hasn't been signaled at $8000, then $A000 provides an autostart.

Note that RESTORE jumps indirectly through (8002) or (A002), using the same tests. This is the so-called warm start at $E37B.

Reset switch. Pin C of the expansion port or pin 3 of the user port starts the reset sequence when grounded. Because the 64's circuitry varies, it's crucial to use a properly designed circuit. Be careful to get the right pins; grounding a 9V user-port line will blow the fuse on the 64 (or worse).

A reset switch has a similar effect to switching on, except that the contents of RAM from $0800 up are left completely intact, apart from a few zero bytes at the start, caused, in effect, by NEW. The whole of BASIC, and its variables, can be recovered; Chapter 6's OLD shows how.

A reset gives control of the 64 to BASIC and the Kernal ROM, but after giving priority to the hardware EXROM and GAME lines. This insures that the computer always starts correctly when turned on. Use of BASIC in RAM is overridden by a reset switch. But if the reset sequence detects the five bytes that indicate a cartridge is present (whether one is actually there or not), execution can be deflected to any address you choose. This is how software can be made reasonably invulnerable to resetting.

SYS 64738 behaves very much like a hardware reset: both call the same routines and set the 64 to its startup condition, except that RAM above $0800 is retained by

SYS 64738. OLD can then be used to restore BASIC, and ML in memory remains intact. SYS 64738 is therefore better than turning the 64 off, then on again, but it has a few odd effects when BASIC's ROMs are switched to RAM, as detailed in Chapter 8.

Autostarting

Autostart cartridge ROMs usually start at $8000, like this:

$8000–$8001:	Startup jump address in cartridge.
$8002–$8003:	RESTORE key jump address in cartridge.
$8004–$8008:	Standard five-byte identifier sequence, $C3, $C2, $CD, $38, $30 (CBM80).

If the reset routine detects the correct identifier sequence, the jump address is taken. Typically, it still goes through most of the normal initialization routines before continuing with its own program.

A cartridge starting at $A000 will autostart without needing identifying bytes, unless it finds the five-byte sequence at $8004. Obviously, such a cartridge must replace the BASIC ROM by grounding GAME and EXROM.

Bypassing cartridge autostart. If you want to run BASIC or ML normally, but have a cartridge which you don't want to unplug, you may want to reset the 64 to its normal state by bypassing autostart. This is easy if you have a good expansion board: just switch the unwanted cartridge off and use SYS 64738.

Cartridges that return you to BASIC are fairly easy to disable. Try the following:

- RUN/STOP–RESTORE.
- SYS 64760.
- Mimic the reset sequence without the cartridge test, by POKEing these bytes into 49152 (and subsequent locations): 120, 162, 255, 154, 232, 76, 239, 252. Call the routine with SYS 49152.

This leaves the cartridge in memory, so SYS 64738 will act like turning the computer on and SYS 49152 will return the machine to normal. Thus, with OLD, BASIC could be switched at will to run with or without some BASIC utility.

Cartridges that don't return you to BASIC can't be bypassed without an expansion board (which has a switch or other means to disconnect EXROM or GAME or both). It is not recommended that you plug in a cartridge while the power is on; you may damage the computer or at least cause it to crash. If this is done, however, the cartridge may become fully present in memory without autostarting.

Commercial Software and Hardware

Programs for the 64 come in several formats, and new devices are being introduced every month. Commodore is only one company making products for the 64, and outside manufacturers are making hardware that is Commodore-compatible. The following section briefly discusses some of the devices which are available.

Cartridge Programs

ROM programs plug into the cartridge port and usually autostart when the computer is turned on—this is convenient and fast.

Because cartridges, even with EPROMs, are much more expensive to make than the easily duplicated tape or disk software, major programs are often supplied on disk. Cartridges therefore often contain games. Sometimes there's a compromise: plug-in software may immediately load another program from disk. Since many applications require a disk anyway, this isn't a problem, and it has the advantage of permitting corrections to be made in the code if bugs are found in the disk program.

The 64's design insures that cartridges compete for the same parts of memory, so some form of expansion board is necessary if you want a choice of easily accessible cartridges. The only exception is some interface cartridges, which relocate their ML and then turn themselves off. But even these can't really be made with an extension piggyback socket, since other cartridges may have incompatible software. In other words, be prepared for your cartridges only to operate individually.

Most cartridges have software protection to prevent a RAM copy of the cartridge from working properly. For instance, an ML instruction like INC $8300 increments the contents of $8300 in RAM, but has no effect on ROM, so an ML program that alters memory locations within itself will work in ROM but not RAM.

It's actually possible to store BASIC programs in cartridge form, with their variables lower in RAM, so that they can autostart without having to be loaded. This is rarely done, since there's only space for 8K.

Cartridges tend to look rough when taken apart. Often there is just a printed circuit board with a ROM or EPROM, plus some other circuitry. If you open the cartridge, you'll be able to follow the tracings and infer the function of much of the hardware. Note how the dominant direction of tracks on top is often perpendicular to that below, allowing connections between the surfaces to be made more easily. The casing provides protection, but isn't necessary to the working of most of these devices, which can be plugged in as plain printed circuit boards.

Examples. Many games and utility programs are packaged as autostart cartridges, as are programming languages like COMAL. Most languages switch BASIC out completely.

BASIC extensions and aids are available on cartridges as well. Simons' BASIC is 16K of ROM from $8000 to $BFFF. It switches the BASIC ROM in and out by controlling the GAME line. Other utilities have features like fast tape operating systems, additions to the disk operating system, extra graphics, and so on. Unfortunately, the programs you write using these cartridges may not run without the cartridge in place. And watch for subtle changes in BASIC—the ',8' may no longer be needed to use the disk drive, or you may find tape no longer works.

Voice and music synthesizers, with extra commands like SPEAK, generally use the SID chip if they don't add hardware to the audio-video socket.

Eighty-column software can be prepared as a cartridge as well. It intercepts vectors, controlling screen output and graphics format to produce characters four dots wide. (There are 320 pixels across the screen to use, so 64 columns are possible, too, with characters five dots wide.) All the normal colors may not be available. Eighty-column output usually requires a monitor, not a TV.

CP/M is an operating system which runs on the Z80 chip. The 64's DMA (Direct Memory Access) line allows a Z80 or other chip to control the 64: when held low, the Z80 controls data, addressing, and input/output, though VIC-II still has access to and controls the display. The Z80's memory addresses are offset $1000 bytes from 6510 addresses, seeing $1000 as $0, and so on.

CP/M has three parts: console processing (CPP), disk operating system (BDOS), and input/output (BIOS), all at the high end of memory. The programs start at $100. But the disk and input/output is specific to the 64 and not transferrable. The Kernal, of course, can be run only by the 6510, not the Z80. In the Commodore package, a hardware switch at $DE00 selects the processor. A CP/M cartridge is necessary to use the Z80; then software is usually loaded from the 1541 disk drive. You may need to transfer information to other machines by modem, since the 1541 disk drives are not compatible with those of most other manufacturers. If you want to try CP/M with minimum hassle, be sure that the programs you'd like to run are available for the 64.

Programs on Disk

Disk programs are often long and slow to load. Because these programs go into RAM, copying can be relatively easy, and this is a problem for the software publishers. Various methods are used to deter copying, like making the directory unreadable and causing the program to force-load in memory, then decode itself in complicated ways (explained in Chapter 15). Don't be surprised if the disk directory looks strange.

Examples. Most cartridge programs of the types listed above are available on disk as well: games, 80-column software, terminal software, some BASIC enhancements. Exceptions include the rare, very long ROM programs, which require bank switching and can't fit into the 64's RAM.

Applications software (word processing, spreadsheets, and data bases) is often available on disk. Unfortunately, the 64 can't autoboot (automatically load and run) software from the disk when the computer is turned on. You have to type in at least a LOAD command. Commodore's *EasyScript* word processor is a typical disk-based application package. LOAD "ES",8,1 forces a load which automatically runs. *EasyScript*'s ML actually starts at $8000.

Programs on Tape

Tape is perhaps the cheapest storage medium, and programs of most types listed above come on tape. Tape access is also usually very slow. But the 64 has plenty of room for alternative tape LOAD programs. Some commercial tapes first load a fast tape-read routine, which reads the specially formatted main program very rapidly. There are several recording methods like this, which are often harder to audio-copy because of the higher frequencies they use.

Another example is BASICODE, which uses a subset of Microsoft BASIC in order to make each program run on a number of different computers. It includes standard routines, different for each computer (for example, GOSUB 100 to clear the screen), and is without significant color, sound, or graphics. Programs can be broadcast by radio and recorded with an ordinary cassette recorder. A special program for the 64 reads this as normal BASIC, which can be saved in conventional 64 tape or disk format.

Transferring tape software to disk. Straightforward programs can be read from tape, then stored on disk. There are public domain utilities available which help with this. Chapter 14 explains how to load standard tape programs anywhere in RAM. But protection methods make this more difficult. For example, programs which use the tape buffer can cause problems. And programs that load and run another program won't work unless the device number is changed from 1 to 8. Nonstandard formats are difficult to put on disk as well: If the loaded program disables RESTORE, mimics a ROM cartridge at $8000, and also uses RAM around $300, resetting with EXROM grounded will sometimes work, but will wipe out the software in low memory.

Hardware Devices

These peripherals perform functions which software alone can't. There's a wide variety, made somewhat confusing because there are often several different ways to achieve the same result. Here is a quick look at the most important broad categories.

Interfaces. Interfaces connect the 64 to non-Commodore equipment so that the external hardware device will work properly. The problem is that since the 64 has no standard interface, it must use an interfacing device with a 64-compatible socket on one end and a socket to fit standard equipment on the other.

If you want to connect a daisywheel typewriter or non-CBM matrix printer to the 64, you'll find this equipment typically uses either Centronics or RS-232 inputs, or both. Be warned that there may well be problems with incompatibility: Test equipment before you buy it to make sure that your software works correctly with it.

Another interfacing problem is CBM's IEEE disks and printers. The double disk drives, like the 8050, are much faster and store far more data than the 1541, but they won't plug in directly to the 64.

The Centronics interface is a 36-pin parallel system found on most good-quality printers and also on some typewriters. It has its own handshaking system and has no need for a baud rate to be set. Some of these interfaces have special features relevant to Commodore machines, like printing BASIC control characters as {RED} or {HOME}, so program listings are more readable.

User port interfaces use port B (plus two lines to control handshaking) as a parallel port. All that's required in hardware is a Centronics cable fitted with an edge connector to plug in the user port. Software controls the user port.

Serial port interfaces connect the serial socket in the 64 (or at the disk) to the Centronics device via hardware, which usually includes an external power supply, some way to define its device number, and some control over the type of ASCII it passes to the printer.

The cartridge port can be used as well, but of course is likely to conflict with other software on cartridge.

RS-232 is a standard for transmitting serial data, which tends to be slower but less expensive to implement than parallel transmission. The RS-232 connector has 25 pins arranged in two rows. The 64 requires voltage conversion to convert its user port output to the correct levels of −12 and +12 volts; Chapter 17 has more details on CBM interfaces.

RS-232 devices are assigned device number 2 when a file is opened with a statement like OPEN *10,2,secondary address,baud rate,* and subsequent input or output uses two buffers at the top of BASIC memory for storage. Software has to be written with this in mind; if it assumes a printer is device 4, it won't work without modification. Also, the baud rate must be set. (Generally, Centronics interfaces are likely to be easier to work with.)

Some IEEE interfacing cartridges plug into the main socket. One type prints its sign-on message, relocates its ML to start at $C000, alters BASIC vectors, then disconnects EXROM so that it disappears from the memory map. Finally, it loads and runs a program from disk. The 24-pin IEEE edge connector protrudes from the cartridge to run CBM IEEE disks and printers.

Another type also has an IEEE connector, but adds BASIC 4.0 commands like CATALOG (which can be found on the later CBM/PET machines) and an ML monitor program, making it highly compatible with other CBM products. The extra software is relocatable. While this is versatile, to make full use of it, you need to know how to determine where programs are stored in memory.

It is also possible to move BASIC into RAM and alter the output routines. This way you can customize and enhance BASIC to meet your needs.

Linking devices. Interpod is a CBM product which plugs into the serial port, converting the 64's serial bus signals into a form acceptable to IEEE devices, and vice versa. Since it has an external power supply, ROM and RAM, an IEEE socket and another serial socket, either or both types of hardware can be connected to the 64's serial bus. This system has the advantage of being *transparent,* not interfering with normal operations. But since 64 software will not normally expect IEEE devices to be present, programs which use the system may need to be specially written.

Interpod is an example of linking; however, some adapters are designed to allow more than one 64 to share the same printer or disk drive. This can be useful in a classroom environment. Some caution is required, though; the simplest interfacing methods don't allow for simultaneous operation, so users must warn each other when they're about to use the shared devices.

Expansion boards. To save wear on the 64's ports, you may want to purchase an expansion board. A typical board has three to five slots, in which cartridges fit upright, facing the user. Each slot has an on/off switch. To work properly this must disconnect the power line and also GAME and EXROM. In this way, several cartridges designed for the same area of memory can be in position simultaneously, though only one can be in operation at any one time. Some boards have a fuse, and a reset switch is a good idea. These boards are relatively simple, and if you have experience with printed circuit board equipment, you could make one using ribbon connectors, rather than rigid boards.

EPROM boards. These are printed circuit boards, with edge connectors to fit the cartridge port, wired to sockets for Erasable Programmable Read Only Memory chips. Some boards allow you to switch between EPROMS with a POKE to a specially reserved location, as well as to select the address of the EPROM. Essentially, they allow you to design and run your own cartridge software. There's considerable scope, since the expansion port has access to all the bus signals.

An EPROM programmer is a hardware device that puts your software into semipermanent form in an EPROM. You will also need a special ML monitor program with a command to start the burn-in. Self-contained units are sold which fit the user port, drawing the high voltage required from it, and in effect using RS-232 to control

EPROM burning. An EPROM can be read back through the user port, but not run as a program; to run the EPROM, it should be mounted on a board.

Communications devices. Modems fit the user port and require terminal software to control communications. Cartridge port modems include their own sign-on message, request to connect, and menu of options. All types, of course, have a connection to the phone line or to an acoustic coupler. Chapter 17 discusses terminal software.

Controllers. Another type of adapter is a controller, or I/O board. This set of relays, or digital/analog converters, connected to the user port, allows the 64 to control external equipment—lighting, machinery, test equipment, alarms, whatever. A set of BASIC subroutines often is adequate to read and drive the controller. Obviously, hardware and software experience are both needed here.

Other devices. A tape interface can connect the 64 to an ordinary recorder, but this is more difficult and not much less expensive than using the Commodore tape units. An ordinary recorder isn't wired to sense keys, so it can't prompt with *PRESS PLAY* or *PRESS PLAY AND RECORD*, and the recorder must be switched on just before use. Some recorders play back an inverted signal, which sounds identical but cannot be read properly. See Chapter 14 for more on this.

Cartridges are sometimes fitted with their own video chips (so the TV connection comes from the cartridge, not the 64). This allows 80-column display, variable line spacing, digital clock display, and so on, but at a price, of course. And cartridges can take speech synthesis chips, again with output derived from the cartridge, either bypassing or mixing with the sound from the SID chip.

Special hardware modifications are becoming available for the 64 as well. One autostart cartridge causes a 1541 disk drive to work at several times the normal speed, subject to some serial bus problems. (It switches itself out of ROM, leaving BASIC modified with connectors to two lines within the 64.) Another reported modification allows the 64 and Apple to share some software.

Chapter 6

Advanced BASIC

- How BASIC Is Stored in Memory
- Special Locations and Features of BASIC
- Dictionary of Extensions to BASIC

Chapter 6

Advanced BASIC

This chapter explains advanced BASIC methods for programming the 64. It should help you learn new techniques and avoid programming bugs. The following sections introduce some special features of BASIC, like programming the function keys, for example. There's a dictionary of enhancements to BASIC, some of them utilities to assist programming, others subroutines which can be incorporated into programs.

How BASIC Is Stored in Memory

Major Memory Locations Controlling BASIC in the 64

The previous chapter explained how pointers controlling BASIC are set up when the 64 is turned on. These pointers are listed below (in decimal):

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)
START OF BASIC RAM = PEEK (641) + 256 * PEEK (642)
END OF BASIC RAM = PEEK (643) + 256 * PEEK (644)
START OF SCREEN MEMORY = 256 * PEEK (648), normally 1024
START OF COLOR RAM = 55296
START OF CURRENT TABLE OF CHARACTER GENERATOR BITS = 16384 * (3−(PEEK (56576) AND 3)) + 1024 * (PEEK(53272) AND 14)

The main set of pointers occupies 14 consecutive bytes, from location 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, for example.

All this means is that the program ends one byte before the pointer and that the variables start exactly at the pointer, so there's no overlap. Such a convention is obviously necessary, and this pattern is common among Commodore machines; this is why the last byte can easily be lost if you are not careful when saving ML programs.

These pointers are most important when considering BASIC on its own, and much of this chapter is devoted to them. For completeness the list also includes other pointers set by the 64 when it is turned on or reset. 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. The screen and character generator pointer values are dependent on the VIC chip, and setting their values is relatively complex; they aren't ordinary two-byte pointers.

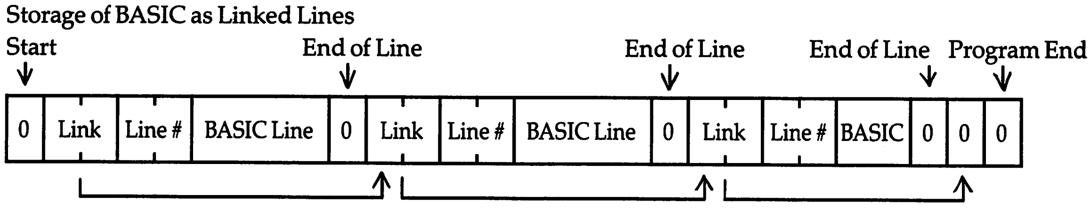
Experimenting with BASIC Storage in Memory

As an introduction to BASIC pointers, turn on your 64 and PRINT some of these two-byte PEEKs, starting with PRINT PEEK (641) + 256*PEEK (642). You should find that the BASIC RAM extends from 2048 ($0800) to 40960 ($A000), and the screen starts at 1024 ($0400). The 64 allows 1024 bytes for the screen, because $40 \times 25 = 1000$, and 1024 (2^{10}, or 1K) is a convenient amount of space to allocate for it. The screen ends at 2023 ($07E7) and there are 24 bytes left over, 2024–2047. The final 8 of these store sprite data, while the rest are free.

You will also find that the actual start of BASIC is one byte beyond the start of BASIC RAM. In other words, locations 43 and 44 together point to 2049 ($0801). This is because BASIC always starts with a zero byte. Usually, therefore, PRINT PEEK (2048) will print a 0 on the screen. The *BYTES FREE* message in this case has already calculated that bytes from 2048 to 40959 (a total of 38,911) are available to BASIC. The pointers to the end of program text, simple variables, and arrays are all set to 2051. BASIC programs have two consecutive zero (or *null*) bytes marking the end, and since there are no variables yet, all these pointers are in their starting positions, just after the program, which is where variables will be stored. Zero bytes (with a PEEK value 0) are convenient for markers because they are easily tested for in ML. So PEEK (2049) and PEEK (2050) both return 0 at present.

BASIC is stored as a set of numbered lines. As stated above, the first byte of a program is a zero, or null. Each line begins with a two-byte forward link address, a two-byte line number, the BASIC line itself, and a zero byte which marks the end. 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 which can be scanned at high speed. Each link needs two bytes, and line numbers also have two bytes, enabling line numbers greater than 255. Figure 6-1 illustrates the concept of linked lines in BASIC and shows that a line link of 0 0 (two consecutive nulls) is used to indicate the end of the program.

Figure 6-1. Linked Lines of BASIC



The built-in operating system automatically arranges BASIC as lines are typed in and entered (by pressing the RETURN key) at the keyboard. When you understand how lines are entered, you can modify BASIC to produce nonstandard effects. You could insert lines longer than usually possible, add normally unavailable line numbers, or arrange a line of BASIC to contain things it ordinarily couldn't.

Here's an easy way to look at BASIC line storage in practice. Type in this simple program:

Program 6-1. BASIC Line Peeker

```
1 PRINT "HELLO"
30 FOR J = 2048 TO 2062
40 PRINT J;PEEK (J);CHR$ (PEEK (J))
50 NEXT
```

When you run this, you will see the following numbers on the computer screen (without the comments):

```
2048    0      ; Zero byte at the beginning
2049   15      ; Link address; points to start of next line
2050    8      ; at 2063 = 15 + 8 * 256
2051    1      ; Line number; this line number is 1
2052    0      ; = 1 + 0 * 256
2053  153      ; Tokenized form of PRINT
2054   32      ; Space
2055   34  "   ; Quote and following characters in PETASCII
2056   72  H
2057   69  E
2058   76  L
2059   76  L
2060   79  O
2061   34  "
2062    0      ; End of line null byte
```

Program 6-1 shows the contents of every byte of a single typical line of BASIC, with notes to show their function. Incidentally, you will get slightly different results if you make small changes to line 1. For example, if you remove the single space character between PRINT and the first quotation mark, this will be missing from location 2054, as you would expect. A link address pointer will point to 2062 instead of 2063, because the next line starts a byte earlier. Similarly, try a different line number in place of 1, and see it reproduced in locations 2051 and 2052 when the program is run.

As noted above, BASIC lines have five bytes of overhead, consisting of a two-byte pointer, a two-byte line number, and a null byte. The end of BASIC is marked by two zero bytes—that is, when a link address is found to be 0, RUN and LIST will automatically treat this as an END, and return to direct mode with READY.

Before performing the following POKEs, save Program 6-1, if you want to be sure to have a copy. Try POKE 2063,0: POKE 2064,0 with the program (in the exact form shown above) in memory. It will now LIST only one line. Now POKE 2063,37: POKE 2064,8 which will replace the original values, if the spacing was *identical* to the version above. LIST now shows the entire program again.

Normally the end-of-BASIC pointer in 45 and 46 will point just beyond 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 validly be zero, but any normal BASIC line will be in $0800 at the absolute minimum, so its high byte will never be less than eight.

Watching BASIC Work

There are several other methods to watch BASIC and its variables as they do their work. Chapter 5's ML program, "MicroScope," can display a dynamic picture of BASIC's storage, an excellent way to get the feel of BASIC lines, variables, and strings. Another way is to move the screen to coincide with the start of BASIC. Type in and run Program 6-2 below.

Program 6-2. BASIC Screen

```
10 FOR J=55296 TO 56296: POKE J,0: NEXT
20 POKE 648,8: POKE 53272,37: POKE 53281,1
30 PRINT "{HOME}{4 DOWN}";CHR$(14): REM YOUR NAME
   {SPACE}HERE
40 FOR J=1 TO 3000:NEXT
50 REM TYPE POKE 648,4 THEN PRESS RUN/STOP & RESTO
   RE TO GET BASIC PROGRAM BACK
```

Line 20 moves the screen to $0800, where BASIC's program storage area begins. Line 30's REM and line 40's loop activity will both be visible onscreen. Try adding new lines of BASIC and new variables. The result is rather unpredictable—clearing or scrolling the screen will remove or alter BASIC.

Another way to display a BASIC program's contents is to POKE it, byte by byte, into the screen, as Program 6-3 demonstrates:

Program 6-3. POKEing BASIC to the Screen

```
10 FOR J=2048 TO PEEK(45) + 256*PEEK(46)
20 POKE 1024+Q, PEEK(J): POKE 55296+Q,1: Q=Q+1: NE
   XT
30 PRINT "{HOME}{4 DOWN}";CHR$(14): REM LOWERCASE
   {SPACE}MODE
```

Table 6-1 shows the significance of each BASIC byte (apart from the links and line numbers). Note that all BASIC keywords are stored as a single byte with bit 7 set (which means they have a value of 128 or more in decimal). This makes it easy for the ML routine to detect a keyword. It also means that when BASIC is POKEd into the screen, BASIC keywords appear as single reverse-video characters on the screen. Generally, BASIC as stored in RAM looks strange, partly because of this compression and partly because the pointers and line numbers become 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 like this is a very valuable space-saving feature. This special coded form is different from the 64's ASCII and also different from the screen display system—which may cause some confusion. Table 6-1 shows all valid bytes as held in BASIC. Some of those not shown will list as apparently meaningful BASIC, but will not run properly.

Table 6-1. Internal Storage of BASIC Bytes

0–31	32–	64–	96–	128–	160–	192–	244–255
	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 64 ASCII characters can be obtained; see Appendix H for a table.

Use this when PEEKing BASIC or modifying BASIC with an ML monitor.

It's easy to show how a table like the one above can be compiled. Type NEW and enter this one-line program:

```
0 X
```

The 64 stores X at location 2049 + 4 = 2053. Therefore, POKEing 2053 with some value, then listing, reveals how BASIC treats the value. POKE 2053,128 lists as 0 END, for example, as the table indicates. If you wish to investigate this methodically, enter:

```
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 change the color of the characters printed on the screen after that to white.

If non-BASIC bytes are POKEd in, LIST will often list them as something apparently sensible, but the line will crash with a *?SYNTAX ERROR* message when you try to run the program. 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 we'll see.

Relatively few values out of the possible 256 comprise valid BASIC. Note that numbers 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 prevent ambiguity, and there is no way that numbers could be compressed without making them resemble tokens or other BASIC features. Note also that the operators +, −, *, /, and the up-arrow (↑) 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.

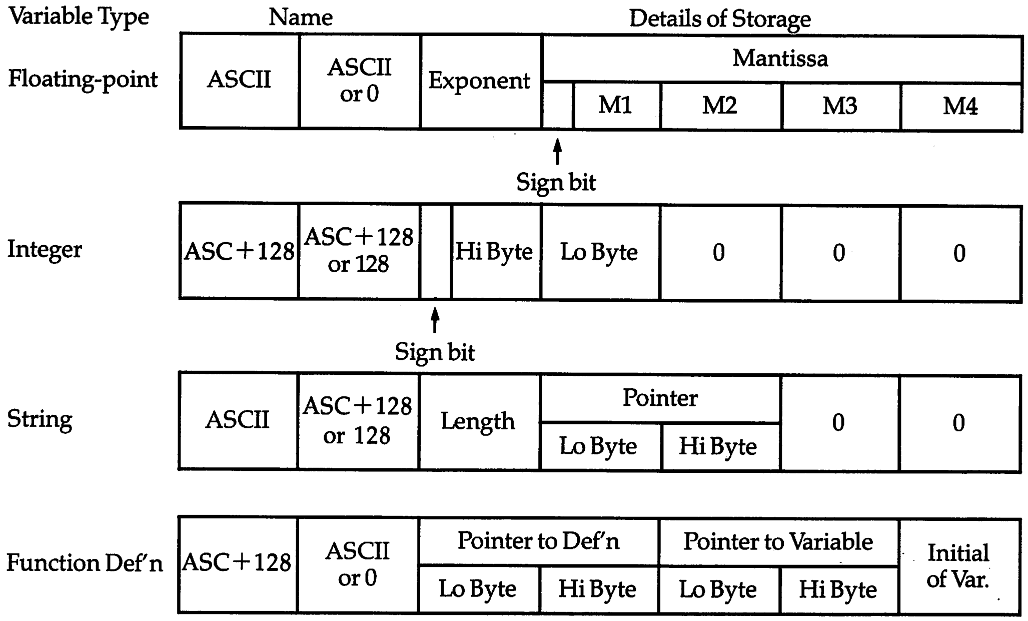
How BASIC Stores Its Variables

Suppose you type A=123 on the 64's keyboard and press RETURN; PRINT A will now print 123. Simple variables are allocated space after the program and before arrays. Even if there is no program, variables are stored in the same way, beginning after the three zero bytes at the start of BASIC RAM. Because variables are stored when they're first used, the sequence in memory is the order in which they were encountered by the 64.

Four types of variables can be stored in this area: floating-point variables (X), integer variables (X%), strings (X$), and function definitions (DEF FN X(Y)). The name is always stored in two bytes (though the second may be a space character), and as Figure 6-2 shows, the four types are distinguished by the high bit of each letter (of the variable name) being set or unset. This obviously gives four permutations from each name and is 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 variable type identifier, which is converted from the BASIC type declarators %, $, and FN (and parentheses for arrays).

Each variable type is allocated seven bytes. This means that when BASIC looks for a simple variable, it always adds a constant offset of seven as it searches the table, thus minimizing search time. However, with this scheme three bytes are wasted with integer variables, two are wasted with strings, and one is wasted with function definitions. Here are some of the different ways that the seven bytes ranging in value from 0 to 255 are interpreted:

Figure 6-2. Storage of Simple Variables



Floating-point, or real variables (X). The value is held in five bytes to an accuracy of one part in 2↑31 (about two billion).

These numbers are subject to rounding errors. Floating-point storage is discussed in more detail later, for 64 owners interested in insuring accuracy in financial or other calculations.

Integer variables (X%). Integer variables are held in signed, two-byte form, within the range −32768 to +32767. The following formula, which allows for the sign bit, gives the value of the integer variable:

(HI AND 127)*256 + LO + (HI>127)*32768

For example, HI=0 and LO=100 correspond to 100; HI=255 and LO=156 represent −100. The two expressions add to 0 with overflow.

Strings (X$). Strings cannot be fitted into seven bytes. To allow freedom in assigning strings (without needing to specify their lengths, as some languages require), they are stored dynamically in RAM. Three of the seven bytes allotted to a string variable are relevant; two are wasted. One byte holds the length of the string; LEN (X$) simply PEEKs this value. Another pair of bytes points to the starting address 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 length 0.

Most strings are stored in the area of free RAM high above the BASIC program and variable storage. String storage begins at the top of the BASIC program space, and grows downward. As strings are redefined (or as new string variables are defined), they fill progressively lower memory locations 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 the BASIC program, 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 it is stored after BASIC. Again, this has consequences which will be examined shortly. For the moment, note that because of this internal storage feature:

10 DIM X$ (200): FOR J=0 TO 200: X$ (J) = "1234567890": NEXT

uses 2000 bytes less memory than the same program with:

X$(J)="12345"+"67890"

In the first case, every string pointer points to the same string inside the BASIC program. In the second case, every pointer points to a separate ten-byte string stored above the program. Any string with any element of calculation is stored after BASIC (for example, one defined via INPUT or GET or even by A$=B$).

Function definitions. These appear in the variables table, too, and it's quicker to store them here than to search the whole program for a definition.

Like strings, function definitions store the solid information elsewhere. A function definition has two pointers, one pointing to the defining formula and one pointing 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

creates two entries in the variable table (X and of the function definition Y). Actually, the formula pointer holds the address of the equal 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 don't lend themselves to the normal seven-byte memory allocation.

All three array types—real, integer, and string—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 arrays are stored.

Figure 6-3. Storage of 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 (like sorting or merging). The general approach is:

```
AL=PEEK(49): AH=PEEK(50)
```

Then DIMension a new array and use it in the program. When finished with the array, delete it with:

```
POKE 49,AL: POKE 50,AH
```

This method simply 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 (DIMension) 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, but:

```
X(11)=1
```

gives a *?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 first two bytes are the array name, followed by the two-byte offset. The offset is the length of the entire array, including numeric data or string pointers; however, it excludes strings, which are stored elsewhere.

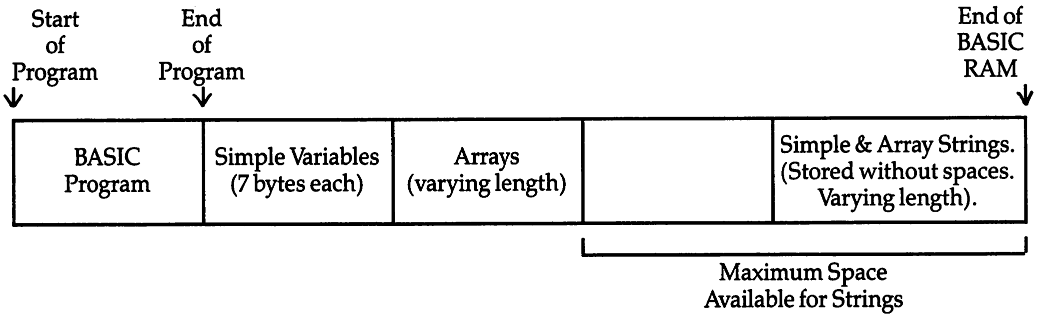
The number of dimensions is the number of subscripts: A(X) is one-dimensional, A(X,Y) is two-dimensional, and so on. The number of elements in each dimension must be stored as well, requiring two bytes, to allow for arrays like A(500) with more than 256 items. DIM+1 in Figure 6-3 is the number of elements dimensioned, since 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 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 shows a typical BASIC program during a run, followed by its variables and arrays. This illustrates how memory sections are allocated for each different kind of BASIC information.

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

Consequences of BASIC's Storage Methods

A number of consequences follow from these methods of storage. Because strings and arrays are of particular significance with serious programs, it's worthwhile to explain them thoroughly. If you understand them, you'll be able to write better programs.

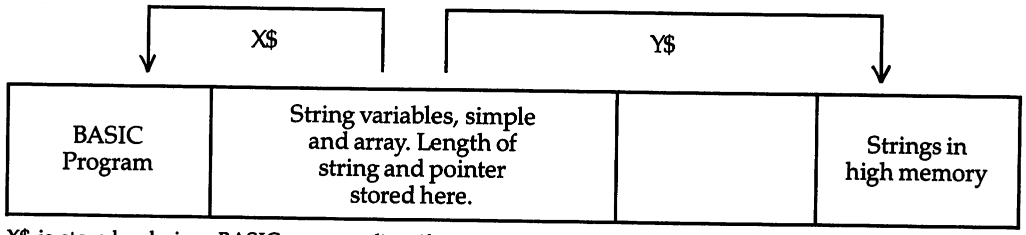
String storage. All strings are stored as ASCII characters in ascending order in memory. For instance, the program lines below show how a pair of pointers (S and E, for Start and End) delimit any current string in memory, and how a string is held in conventional sequence in ASCII.

```
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
```

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, and it is important to note that if a string is redefined, the old string is still left behind in memory; redundant data like this is called *garbage*.

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.

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$="HELLO"). Strings which must be built from other strings are stored in high memory (for example, 20 Y$=H$+Z$).

Loops to input data in a controlled way, using GET, do something very similar. They 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.

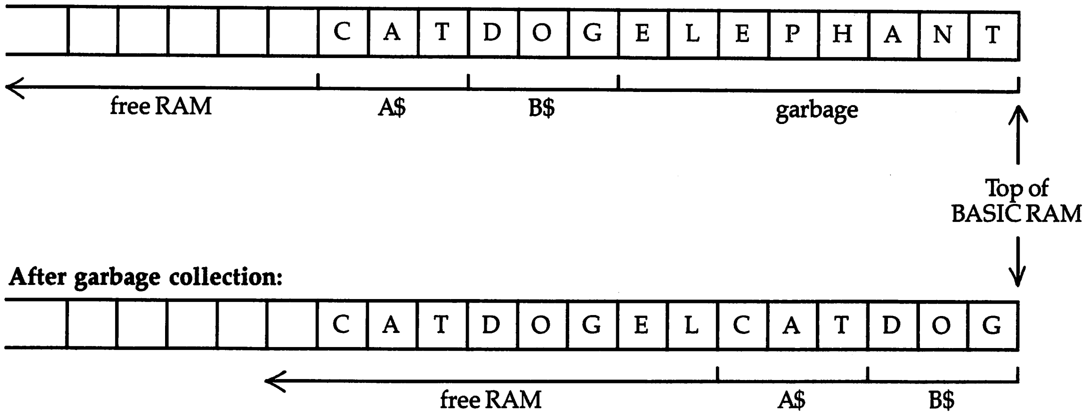
Corruption of data in RAM. Provided the end of BASIC is correctly set, this will not be a problem. For example, user-defined characters are popularly stored from 12288 ($3000). Provided POKE 55,0: POKE 56,48: CLR is executed early in the BASIC program, and it is shorter than 10K, the program will run perfectly because all strings are separated from the graphics definitions.

Garbage Collection

Programs using many strings are subject to *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.


C A T D O G E L E P H A N T
free RAM
A$
B$
garbage
Top of BASIC RAM
After garbage collection:
C A T D O G E L C A T D O G
free RAM
A$
B$

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 on the order of 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 64 takes roughly 0.00075 seconds times the number of strings squared to free memory. The actual relationship is a quadratic, while this is only an approximation. For instance, 100 strings take 0.9 seconds, 200 take over three seconds, 300 take over seven seconds, and so on.

Note that, during garbage collection, the keyboard locks in an apparent hang-up. This is normal; if a long ML routine runs, the RUN/STOP key has no chance to work. RUN/STOP–RESTORE will interrupt the collection if you find it necessary. In practice, you'll be likely to encounter garbage collection only if you're using string arrays; 100 ordinary strings will cause an occasional delay of less than a second. Try the following example, which calculates the time required to perform an FRE(0); as stated above, this uses the garbage collection routine.

```
10 INPUT D:DIM X$(D):FOR J=1 TO D:X$(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 can help (by shifting the delay to an acceptable period). It's also possible to do a limited FRE on part of the strings, altering the pointers at 55 and 56 down or 53 and 54 up.

Calculating Storage Space

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 economical. If you have a large amount 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 10 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, less time will be necessary. Second, if BASIC finds a new simple variable and there are arrays in memory, the array table has to be moved up in memory.

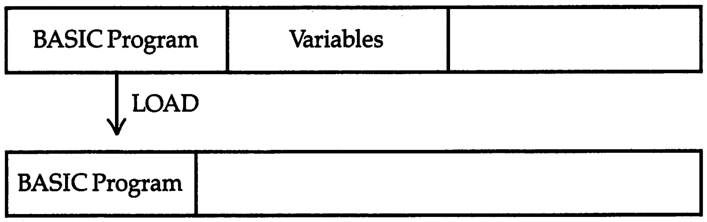
DIM is usually the most efficient way to define variables. It operates on simple variables just as it does on arrays. A statement like DIM J, X, S%, M$, X$(23) has the same effect on BASIC as searching for each variable, not finding it, and therefore positioning it with its default value of zero or the null string after the program.

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 economical 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 *chains*. The next tape (or another disk program) is loaded, generally into normal BASIC memory, overlaying the program which performed the LOAD. Automatically, a GOTO is executed which points to the first line of the LOADed program, so the new program runs while retaining variables from the earlier program. In this way, extremely long BASIC programs can still be run; for example, a series of visual screens in hi-res mode could be chained to provide an interesting advertising display. 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 material toward the end will be corrupted. For a complete solution to this problem, see OLD, later in 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, if used, 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 64 is the fact that numbers as they are printed don't always 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 example yields 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 be used to 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).

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

Floating-point variable values are stored in five bytes (see 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 high bit holds the sign, corresponding with the minus flag of the 6510 chip.

The 31 bits that hold the mantissa span a range of 1 to 1.9999. . . which, when multiplied by the exponent (in the form 2 to the nth power), 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 number stored in this form into a more understandable 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 64 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 negative numbers have the sign bit set. *Note also that an exponent of zero always indicates a zero value with this number system.*

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^{-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. 13.2681/8 is equal to (1.6585125), and the fractional portion is the number stored by the 31 bits in the mantissa:

```
.6585125   * 128 = 84.2896
.2896      * 256 = 74.1376
.1376      * 256 = 35.2256
.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, or bits, 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 64, 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, . . . $1/2^{31}$, 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 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 some of the more useful methods. The keyboard and some aspects of screen handling are also included here, as they are 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 a specific function. The program "Micro-Scope" (from the preceding chapter) allows you to watch the first two of these in action.

Input buffer. Program 6-4 demonstrates the use of the input buffer.

Program 6-4. Using the Input Buffer

```
10 N$="FORX=1TO5:PRINT X:NEXT X" + CHR$(0)
20 FOR J=1 TO LEN(N$)
30 POKE 511+J,ASC(MID$(N$,J)): NEXT
40 POKE 781,255: POKE 782,1
50 SYS 42118
```

An ASCII string, terminated by a null (zero) 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 command in the buffer, after first setting a pointer to $01FF (one less than the start of the buffer). The buffer would also be executed if the end of the program were reached, or if an END statement were encountered, but using the SYS shown in Program 6-4 is often more useful.

Keyboard buffer. It's easy to show that the 64 has a queuing system for keystrokes. The short routine:

1 GET X$: PRINT X$: FOR J=1 TO 2000: NEXT: GOTO 1

prints characters which have been typed faster than the computer can process them. Up to ten characters can be stored here. You can POKE 649 ($289) to change this, but if the value exceeds 10, you could corrupt some pointers (it is usually possible to use up to 15, however).

Location 198 ($C6) independently stores the number of characters in the buffer. POKE 198,0 therefore causes all characters to be ignored; it has the same effect as FOR J=1 TO 10: GET X$: NEXT. The key combination, SHIFT–RUN/STOP, puts LOAD, carriage return, RUN, carriage return into this buffer, using a routine at $E5EE.

Many examples in this chapter (AUTO, DELETE, LIST) rely on this buffer. The following short routine shows how POKEs into the buffer work. This is another important programming technique.

10 DATA 72,69,76,76,79
15 FOR J=631 TO 635:READX:POKE J,X:NEXT
20 POKE 198,5

POKEing one or more RETURN characters, CHR$(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 6-5, 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 allows the entire string to be input 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 6-5. Using a Quote Before INPUT

```
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)
     : NEXT
1020 POKE 198,P: POKE 631,34: INPUT X$
1030 PRINT X$: FORJ=1 TO 1000: NEXT: GOTO 1000
```

Program 6-5 moves the characters along the buffer, just as the 64's operating system does.

A more exotic use is to transfer BASIC programs to the 64 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 64'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; 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 64 has 1K of RAM at the low end of memory for its own use, but some isn't allocated and can be used safely. Locations 251–254 ($FB–$FE), 679–767 ($02A1–$02FF), 784–787 ($0310–$0313), 820–827 ($0334–$033B), and 1020–1023 ($03FC–$03FF) are available. The second of these areas is 95 bytes long; the tape buffer has 192 bytes, but 820–1023 are free for noncassette users (204 bytes). BASIC does not use the 4K of RAM from 49152–53247 ($C000–$CFFF), which is also free for ML programs or other purposes.

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 incremented, 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 with power sources, and between VICs and 64s, but the maximum error will not be more than one part in 33,000 (a couple of minutes a day).

Disabling RUN/STOP and RUN/STOP–RESTORE

Blocking out the RUN/STOP key is a useful way to provide some program security, to guard against accidental use of SHIFT–RUN/STOP (which will cause the 64 to try to load and run a program), and to keep the user from exiting a machine language program inappropriately.

Four software methods are given below. Remember, however, that if your computer goes into an infinite loop with RUN/STOP disabled, you may have to turn your 64 off to correct the problem. Be sure to include a subroutine to reenable RUN/STOP.

To disable both RUN/STOP and RUN/STOP–RESTORE (method 1): POKE 808,54: POKE 809,188. To reenable, POKE 808,237: POKE 809,246. This is one of the best methods, since it leaves the clock working, disables both RUN/STOP and RUN/STOP–RESTORE, doesn't affect tape operations, and leaves LIST working normally.

To disable both RUN/STOP and RUN/STOP–RESTORE (method 2): POKE 808,234. To reenable, POKE 808,237. This simple POKE disables both RUN/STOP and RUN/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 RUN/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 RUN/STOP: POKE 788,52. To reenable, POKE 788,49. This POKE modifies the interrupt vector so that it bypasses the Kernal routine to increment the clock and check RUN/STOP. However, it doesn't disable RESTORE. Tape operations defeat this procedure; during READing from tape the interrupt sequence is reset, and RUN/STOP breaks into the program. The jiffy clock is turned off by this POKE. RUN/STOP can also be disabled with POKE 808,239.

To disable RUN/STOP–RESTORE only: POKE 792,193. To reenable, POKE 792,71. This alters the NMI (Non-Maskable Interrupt) vector so that it returns without having any effect.

How RUN/STOP works: The RUN/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 RUN/STOP key is called. That routine looks at location $91 (145), and if the highest bit is low—that is, if the contents equal $7F (127)—RUN/STOP is currently being pressed. ML programmers can therefore check RUN/STOP with JSR $FFE1:LDA $91:BPL.

LIST and RUN also use the Kernal routine that tests the RUN/STOP key, which is why a listing or a running program can be stopped. Tape LOAD and SAVE also use it, as does RESTORE.

The Kernal routine at location 65505 ($FFE1) jumps to the address vectored in locations 808 and 809. Method 1 changes this destination from $F6ED to $BC36. The ML it finds there is simply LDA #$1/RTS. This value insures that RUN/STOP will never occur.

Function Keys

The simplest programming method is to use a simple GET; the range of ASCII values for f1–f8 are 133, 137, 134, 138, 135, 139, 136, and 140. Program 6-6 is a BASIC loader which enables all eight function keys to be defined with individual strings up to 32 characters long.

Program 6-6. Function Keys

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
Ø DATA 32,253,174,32,158,173,32,141,173,32,247,183
  ,136,152,1Ø,1Ø,1Ø                          :rem 2Ø4
1 DATA 1Ø,1Ø,133,253,169,29,133,254,32,253,174,32,
  158,173,32,143,173                          :rem 12
```

```
2 DATA 160,0,177,100,240,22,170,200,177,100,133,25
  1,200,177,100,133                         :rem 179
3 DATA 252,160,0,177,251,145,253,200,202,208,248,1
  38,145,253,169,28                         :rem 228
4 DATA 141,144,2,169,75,141,143,2,96,24,165,215,23
  3,132,201,8,144,3                         :rem 213
5 DATA 76,72,235,170,189,124,28,133,253,169,29,133
  ,254,160,0,177,253                        :rem 40
6 DATA 240,237,201,95,240,6,32,210,255,200,208,242
  ,166,198,169,13,157                       :rem 71
7 DATA 119,2,230,198,208,216,0,64,128,192,32,96,16
  0,224                                     :rem 152
10 REM PROGRAMMABLE FUNCTION KEYS FOR THE 64
                                            :rem 146
20 REM TYPICAL EXAMPLES OF SYNTAX:          :rem 113
30 REM SYS 40448,1,"THIS IS FUNCTION KEY 1":CLR
                                            :rem 134
40 REM SYS 40448,2,"[CTRL-RED]":CLR         :rem 41
50 REM SYS 40448,3,"LIST 50-100←":CLR ← ADDS RETUR
   N                                        :rem 79
60 REM SYS 40448,5,"LOAD"+CHR$(34)+"$"+CHR$(34)+",
   8":CLR:REM LOADS DIRECTORY               :rem 205
90 REM S1=49192 TO PUT ROUTINE AT $C000     :rem 101
100 POKE 56, PEEK(56)-2: CLR: REM LOWERS MEMORY BY
     512 BYTES                              :rem 63
110 S=PEEK(56):S1=256*S                     :rem 24
120 FOR J=S1 TO S1+131:READ X:POKE J,X:NEXT
                                            :rem 202
130 POKE S1+22,S+1:POKE S1+65,S:POKE S1+90,S:POKE
    {SPACE}S1+94,S+1                        :rem 19
140 FOR J=S1+256 TO S1+511:POKE J,0:NEXT:REM SET A
    LL FN KEYS NULL                         :rem 174
150 PRINT "{CLR}USE SYNTAX:-                :rem 19
160 PRINT "SYS" S1 ",N,STRING:CLR"          :rem 199
170 LIST 20-60                              :rem 201
```

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 6-6 is first 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. The program has REM statements which include examples.

Typically, SYS 40448,5,"NAME": CLR calculates the starting point where the string is to be stored and copies it there. CLR sets BASIC pointers correctly. (This example makes f5 print NAME on the screen.)

The vector at $028F–90 is central to the method. Provided it isn't changed and provided the strings and ML aren't overwritten, the function keys will operate as programmed indefinitely.

The software is written so that the left-arrow key can be used to insert a carriage return into the keyboard buffer:

SYS 40448,1,"LIST 100-300←":CLR

This defines function key 1 to type LIST 100–300 and press RETURN so that those lines will be listed. Typical applications of user-defined keys are POKEs into hard-to-memorize locations, SYS calls to routines in memory, and printouts of current values of variables.

Keyboard

Commodore keyboards are very reliable. The 64 keyboard has 66 keys, which can be pried off and replaced. It is possible to rearrange the keys, perhaps into German GWERTZ style or the Dvorak layout with ergonomically arranged letters. The software which decodes the keyboard can allow for this. Single-key entry of BASIC keywords is possible, and as illustrated above, the function keys can be programmed to output strings.

The CTRL key delays screen scroll and acts with keys 0–9 to set the major colors and reverse on and off. CTRL-A through CTRL-Z also output CHR$(1) through CHR$(26); for example, CTRL-E sets white characters, CTRL-N lowercase, and CTRL-S cursor home. Also, CTRL-H locks the keyboard mode and CTRL-I unlocks it, something otherwise tiresome to achieve. A few other keys act with CTRL, too. An especially useful combination is CTRL-[(open bracket, or SHIFTed :), which is CHR$(27). This is a special printer code, called ESCape, which triggers features like underline, double strike, and emphasized.

Reading the keyboard. When the 64 is operating normally, its interrupt routine performs several functions. The clock is updated and a RUN/STOP key location updated, the cursor may be flashed, the cassette motor is turned off unless a flag is set, the keyboard is scanned, and the keyboard buffer is updated on a keypress. This can be traced in a ROM disassembly. When an interrupt is generated, the 6510 finishes its instruction, saves a few values, and jumps to the address in $FFFE at the very top of memory. It is tested to see if it's an interrupt (not a BRK instruction), and jumps to the address held in $0314–$0315 (788–789), which is normally $EA31. The first instruction is JSR $FFEA (UDTIM: increment clock, save RUN/STOP status), followed by screen and tape handling, and a call to the Kernal's SCNKEY routine at $FF9F which, as the label implies, scans the keyboard.

Only 64 keys of the 66 are detectable by the keyscan. RESTORE is unreadable; it causes a non-maskable interrupt (NMI) by making a circuit when pressed and isn't decoded with the other keys. SHIFT/LOCK is the other undetected key. The 64 keys are wired into a matrix of eight rows and eight columns, which can be scanned using only two bytes. The arrangement of keys within the matrix is different on the VIC-20 and the 64.

Briefly, two ports of CIA 1, at $DC00 (columns) and $DC01 (rows), are examined for bits set to 0. These bits are almost all 1; only the grounding action of a key sets a zero value. This is why Table 6-3 below has values of 127, 191, etc.—the bit patterns are 01111111, 10111111, and so on.

When the scan begins, $DC00 is set for output and $DC01 for input, by the default configuration of their data direction registers $DC02 and $DC03 (which hold $FF and $00, respectively). In turn, the row register is rotated to take one of 8 values, and each bit of the column is tested each time. A bit will have a 0 value if its corresponding key is pressed, or 1 if the key is not pressed. The value is $FF when no keys are pressed. A counter increments with each loop; it is this counter value (not the value read from the matrix) which is stored when a keypress is found. Since there are eight rows and columns, 64 different values are theoretically possible, and the 64 uses all of these.

Reading more than one key *simultaneously* is generally possible only with ML; Chapter 13 has an example. Even with ML, it is difficult to detect more than 2 of the 64 keys at once; for example, with keys 9 and K pressed, neither the + nor the : key can be distinguished.

Table 6-3. Decoding the Keyboard

Contents of $DC01 (Row)

Contents of $DC00 (Column)

	$7F (127)	$BF (191)	$DF (223)	$EF (239)	$F7 (247)	$FB (251)	$FD (253)	$FE (254)
$7F (127)	RUN STOP	Q	C=	Space	2	CTRL	←	1
$BF (191)	/	↑	=	Right SHIFT	CLR HOME	;	*	£
$DF (223)	,	@	:	.	−	L	P	+
$EF (239)	N	O	K	M	0	J	I	9
$F7 (247)	V	U	H	B	8	G	Y	7
$FB (251)	X	T	F	C	6	D	R	5
$FD (253)	Left SHIFT	E	S	Z	4	A	W	3
$FE (254)	⇕	f5	f3	f1	f7	⇔	RETURN	INST DEL

The column is always set to 127 (at $EB42 in ROM, to be precise) apart from during the actual reading; so RUN/STOP can be detected merely by testing whether $DC01 has bit 7 clear. Left SHIFT, X, and several other keys can easily be checked like this, too. Machine language is necessary to read the keyboard. The following BASIC program and ML routine illustrate the way the rows and columns interact. At this level, SHIFTed and unSHIFTed keys aren't distinguished.

```
10 POKE 808,234 :REM DISABLE STOP
20 INPUT "COLUMN",C :REM USE 127, 191, ETC
30 POKE 829,C
40 SYS 828: GOTO 40
```

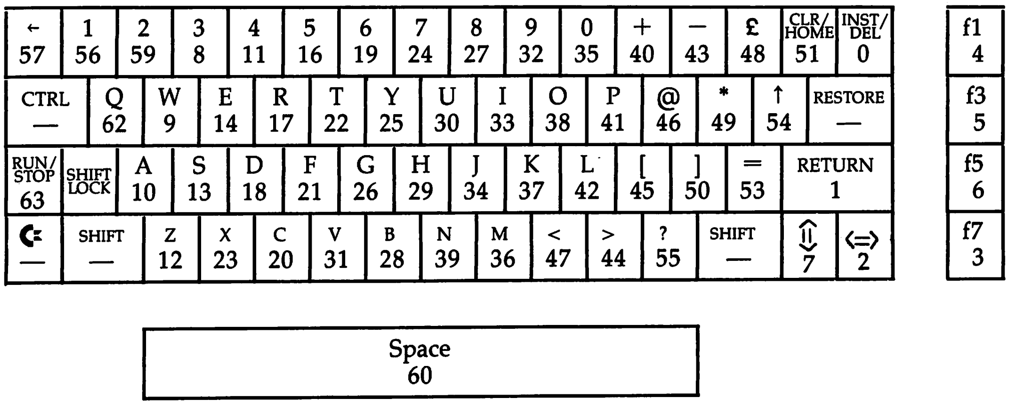
```
$033C  LDA  #$00
$033E  STA  $DC00 ;SET COLUMN
```

```
$0341  LDX  $DC01 ;FETCH ROW
$0344  LDA  #$00
$0346  JSR  $BDCD;PRINT X
$0349  LDA  #$0D
$034B  JMP  $FFD2 ;NEW LINE
```

The keyboard cannot function correctly if you change the default values in the data direction registers $DC02 (56322) or $DC03 (56323). POKE 56322,253 is an example; this turns off 3,4, left SHIFT and several letters, so something like P SHIFT-O 56222+100,252+2+1 is needed to reenter the normal value of 255. Joysticks and paddles are wired together with the keyboard; games port 1 connects with the rows, port 2 with the columns. This is the reason that a joystick in port 1 generates apparent keypresses, and pressing certain keys has the effect of closing switches in the port 1 joystick.

Decoding the keyboard. When a key is pressed, an identifying number from 0 to 63 is stored in $CB (203). The value of the most recent keypress is in $C5 (197), and comparing the two shows whether a new key is being pressed. This prevents a long string of X's from being entered, for example, if you press X and hold it down too long. PEEKing $CB or $C5 (see Figure 6-9) is a very useful way to test for key depressions without using GET, and it has the advantage of working at all times. The default value is 64, which is stored if no keys are pressed, so PEEK(203) = 64 means no key is pressed. In practice, $C5 and CB are indistinguishable. Note that the values stored in these locations are not ASCII codes.

Figure 6-9. Keyboard Values Stored in $CB and $C5



SHIFT, Commodore, and CTRL. Location $028D (653) stores information on these keys, which are assigned the values 1, 2, and 4, respectively. The following line of BASIC checks the value and prints it to the screen:

```
FOR J=1 TO 9E9: PRINT PEEK(653): NEXT
```

The values add up, so this location will contain a number from 0 to 7, depending on which of the three keys is pressed. For example, if SHIFT and the Commodore key are held down simultaneously, the value should be 3 (since 1 + 2 = 3).

RUN/STOP. Location $91 (145) stores a copy of the normal keyboard row and is used to indicate that RUN/STOP is pressed. If you PRINT PEEK(145) in a loop with STOP disabled, the result is 127.

Converting these values into ASCII is the final stage. The 64 has four character tables built into ROM, for unSHIFTed, SHIFTed, Commodore key, and CTRL sets, starting at $EB81, $EBC2, $EC03, and $EC78, respectively. Each is 65 bytes long and converts $C5's contents into ASCII values, making allowance for the SHIFT or Commodore keys. The last byte in each table contains $FF, the value used to show that no key is pressed.

After copying the key value (0–64) into $CB, an indirect jump is executed via $028F to $EB48. This routine's function is to set the address in ($F5) to point to one of the four keyboard matrices, depending on the SHIFT keys in effect. It has a subsidiary function, that if $0291 (657) is less than 128, SHIFT-Commodore key will switch graphics sets from lowercase with uppercase to uppercase with graphics, or vice versa.

Now, with $F5 pointing to one of the four keyboard tables, the routine at $EAE0 is entered, and the key's ASCII value is determined. The keyscan routine then deals with keyboard repeats, cursor control, and other special keys. Finally, the ASCII value of the key is put into the keyboard buffer (if room is available there), and control returns to BASIC until the next IRQ interrupt.

The vector at $028F can be changed to point to your own RAM routine so that keys can be intercepted and their effect changed. This is an ML technique only; an earlier example shows how to program the function keys to print out strings of characters.

Intercepting Keys

Generally, if you wish to intercept keys to trigger an activity, like printing a message, the technique is to test either $CB or $D7 for your key, or keys. $D7 (215) stores the ASCII value of the last key printed to the screen. Check $CB if you're only concerned with the physical key, and check $D7 if you need to distinguish between unshifted and shifted keys. Left SHIFT can be tested with $91; SHIFT, CTRL, and Commodore key can be separately detected at $028D. Jump to $EB48 if the looked-for key isn't pressed; end your own routine typically with JMP $EB42, which reloads $DC00 with the correct value for the next keyscan. In this way, keyboard processing is exactly as normal, SHIFT keys and all, apart from your own specially inserted routine. This very simple example changes the 64's background color whenever the left-arrow key is pressed:

```
($028F should point to the starting address)
      LDA  $CB
      CMP  #$39   ;Left-arrow key
      BEQ  LABEL
      JMP  $EB48  ;Continue normal keyboard operation
LABEL INC  $D020  ;Increment border color register
      JMP  $EB42  ;This exit doesn't print ←. Allows repeat.
```

As a more complex example, consider how we might print BASIC keywords with single keystrokes (assisted by SHIFT keys, CTRL-Commodore key plus a key, or some other combination). There are about 64 keywords, so just about every key can be assigned a unique word. The example program below uses the Commodore key in combination with other keys to print BASIC words. For example, the Commodore key with A prints RUN. One of the points to watch for is the debouncing feature just after LABEL—without this, the words would print repeatedly, instead of only once.

The following assembler listing shows the flow of the single key BASIC entry system. The vector at $028F should point to the start of this routine:

```
       LDA  $028D
       CMP  #$02     ; Is Commodore key pressed?
       BEQ  LABEL
EXIT   JMP  $EB48    ; Continue as usual if not.
LABEL  LDY  $CB      ; Now look at ordinary scanned keys;
       CPY  #$40
       BEQ  EXIT     ; exit if no 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  $A09C,X  ; BASIC words are stored from $A09E.
       BPL  LOOP     ; Look for high bit set,
       DEY           ; and, when found, decrement
       BNE  LOOP     ; Y until it counts down to 0.
PRINT  INX
       LDA  $A09C,X  ; Load and print consecutive characters.
       BMI  LAST     ; end signaled by high bit set.
       JSR  $FFD2    ; CHROUT
       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  $EB48    ; Continue normal keyscan.
```

The routine described above is listed below in the form of a BASIC loader.

Program 6-7. Single-Key Keyword Entry

```
Ø DATA 173,141,2,2Ø1,2,24Ø,3,76,72,235,164,2Ø3,192
  ,64,24Ø,247,196
1 DATA 197,24Ø,243,132,197,2ØØ,162,Ø,232,189,156,1
  6Ø,16,25Ø,136
2 DATA 2Ø8,247,232,189,156,16Ø,48,7,32,21Ø,255,48,
  245,16,243,41
3 DATA 127,32,21Ø,255,76,72,235
2Ø S=49152:REM S=828 WORKS ALSO
```

```
30 FOR J=S TO S+54:READ X:POKE J,X:NEXT
40 POKE 656,S/256:POKE 655,S-INT(S/256)*256:REM PU
   'TS S INTO ($028F)
```

Variable S in Program 6-7 controls the place in memory into which the routine is POKEd; any free RAM area is acceptable since the routine is relocatable.

Redefinition of Keyboard

If you wish to redefine the keyboard, the most elegant way to do this with the 64 is to transfer BASIC and the Kernal into RAM, as explained in Chapters 5 and 8. This leaves the keyboard tables free to be redefined in any way you choose. The four tables, at 60289 ($EB81, unSHIFTED), 60354 ($EBC2, SHIFTED), 60419 ($EC03, Commodore key), and 60536 ($EC78, CTRL), each have 64 bytes and a terminating byte that holds $FF. Unused keys, such as CTRL-function keys, also appear as $FF and can be redefined. Special keyboards can be saved and reloaded later, and turned on or off at will by switching ROM out or in.

As a simple example, these four POKEs *with RAM under ROM activated* cause f1 and f5 to output CTRL-Black and CTRL-White, and f2 and f6 to output CTRL-RVS/ON and CTRL-RVS/OFF:

POKE 60293,144: POKE 60295,5: POKE 60358,18: POKE 60360,146

It is also possible to cause BASIC to process keys differently; for example, CTRL-G could be used to set a graphics mode. This of course involves work beyond simple key redefinition. Another possibility is to extend the character set for printing foreign characters to the screen.

The keyboard as a device. The keyboard is treated as device number 0 by the operating system. We can open a file to the keyboard and treat it as an input device:

10 OPEN 5,0: REM OPEN FILE 5 FOR USE WITH DEVICE 0 (KEYBOARD)
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 won't now give *?EXTRA IGNORED,* because a file is considered open, but parts of a string may be lost. The normal question mark prompt isn't printed.

Repeat Keys

Location 650 controls which keys, if any, repeat when the key is held down. For example, the following POKEs easily modify the way the keyboard functions:

POKE 650,0 :REM SPACE BAR AND CURSOR CONTROL KEYS REPEAT.
POKE 650,64 :REM NO KEYS REPEAT.
POKE 650,128 :REM ALL KEYS REPEAT.

With BASIC in RAM, the rate of repeat and delay before repeat takes place can be controlled by POKEing 60189 and 60138, respectively. Otherwise, an easy way to alter the repeat rate is to change the rate at which interrupts occur.

Location 652, the repeating key delay register, can be used to step from one value to another through a range of values which may be very large. Program 6-8 is a simple example of the method.

Table 6-4. Summary of Keyboard Data Locations

$91	145	RUN/STOP key/record
$C5	197	Newest key pressed
$C6	198	Number of characters in keyboard buffer
$CB	203	Previous key pressed
$D7	215	ASCII value of key pressed *
($F5)	(245)	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, cursor; 64 no keys; 128 all keys)
$028B	651	Repeat delay (4 to 0, so 12 repeats per second) *
$028C	652	Repeat countdown (16 to 0, so 1/4 second before repeat) *
$028D	653	SHIFT, Commodore key, CTRL register (1, 2, and 4 respectively)
$028E	654	Previous configuration of SHIFT, Commodore key, and CTRL
($028F)	(655)	Vector enabling user-written keyboard intercepts *
$0291	657	Commodore key SHIFT mode switch enable/disable (128 disables)
$EA31	59953	Interrupt sequence comes here *
$FF9F	65439	Kernal routine to read the keyboard (SCNKEY) also #EA87

* Means ML is required to use this.

Program 6-8. Fast Step

```
10 REM USE + OR - KEY
100 GOSUB 1000: T=T+INC: PRINT T: GOTO 100
1000 POKE 650,128: REM ALL KEYS REPEAT
1010 GET X$: IF PEEK(652)>0 THEN INC=0
1020 IF X$="+" THEN INC=INC+1:IF INC>20 THEN INC=2
     0
1030 IF X$="-" THEN INC=INC-1:IF INC<-20 THEN INC=
     -20
1040 IF X$="" GOTO 1000
1050 RETURN
```

The above program can cause some strange results, so take the disk out of the disk drive (if you are using one) before running the program. The plus key increases the value printed to the screen at an increasing rate if the key is held down, but shorter keypresses step forward in smaller increments. The minus key steps down. Chapter 13's "SIDmon" has a similar method for controlling values put into sound locations; the values there can range from almost 0 to 65535.

RAM Areas Free for BASIC Use

Sections of free RAM. Locations 49152–53247 ($C000–$CFFF) make up 4K of RAM which is isolated from BASIC and therefore invaluable for storing ML routines or data. A VIC-20, fitted with 8K expansion, can have a similar isolated RAM area, but generally other CBM machines have to alter BASIC memory or use a tape buffer to store ML. Note that, if BASIC is switched over to RAM, a software reset will treat the whole area up to $D000 as RAM (Chapter 8 explains in depth). The only other problem with $C000–$CFFF is that everyone's ML tends to start at $C000. If several routines are to coexist, some will have to be relocated; Chapter 9 shows how to do this.

The 64 also has RAM beneath ROM from $A000 to $BFFF and from $E000 to $FFFF. However, this RAM cannot be used by BASIC without ML, except in the sense that BASIC has some redundancy, so parts of BASIC or the Kernal could be used for storage. For example, if BASIC and the Kernal are both in RAM, locations $E4B7–$E4D9 can be used freely; on a larger scale, $E264–$E377 can be used, provided the trigonometric functions (COS, SIN, TAN, ATN) are avoided.

Use LDA #$35:STA $01 to flip out the BASIC and Kernal ROMs, and access their underlying RAM. To switch the ROMs back in, use LDA #$37:STA $01.

Because of the way the VIC chip works, this hidden RAM can also store graphics information, provided the screen is moved to the same general area. For example, the screen might start at $C000, and up to 352 sprite definitions or 11 sets of character definitions could be stored simultaneously, ready for access. Chapter 8 provides more information about how to access the RAM under ROM.

Small areas of free RAM. The 64 has 1K of RAM at the start of memory which is largely allocated for BASIC. In zero page, the four bytes from 251 to 254 ($FB–$FE) are left untouched by BASIC. Locations 2–6 are rarely used (predominately by ML number conversion programs), and 247–250 ($F7–$FA) are used for RS-232 pointers. The stack (256–511, $100–$1FF) can be used with caution at the low end; 318 ($13E) is a safe starting point if tape is going to be used. However, don't store ML in the stack if you don't understand it; for example an *?OUT OF MEMORY ERROR* may delete your ML.

679–767 ($2A7–$2FF) has 89 spare bytes; 784–787 ($310–$313) has 4; and 820–1023 ($334–$3FF) has 204, of which the tape buffer (828–1019, $033C–$03FB) takes 192.

Screen

The screen is treated as device number 3, so files can be opened to the screen for input and output with a statement like OPEN 3,3. This provides INPUT without the normal prompt and can occasionally be useful in lines like the following one, which reads the top line from the screen, subject to the usual rules governing INPUT.

```
OPEN 3,3: PRINT {HOME};: INPUT #3,X$
```

Screen handling can be complicated; each ASCII value has to be converted into a POKE value, and if it has some special purpose like clearing the screen or setting reverse mode, a subroutine must carry this out. Color RAM and start-of-screen must be allowed for.

ML programmers can trace this process at $FFD2, the Kernal routine CHROUT, which prints a character (which jumps to $F1CA and $E716). From here, 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.

Sixteen bytes just after the screen (usually 2024–2039, $7E8–$7F7), and 16 color RAM nybbles, can store ML or data. (Eight bytes of sprite shape pointers follow this.) Clearing the screen leaves the area intact; but obviously incorrect POKEs to the screen area can easily overwrite this storage position.

Table 6-5 is a quick reference list for screen locations and ROM routines.

Table 6-5. Summary of Screen Locations and ROM Routines

Screen Locations

$C7	199	Reverse flag (0 reverse off, character reverse on)
($C9)	(201)	Cursor row and column for input from screen
$CC	204	Cursor flash (0 flashes cursor, e.g., with GET)
$CD	205	Cursor countdown (from 12 to 0)
$CE	206	Character under cursor
$CF	207	Cursor blink phase (0 or 1)
$D0	208	Input from screen/keyboard (flag is 3 or 0)
($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 (39 or 79)
$D6	214	Cursor's row
$D9–$F2	217–242	Table of screen line links, 4–7: Line continues; $84–$87: It doesn't
($F3)	(243)	Color RAM address
$0286	646	Color code (0, black, through 15, light gray) in use
$0288	648	High byte of start of screen (usually 4)

ROM Routines:

$E544	58692	Clear screen
$E5A8	58792	Set VIC chip to normal values
$E632	58930	Input from screen (or keyboard) comes here
$E8CB	59595	Converts CHR$(color) in A into 0–15
$E8EA	59626	Scrolls screen up 1 row
$E981	59777	Scrolls screen down 1 row (contents of 677 may affect this)
$E9FF	59903	Clear entire row (e.g., POKE 781,X:SYS 59903, when X is 0–24)
$EA13	59923	Plots character and color in screen. A=Char, X=Color (0–15)
$EA24	59940	Finds color RAM relevant to current cursor position

Dictionary of Extensions to BASIC

The 64's BASIC is limited, lacking many useful commands built into some other BASICs. Many can be simulated easily, though. The examples that follow are grouped under headings of typical keywords, which indicate their functions.

These are BASIC subroutines, which must be run as usual, or machine language routines called by SYS. *The actual words listed (such as APPEND) are not new keywords in this case; they will not by themselves activate any of these routines.* A *wedge* altering a BASIC input vector is necessary to incorporate new keywords. The recently published book *COMPUTE!'s Machine Language Routines for the Commodore 64* includes most of these aids, as well as character and sprite editing systems—all in ML. The following examples, though, will make it easier to understand how the commands work.

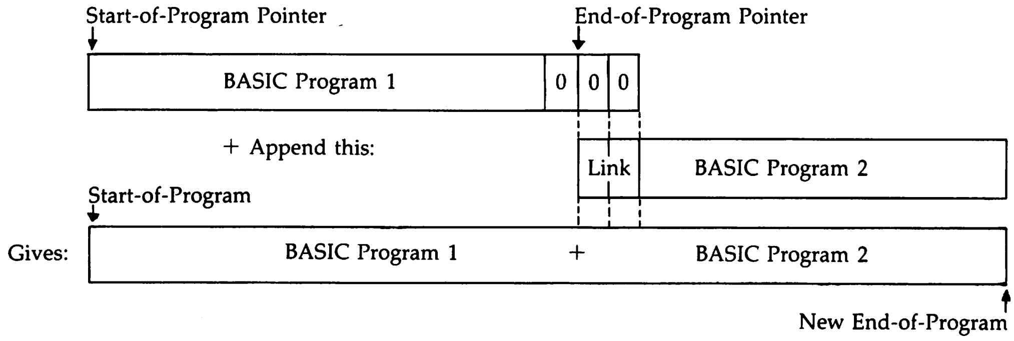
APPEND

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 as well (see Chapter 15). And tape files can be appended, but since the 64 has only one tape port, the process is more difficult.

BASIC programs are easy to append because the LOAD address is easily altered. Standard subroutines with high line numbers can be put onto the end of programs without the usual 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 a link address of two zero bytes (following the normal end-of-line zero byte) signifies the end of the program. If the new program loads and overwrites the double 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 44,8 to start BASIC at $800. Perfectly appended BASIC should be the 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 BASIC, which automatically generates line numbers. Many utility packages include this command. This example 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. Obviously, this would be better if implemented in ML, wedging (for example) into the main BASIC loop, IMAIN, at $A480.

Program 6-9. Auto

```
60000 INPUT "ENTER START, INCREMENT";S,I
60010 PRINT "{CLR}{2 DOWN}"; S;: POKE 204,0
60020 GET X$: IF X$="" GOTO 60020
60030 PRINT X$;: IF ASC(X$)<>13 GOTO 60020
60040 PRINT "S=" S+I ":I=" I ":GOTO60010": PRINT "
      {HOME}";
60050 POKE 631,13: POKE 632,13: POKE 198,2: END
```

BLOCK LOAD and BLOCK SAVE

The 64's LOAD and SAVE commands are designed solely for the benefit of users of BASIC. They automate BASIC program storage and recovery in a way which is transparent. Programs load into the correct area of memory 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 to tape or disk, normal saving won't 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 by the BASIC line-linking routines.

Note that ML monitors (like *Supermon)* have commands like .S "NAME",01, 1000,2000 (save the contents of $1000–$1FFF to tape and call it NAME) and .S "NAME",08,1000,2000 (save the same data to disk) to perform block loads and saves.

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 44,48: POKE 45,0: POKE 46,64: SAVE"NAME",1,1. This will save data from locations $3000 to $3FFF, because the value in the start-of-BASIC pointer is changed to $3000 and the value in the end-of-BASIC pointer is changed to $4000. As far as the 64 is concerned, this becomes the correct area to save. (Note that the last byte at $4000 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.

However, there may be problems in using this technique from within a BASIC program since it's necessary to restore the pointers after use.

This short routine illustrates one technique:

```
1000 SYS 57812 "NAME",1,1 : REM SET PARAMETERS FOR LOAD
1010 POKE 193,0: POKE 194,48 : REM $3000 IS START ADDRESS ...
1020 POKE 174,0: POKE 175,64 : REM $4000 (−1) IS END ADDRESS.
1030 SYS 62957 : REM PERFORMS SAVE
```

SYS 57812 takes in the parameters which saving or loading needs: the device number, name length, pointer to name, and secondary address. You can watch its effect—using the following BASIC line, alter the parameters to see the effect on these locations:

SYS57812"HI",8,1:PRINT PEEK(186);PEEK(183);PEEK(187)+256*PEEK(188);PEEK(185).

BLOCK LOAD. To load a machine language routine into memory, the easiest way is simply to use LOAD "NAME",1,1 (or ,8,1). Within a program, a flagging technique is needed to avoid the automatic chaining feature:

```
0 IF X=1 GOTO 20
1 X=1: LOAD "NAME",1,1
2 REM CONTINUE FROM HERE...
```

Listed below is a simple, trouble-free method of loading, which works within programs without interrupting their flow:

```
1000 POKE 147,0 : REM THE LOAD/VERIFY FLAG. 0 IS LOAD
1010 SYS 57812 "SCREEN",8,1 : REM SETLFS IN THE KERNAL SETS PARAMETERS
1020 SYS 62631 : REM NOW LOAD
```

Program 6-10 saves a screen of information and reloads it. The technique can be useful for many applications, including games, notepads and word processors (which allow viewing two files on alternate screens).

Program 6-10. Screen Save and Load

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
1 REM SAVES SCREEN,COLOR RAM AND VIC CHIP:RUN PERF
  ORMS SAVE,RUN 500 RELOADS                  :rem 71
7 REM SCREEN ASSUMED TO START $400          :rem 214
8 REM WRITE 3 NEW FILES TO DISK               :rem 5
9 REM FOR TAPE,USE 1 INSTEAD OF 8            :rem 77
10 SYS 57812 "@:SCREEN",8,1:POKE 193,0:POKE 194,4:
   POKE 174,0:POKE 175,8                     :rem 58
15 SYS 62957                                :rem 114
20 SYS 57812 "@:COLOR",8,1:POKE 194,216:POKE 175,2
   20:SYS 62957                              :rem 62
30 SYS 57812 "@:VIC REGISTERS",8,1:POKE 194,208:PO
   KE 175,209:SYS 62957                      :rem 98
50 END                                       :rem 60
499 REM LOAD BACK 3 FILES                    :rem 97
500 POKE 147,0:SYS 57812 "SCREEN",8,1:SYS 62631
                                            :rem 252
510 SYS 57812 "COLOR",8,1:SYS 62631          :rem 91
520 SYS 57812 "VIC REGISTERS",8,1:SYS 62631
                                            :rem 119
600 GOTO 600:REM DISPLAY SCREEN TILL 'RUN/STOP' KE
    Y                                        :rem 47
```

Lines 10–30 save $0400–$07FF to disk under the name "SCREEN". Line 20 saves the corresponding color RAM, from $D800 to $DBFF, and line 30 saves the VIC registers. If user-defined characters were used, they must be saved, too. Between them, they completely define any picture starting at $0400; for example, the border color and background are controlled by the VIC chip.

Lines 500–520 reconstruct the picture. Try typing this program into a 64 attached to a disk drive, then put a few random colored characters on the screen. RUN will store the screen's information on disk, if connected. Clear the screen; type RUN 500. You'll see the screen reconstruct itself.

Tape is equally simple: Just change each device number from 8 to 1, and remove the redundant @: from within the filename. The screen files are always stored and reloaded in the correct sequence.

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 a time, and the menu is reentered on exit from any called program. Commodore 64 BASIC (and PET/CBM and VIC BASICs) chains whenever LOAD takes place inside a program. A LOAD is automatically followed by RUN without CLR (to retain previous variables).

Although 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 tape 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 are no LOADing messages unless the cassette button isn't pressed. (To try this example with a disk drive, use 30 LOAD"SECOND PROGRAM",8.)

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. POKE 53281,SC:POKE 53280,BC has the same effect as COLOR SC,BC. POKE 646,LC sets the color of the letters output to the screen with PRINT.

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, the *object code.* Typically, it will LIST as a single SYS command, which is followed by a large (but unlisted) ML program. Compilation is on a higher plane of sophistication than the other utilities discussed here, but a short discussion is justified.

Briefly, any compiler of an unstructured language like BASIC must first build up a table of all the program's variables and arrange a position for each of them in memory. Strings need pointers and will be subject 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 BASIC, could be replaced by just five bytes of machine language. The line 1000 FOR J=0 TO 1000: POKE SC+J,32: NEXT is a loop, and is the sort of BASIC which a compiler has great difficulty turning into efficient ML. Tiny compilers working with a restricted set of BASIC (to save the effort of implementing every command) also exist and are interesting educational tools.

Compilers invariably need disk drives, for speed and because multiple files are required. If you don't have a disk drive, someone else can compile your BASIC for you, in which case, the result will need to be transferred to tape; some compilers have a feature to permit this.

Typical commercial compilers are *PETSPEED* and the *DTL* compiler. Each has a limit on the size of BASIC program that is compilable and the number of variables in it. You may find it necessary to shorten a very long program to compile it. The ML object code is often longer than the original BASIC because it has to include a long library of standard subroutines.

Computed GOSUB and Computed GOTO

These functions use a formula or label, instead of a number, for their destination line. Some computer languages allow the use of GOSUB VALIDATE to perform a subroutine called VALIDATE. Obviously, statements like this are likely to be more readable than BASIC's GOSUB 10000, or wherever. (Don't confuse computed destinations with ON-GOTO, which provides a choice of destinations according to the value just after ON.) Any parts of BASIC using computed destinations can't usually be renumbered by a utility.

Program 6-11 shows how computed GOTO and GOSUB can both be implemented on the 64.

Program 6-11. Computed GOTO and GOSUB

```
90 FOR J=40960 TO 49191: POKE J,PEEK(J): NEXT
91 FOR J=57344 TO 65535: POKE J,PEEK(J): NEXT
92 POKE 1,53
100 DATA 32,138,173,76,247,183
110 FOR J=0 TO 5: READ X: POKE 58551+J,X: NEXT
120 POKE 43169,183: POKE 43170,228
```

With BASIC in RAM, all that's required is to intercept BASIC at $A8A0 and include a routine to evaluate a formula, rather than simply take in an ASCII line number. This version copies BASIC from ROM to RAM and stores the extra ML into a part of the RAM which isn't used by the copied-down BASIC.

While ordinary BASIC runs a little slower in RAM, you can now have expressions like GOTO DATE or GOSUB CHECK, where perhaps DATE=1000 was previously defined, and line 1000 starts the routine called DATE.

CRUNCH (and UNCRUNCH)

The idea of *crunching* a program is to delete as much of it as is possible without altering its function negatively, with the aim of increasing BASIC's execution speed and decreasing the memory required to store it. Conversely, *uncrunching* means spacing a program out to make it more readable. For example, if you wish to decipher someone else's crunched program, a utility which lists each instruction on separate lines and puts in spaces may well help legibility. (See LIST in this chapter, and see also Chapter 8.)

The rationale of CRUNCH is that REM statements, spaces, and short lines slow the BASIC translator by making it waste time jumping past spaces, switching to new lines, and so on. CRUNCH doesn't usually speed up programs a great deal, but many programmers like to pack their programs into the smallest space possible. Combined with renumbering lines starting at 0 and incrementing by ones, and adding an extra line or two of DIM statements to order the main variables, you can reduce the execution time of your BASIC programs noticeably.

Crunching should remove REMs, but if these are referenced by GOTO or GOSUB, they should either be retained or the reference changed to point to the next line. It should remove all spaces not within quotation marks, but avoid confusion between keywords and variables (X = T AND U after crunching could be confused with the function TAN).

As many lines as possible should be merged together. Lines spanning more than 255 bytes are unreliable, since many BASIC pointers are single bytes (those for DATA, for example). So the longest line is generally limited to 250 BASIC characters. A line in the program might be referenced by GOTO or GOSUB, and this should be handled properly. Lines of such lengths cannot be simply keyed in; they must be POKEd in, then the links must be readjusted.

To make the CRUNCH even more interesting, it could renumber from 0 upward in steps of one, reduce all variable names to a single character (if possible), and remove spare semicolons from PRINT statements. It might modify CHRGET to remove its test for spaces (see Chapter 10), slow the rate of interrupts (or temporarily stop them), and remove wedges which intercept BASIC and usually slow its operation.

DEEK

This double-byte PEEK returns the value in two consecutive addresses, assuming they follow the 6510 convention of low byte, then high byte. Use this formula:

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

DELETE (DEL)

This command allows deletion of unwanted BASIC lines. DEL *a–b* is the syntax, which is similar to that of LIST (except that DEL alone should not delete everything). DEL seems to have been omitted from Commodore's original BASIC specifications. This subroutine in BASIC, designed to sit at the end of a program, works by searching for line numbers within a specified range, then deleting the line by using a trick with the keyboard buffer, which simulates entry of the line number at the keyboard.

Program 6-12. Delete

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
61000 INPUT "DELETE FROM, TO"; L,U:A=PEEK(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 610
      10                                 :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 198
      ,3:END                             :rem 0
```

Line 61020 skips through link addresses until a line number in range is found. Line 61030 stops either out of the range or at the end of a program. Lines 61040–50 print to the screen, and the rest of the subroutine simulates keypresses for HOME, RETURN, and another RETURN.

DOKE

This double-byte POKE puts a value from 0 to 65535 into any two adjacent addresses, assuming the standard 6510 convention of low byte/high byte. 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

Screen dump. This prints a duplicate of the screen onto paper. 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 others may not.

Program 6-13A is a BASIC screen dump which assumes the uppercase character set and correctly prints all characters, including SHIFTed and Commodore key symbols; however, the quotation mark character is not processed properly by some printers:

Program 6-13A. Screen Dump

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
40010 OPEN 4,4:CMD 4                          :rem 253
40100 FOR J=0 TO 24:FOR K=0 TO 39:X=PEEK(1024+J*40
      +K)                                     :rem 9
40110 IF X>128 THEN X=X-128:PRINT CHR$(18);:rem 87
40120 IF X<32 THEN PRINT CHR$(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:CLOSE 4:END :rem 142
```

Line 40010 opens a file to the printer. Line 40100 starts a loop, which PEEKs every individual screen location, and line 40110 looks for reverse characters; use the reverse character appropriate to your printer, if one is available. CHR$(18) is Commodore's reverse character printing signal, with CHR$(146) also needed (in line 40160) to turn it off. Chapter 17 has more information on the use of printers.

Variable dump. This lists the current values of variables. Often array variables are ignored by these routines, because they are more difficult to handle. 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; we'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, which is 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 (like VARPTR) and printed with its name. This last method is the one used by Program 6-13B.

Program 6-13B. Variable Dump

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
0 DATA 165,22,72,160,32,162,11,189,185,3,157,32,2,
  202,16,247,140,41,2                          :rem 45
1 DATA 192,32,240,28,140,36,2,208,23,142,34,2,32,2
  25,255,234,234,208                          :rem 253
2 DATA 8,104,133,22,104,104,76,116,164,162,48,142,
  35,2,169,32,133,122                          :rem 49
```

```
3 DATA 169,0,133,72,32,134,174,165,72,240,19,169,3
  2,133,122,173,34,2                          :rem 12
4 DATA 141,39,2,173,35,2,141,40,2,32,157,170,174,3
  5,2,232,224,58,144                           :rem 0
5 DATA 211,224,65,144,247,224,91,144,203,174,34,2,
  232,224,65,144,251                          :rem 12
6 DATA 224,91,144,171,160,36,204,41,2,240,174,144,
  139,200,208,136,34                           :rem 6
7 DATA 32,32,65,146,61,34,32,65,32,59         :rem 30
10 REM SYS 828 (DIRECT MODE) DUMPS NON-ARRAY VARIA
   BLES' VALUES                               :rem 199
20 REM RELOCATE BY POINTING 185,3 IN LINE 0 TO 34
   {SPACE}AT END OF LINE 6                    :rem 237
100 FOR J=828 TO 963:READ X:POKE J,X:NEXT   :rem 68
```

FIND

See SEARCH.

LIST

One of the most used commands in BASIC is LIST. Luckily, it can be modified easily. The two routines below are examples of modified listing. Program 6-14 creates a window on 12 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. Program 6-15 is in machine language; it alters LIST to expand the reverse characters of 64 listings into a more readable text form. Printer owners may like to list their programs in this format.

Window LIST. Append Program 6-14 to the end of your BASIC programs to use it. If you RUN 63000 with this subroutine in memory, it will list several lines on the screen—the number of lines listed can be adjusted in line 63010. Pressing the f1 key causes the listing to move upward past the stationary window, while pressing the f7 key causes the listing to move downward. Obviously, since any single logical line of BASIC can take two physical screen lines, 13 lines of BASIC may be too much for the screen to hold.

Lines 63020 and 63030 are printed on the screen, and they list several lines in white before returning to test for f1 or f7. The current starting line is the Mth line number, and subroutine 63300 scans the program, finding which line numbers to list. After LIST, the keyboard buffer is POKEd to simulate {CLR} RETURN {CLR} {DOWN} RETURN.

Program 6-14. Window LIST

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 REM WINDOW LISTER                          :rem 240
63000 DEF FN DEEK(L)=PEEK(L)+256*PEEK(L+1) :rem 33
63010 N=M+1:GOSUB 63300:L1=L:N=M+12:GOSUB 63300
                                              :rem 122
63020 PRINT "{CLR}{BLU}LIST" L1 "-" L:REM THIS LIS
      TS 12 LINES                             :rem 183
```

```
63030 PRINT "{BLU}M=" M ":GOTO 63200{WHT}":rem 122
63040 POKE631,19:POKE632,13:POKE633,19:POKE634,17:
      POKE635,13:POKE198,5:END                :rem 57
63200 GET X$:IF X$="" GOTO 63200              :rem 81
63210 IF X$="{F1}" THEN M=M+1:REM OR LARGER INCREM
      ENT                                     :rem 144
63220 IF X$="{F7}" AND M>0 THEN M=M-1:REM OR LARGE
      R                                       :rem 127
63230 GOTO 63010                              :rem 49
63299 REM FIND N'TH LINENUMBER OF BASIC       :rem 11
63300 J=0:L=FN DEEK(43)                       :rem 219
63310 J=J+1:IF J<N THEN L=FN DEEK(L):IF L>0 GOTO 6
      3310                                    :rem 130
63320 IF (L=0) OR (FN DEEK(L)=0) THEN L=63999:RETU
      RN                                      :rem 38
63330 IF J=N THEN L=FN DEEK(L+2)              :rem 194
63340 RETURN                                  :rem 224
```

BASIC can't be edited with Program 6-14 running—the entire process is under program control, and listing is all that's allowed. However, it would be possible to write a line-editing program with this method, plus parts of AUTO.

Legible LIST. Program 6-15 is a transparent ML program which locates itself in memory.

Program 6-15. Legible LIST

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
3 REM POKE 49259,5 TO CANCEL [SPC]            :rem 155
4 REM POKE 49260,5 TO CANCEL [SH-SPC]         :rem 92
10 FOR J=49152TO 49493:READ X:POKE J,X:NEXT
                                              :rem 226
20 SYS 49152                                  :rem 102
30 PRINT "{CLR}{WHT}SYS 49152 TOGGLES SPECIAL LIST
   ON/OFF"                                    :rem 201
500 DATA 173,7,3,73,103,141,7,3,173,6,3,73,11,141,
    6,3,96,8,133,254,152                      :rem 98
501 DATA 72,36,15,48,8,104,168,165,254,40,76,26,16
    7,162,0,232,189,80                        :rem 36
502 DATA 192,240,240,197,254,208,246,160,0,200,185
    ,118,192,201,91,208                       :rem 67
503 DATA 248,202,208,245,32,210,255,200,185,118,19
    2,201,93,208,245,32                       :rem 64
504 DATA 210,255,104,168,165,254,40,76,246,166,144
    ,5,28,159,156,30,31                       :rem 81
505 DATA 158,18,146,129,149,150,151,152,153,154,15
    5,147,19,148,20,145                       :rem 86
506 DATA 17,157,29,32,160,255,133,137,134,138,135,
    139,136,140,0,91,66                       :rem 78
507 DATA 76,65,67,75,93,91,87,72,73,84,69,93,91,82
    ,69,68,93,91,67,89                        :rem 111
```

```
508 DATA 65,78,93,91,80,85,82,80,76,69,93,91,71,82
    ,69,69,78,93,91,66                        :rem 101
509 DATA 76,85,69,93,91,89,69,76,76,79,87,93,91,82
    ,86,83,93,91,82,86                        :rem 123
510 DATA 83,32,79,70,70,93,91,79,82,65,78,71,69,93
    ,91,66,82,79,87,78                        :rem 93
511 DATA 93,91,76,32,82,69,68,93,91,68,32,71,82,65
    ,89,93,91,77,32,71                        :rem 76
512 DATA 82,65,89,93,91,76,32,71,82,69,69,78,93,91
    ,76,32,66,76,85,69                        :rem 96
513 DATA 93,91,76,32,71,82,65,89,93,91,67,76,82,93
    ,91,72,79,77,69,93                        :rem 95
514 DATA 91,73,78,83,93,91,68,69,76,93,91,85,80,93
    ,91,68,79,87,78,93                        :rem 111
515 DATA 91,76,69,70,84,93,91,82,73,71,72,84,93,91
    ,83,80,67,93,91,83                        :rem 80
516 DATA 72,45,83,80,93,91,80,73,93,91,70,49,93,91
    ,70,50,93,91,70,51                        :rem 54
517 DATA 93,91,70,52,93,91,70,53,93,91,70,54,93,91
    ,70,55,93,91,70,56,93                     :rem 209
```

Most of Program 6-15 consists of two tables, one of ASCII character values and the other of their translated form within brackets. Therefore, the program can easily be modified to allow for graphics characters or to output your own alternative forms. The ASCII values of the brackets [and] are 91 and 93. The ASCII values and special characters of the program in its current form are listed below:

Table 6-6. Legible LIST ASCII Table

[BLACK]	144	[ORANGE]	129	[CLR]	147	[PI]	255
[WHITE]	5	[BROWN]	149	[HOME]	19	[F1]	133
[RED]	28	[L RED]	150	[INS]	148	[F2]	137
[CYAN]	159	[D GRAY]	151	[DEL]	20	[F3]	134
[PURPLE]	156	[M GRAY]	152	[UP]	145	[F4]	138
[GREEN]	30	[L GREEN]	153	[DOWN]	17	[F5]	135
[BLUE]	31	[L BLUE]	154	[LEFT]	157	[F6]	139
[YELLOW]	158	[L GRAY]	155	[RIGHT]	29	[F7]	136
[RVS]	18			[SPC]	32	[F8]	140
[RVS/OFF]	146			[SH-SP]	160		

Program 6-15, activated by SYS 49152, modifies a LIST vector to point within the special ML routine, which checks all characters in quotation marks. This part of the program is quite small. The program then outputs the special characters in brackets. Printers can use this program, but lines containing special characters will be longer than usual. SYS 49152 also turns off the special listing function, acting as a toggle.

Chapter 8 has a short ML routine which checks for colons and is able to LIST separate statements on new lines. If you wish to modify LIST in your own way, but have little ML experience, Program 6-16, an outline BASIC program, will help; it reads BASIC with PEEKs and is easy to understand. Append it to BASIC when you want to use it. Add your own selection of special characters at the end of the program.

Program 6-16. BASIC LIST

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
63499 REM SET UP VALUES                          :rem 84
63500 A=2049:INPUT "LOWER, UPPER LINE NUMBERS";F,T
                                                 :rem 223
63510 DIM T$(76):FOR K=1 TO 76:READ T$(K):NEXT
                                                 :rem 220
63520 DEF FN DEEK(A)=PEEK(A)+256*PEEK(A+1):CM=39:R
      EM SET MAX LINE LENGTH                     :rem 84
63599 REM START NEW BASIC LINE                   :rem 246
63600 L=FN DEEK(A+2):X=FN DEEK(A):Q=0:IF X=0 OR L>
      T THEN END                                 :rem 61
63610 IF L<F THEN A=X:GOTO 63600                 :rem 212
63620 PRINT RIGHT$("{4 SPACES}"+STR$(L),5)" ";:CC=
      6:REM ALIGN LINENUMBERS                    :rem 138
63699 REM PEEK AND PRINT BASIC                   :rem 220
63700 FOR K=A+4 TO A+93:P=PEEK(K):REM P IS CURRENT
       CHARACTER                                 :rem 204
63710 IF CC>CM-7 THEN PRINT:PRINT "{6 SPACES}";:CC
      =6                                         :rem 125
63720 IF P=0 THEN PRINT:A=X:GOTO 63600:REM NEXT LI
      NE                                         :rem 17
63730 IF P=34 THEN Q=NOT Q:REM TOGGLE QUOTE FLAG
                                                 :rem 13
63740 IF Q THEN GOSUB 63900:NEXT:REM LOOK FOR SPEC
      IAL CHARACTERS                             :rem 41
63750 IF NOT Q AND P>127 THEN PRINT T$(P-127);:CC=
      CC+LEN(T$(P-127)):NEXT                     :rem 223
63760 PRINTCHR$(P);:CC=CC+1:REM CC COUNTS CHARACTE
      RS                                         :rem 175
63770 NEXT K                                     :rem 145
63799 REM KEYWORDS IN TOKEN ORDER                :rem 2
63800 DATA " END "," FOR "," NEXT "," DATA "," INP
      UT# "," INPUT "," DIM "                    :rem 51
63810 DATA " READ "," LET "," GOTO "," RUN "," IF
      {SPACE}"," RESTORE "," GOSUB "             :rem 98
63820 DATA " RETURN "," REM "," STOP "," ON "," WA
      IT "," LOAD "," SAVE "                     :rem 44
63830 DATA " VERIFY "," DEF "," POKE "," PRINT# ",
      " PRINT "," CONT "," LIST "                :rem 130
63840 DATA " CLR "," CMD "," SYS "," OPEN "," CLOS
      E "," GET "," NEW "," TAB("                :rem 152
63850 DATA " TO "," FN "," SPC("," THEN "," NOT ",
      " STEP ",+,-,*,/,↑," AND "                 :rem 99
```

```
63860 DATA " OR ",>,=,<," SGN "," INT","ABS"," USR
      "," FRE"," POS"," SQR"                    :rem 185
63870 DATA " RND"," LOG"," EXP"," COS"," SIN"," TA
      N"," ATN"                                 :rem 78
63880 DATA " PEEK"," LEN"," STR$"," VAL"," ASC","
      {SPACE}CHR$"                              :rem 115
63890 DATA " LEFT$"," RIGHT$"," MID$"," GO "
                                                :rem 61
63899 REM USER-DEFINABLE SPECIAL CHRS           :rem 14
63900 IF P=5 THEN PRINT "[WHT]";:CC=CC+5        :rem 28
63910 IF P=17 THEN PRINT "[UP]";:CC=CC+4         :rem 1
63920 IF P=18 THEN PRINT "[RVSON]";:CC=CC+7
                                                :rem 249
63930 PRINTCHR$(P);:CC=CC+1:RETURN              :rem 104
```

MERGE

A program that combines two BASIC programs into a single program with the lines sorted correctly is called a *merge*. Standard subroutines, for example, can be inserted without the need for reentering them. Many BASIC extension packages have MERGE. Because of the flexible way merging is done, this command can also perform some extra functions, such as loading PET/CBM tapes more easily into a 64.

Tape merge. The following procedure involves storing the subroutines to be merged as sequential files, not as tokenized programs, then reading them back using the keyboard buffer to simulate entry of each line.

Use this line to save a subroutine on tape as a sequential file:

OPEN 1,1,1,"NAME OF SUBROUTINE": CMD 1: LIST [OPTIONAL LOW-HIGH LINES]

When the cursor returns, type the following line to close the file and write the last portion of data to tape:

PRINT#1:CLOSE 1

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 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 drive, 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 {CLR} 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 6-17 alters BASIC, allowing it 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 6-17. Disk Merge

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
59ØØØ FOR J=4Ø96Ø TO 49151:POKE J,PEEK(J):NEXT:REM
       PUT ROM->RAM                          :rem 126
59Ø1Ø POKE 42231,56:POKE 42288,96:POKE 42585,96
                                             :rem 217
59Ø2Ø FOR J=83Ø TO 9ØØ:READ V:POKE J,V:NEXT
                                             :rem 159
6ØØØØ DATA 162,8,32,198,255,32,2Ø7,255,32,2Ø7,255,
      165                                    :rem 56
6ØØ1Ø DATA 1,41,254,133,1,16Ø,Ø,32,2Ø7,255,32,2Ø7
                                             :rem 77
6ØØ2Ø DATA 255,24Ø,32,32,2Ø7,255,133,2Ø,32,2Ø7,255
      ,133                                   :rem 81
6ØØ3Ø DATA 21,32,2Ø7,255,153,Ø,2,24Ø,3,2ØØ,2Ø8,245
                                             :rem 127
6ØØ4Ø DATA 152,24,1Ø5,5,168,32,162,164,76,79,3,165
                                             :rem 165
6ØØ5Ø DATA 1,9,1,133,1,32,89,166,76,128,164:rem 7Ø
```

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,"I": 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 4:00, and D=256: PRINT FN MOD (50000), which prints the low byte of 50000.

OLD

Originally, OLD was used to restore a program which had been inadvertently removed by NEW. However, the 64 offers two other important uses, which can be considered under the heading BASIC recovery.

Program 6-18. Old

```
1Ø FOR J=53ØØØ TO 53Ø25:READ X:POKE J,X:NEXT
2Ø DATA 169,148,141,Ø,16Ø,169,1,168,145,43,32,51,1
   65
3Ø DATA 165,34,1Ø5,2,133,45,165,35,1Ø5,Ø,133,46,96
```

OLD for program recovery. This restores BASIC because NEW leaves most of the program intact. NEW simply arranges pointers as though no program were present, and puts a zero link address at the very start of BASIC; it also sets GETCHR and the RESTORE pointers, clears variables, and closes files. Once Program 6-18 is in memory, to recover from an unwanted NEW, type SYS 53000: LIST. The program is restored, and 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 in 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 53000:CLR at the start of the new program will prevent corruption. (CLR is needed to remove garbage after the program, which may appear as pseudodata; it is not possible to recover the overwritten variables.)

OLD restores BASIC after SYS 64738. Old can be used after a hardware reset has occurred, returning the machine to its startup state. Both these routines leave BASIC RAM unaltered and in effect perform NEW, so SYS 53000 is just as effective as with NEW. Any BASIC program, including one which disables RUN/STOP and RESTORE, can be reset and recovered by this method if you have a hardware reset switch.

How OLD works. OLD makes use of the ROM routine which links BASIC lines. A nonzero value is POKEd into the first link address, correcting for the zero bytes NEW inserts, and JSR $A533 relinks the lines of BASIC. The end-of-program pointers must also be reset. OLD also corrects location $A000, without which a hardware reset can corrupt BASIC if it is stored in RAM.

BASIC version of OLD. A BASIC equivalent of the above is simple, but 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 42291: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 LIST properly.

ONERR

The 64 has an indirect vector at $300–$301 (768–769) to process error messages. Usually, this is set to $E38B and the actual error is dictated by the byte in the X register. POKE 781, *number from 1 to 30*: SYS 42042 will print a message to the screen.

ONERR usually works by specifying a line number to GOTO in the event of error. The advantage is that the program cannot crash, while the drawback is that processing ONERR properly is liable to take a lot of memory space and slow execution time.

PAUSE

There are two versions of this command: one waits for a timed delay and the other temporarily freezes BASIC or ML.

Timed delays are useful with some types of music programs. BASIC delay loops (FOR J = 1 TO 500: NEXT) work well, 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 solves the problem. The 64's internal clock is another obvious way to get accurate timing. The clock is stored in 160, 161, and 162, with 162 changing fastest. One short routine is POKE 162,X: WAIT 162,2↑N, which has a maximum delay of 128/60, or about two seconds. To explain the formula, note that WAIT stops until just one bit is set. So POKE 162,0: WAIT 162,64 delays until location 162 reaches 64, pausing for 64/60 seconds. The timing is reasonably constant, although the first POKE could occur any time between interrupts, so there's 1/60 second maximum difference in pauses. Unless the interrupt rate is changed, resolution below about 1/60 second isn't possible. Delays longer than about two seconds require the use of location 161. POKE 161,0: POKE 162,0: WAIT 161,2 pauses for 2*256/60 seconds (or about eight seconds).

The easiest implementation of a system pause (halting execution until some event occurs) is to intercept the 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 routine will do the trick with normal keys, for example, the left-arrow. You could modify the program to check the keyboard twice, so the key could toggle the function on and off, or to test for Commodore, SHIFT, or CTRL keys:

```
Send interrupt routine here:  PAUSE  JSR   $FF9F    ;SCAN KEYBOARD
                                     LDA   $C5      ;LOOK AT KEYPRESS
                                     CMP   #$39     ;CHECK FOR BACK-ARROW
                                     BEQ   PAUSE    ;PAUSE WHILE PRESSED
                                     JMP   $EA31    ;NORMAL IRQ ROUTINE
```

POP

This command discards a RETURN from the stack; this erases the effect of the previous GOSUB so that if RETURN is encountered, the address returned to will be that of the next-to-last GOSUB, or *?RETURN WITHOUT GOSUB* will be signaled. This is useful in providing a premature escape from BASIC subroutines, which otherwise causes problems. To explain, GOSUB causes the computer to store a return address on the stack. When the subroutine is finished, RETURN pulls this address off the stack, using it to resume execution at the correct spot in your BASIC program. If you repeatedly exit a subroutine without performing a RETURN (using GOTO, for example), the stack eventually fills up with unused return addresses, causing an *?OUT OF MEMORY ERROR*. You can cure this problem and others like it with Program 6-19 below.

POP is relocatable, but this version starts at 830 and is called by SYS 830 from within a program. RUN and test with SYS 830 in direct mode; you should get a *?RETURN WITHOUT GOSUB ERROR.*

Program 6-19. Pop

```
10 DATA 104,104,169,255,133,74,32,138,163,201,141
20 DATA 240,3,76,224,168,232,232,232,232,232,154,9
   6
30 FOR J=830 TO 852:READ X:POKE J,X:NEXT
```

POP mimics RETURN in all respects apart from the actual change in program control. With this routine in memory, use SYS 830 immediately before any premature exit from a subroutine.

A more thorough POP, using a part of CLR, clears away all loops and subroutines within a program by resetting the stack pointer, thus deleting all evidence of FOR-NEXT and GOSUB. Variable values, DATA pointers, and so on are retained. On an abort or escape, this routine cuts through any tangle of loops and subroutines. With the 64, machine language is required to perform this POP:

```
PLA        ; REMOVE SYS ADDRESS
PLA
JMP $A67E  ; ENTER CLR TO RESET THE STACK
```

In decimal, this looks like:

```
10 DATA 104,104,76,126,166
20 FOR J=830 TO 834: READ X: POKE J,X: NEXT: REM SYS 830 FOR THIS POP
```

PRINT @

This moves the cursor rapidly to any place on the screen, as specified by horizontal and vertical parameters (HTAB and VTAB are other forms of this command). Graphics in BASIC can often be much improved with one of these methods, in place of printing {HOME} and many cursor moves. The fastest versions contain their own ML routines and therefore require storage space. Slightly slower versions use ROM routines and are more convenient.

To use the fast ML version below, type in the lines and run the program. SYS 828,*H*,*V* takes in horizontal and vertical parameters and puts them into the Kernal PLOT routine vectored at $FFF0.

```
0 DATA 32,155,183,138,72,32,155,183,104,170,164,101,24,76,240,255
10 FOR J=828 TO 843: READ X: POKE J,X: NEXT
```

Although it is no simpler to do the same routine with BASIC, POKE 781,*V*: POKE 782,*H*: POKE 783,0: SYS 65520: PRINT "HELLO!" will work.

PRINT USING

Some computer languages allow you to specify the format in which numbers are printed. This 64 program allows easy and fast output in a variety of formats, (rounded to two decimal places, or including a leading $ symbol, for example). The overall length of the output (including leading spaces) is programmable, so lining up columns of figures is made simpler. Also, output can be directed to a printer.

Program 6-20. Print Using

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
Ø DATA 1,1Ø,2,32,162,Ø,221,Ø,1,24Ø,6,232,224,12,2Ø
  8,246,24,96,169,69,32,-162                :rem 113
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,-164,232,136,208,252,236,-16
  5,176,33,172,-165,169,0                    :rem 3
4 DATA 153,1,1,189,0,1,201,32,208,3,169,32,234,153
  ,0,1,202,16,6,173,-163,136                 :rem 114
5 DATA 16,244,136,16,231,169,0,133,97,160,1,132,98
  ,96,169,0,32,-162,144,240                  :rem 106
6 DATA 138,168,173,2,1,240,9,169,46,32,-162,144,2,
  138,168,152,170,202,16,181                 :rem 156
7 DATA 0,32,158,173,32,221,189,32,-148,32,30,171,9
  6                                          :rem 202
30 T=49318                                   :rem 253
40 L=T-166                                   :rem 11
50 FORJ=L TO T-1                             :rem 116
60 READ X%:IF X%<0 THEN Y=X%+T:X%=Y/256:Z=Y-X%*256
   :POKE J,Z:J=J+1                           :rem 198
70 POKE J,X%:NEXT                            :rem 2
80 GOTO 130                                  :rem 53
100 X%=L/256: Z=L-X%*256                     :rem 241
110 POKE 55,Z:POKE 53,Z:POKE 51,Z            :rem 93
120 POKE 56,X%:POKE 54,X%:POKE 52,X%         :rem 202
130 PRINT "{DOWN}SYS " L+153 " FOLLOWED BY ANY NUM
    ERIC                                     :rem 153
131 PRINT "EXPRESSION IN PARENTHESES"        :rem 79
132 PRINT "PRINTS FORMATTED VALUE.           :rem 118
134 PRINT                                    :rem 37
140 PRINT L{3 SPACES}"=DEC/INT FLAG"         :rem 239
150 PRINT L+1 "=OUTPUT LENGTH"               :rem 255
160 PRINT L+2 "=DEC. PLACES"                 :rem 0
170 PRINT L+3 "=LEADING CHRS"                :rem 116
180 PRINT L+98 "=+VE LEAD CHR"               :rem 72
190 PRINT "{DOWN}EG SYS" L+153 "(-1234.567) PRINTS
     -1234.56"                               :rem 227
200 PRINT "{DOWN}SET UP NOW WITH LENGTH 11, 2 DEC.
     PLACES, & LEADING SPACES."              :rem 156
210 PRINT "{DOWN}SAVE FROM" L "TO" T-1;      :rem 248
220 PRINT "($";:GOSUB 500:PRINT " TO $";:L=T-1:GOS
    UB 500:PRINT ")"                         :rem 64
230 END                                      :rem 108
500 L=L/4096:FORJ=1TO4:L%=L:L$=CHR$(48+L%-(L%>9)*7
    ):PRINTL$;:L=16*(L-L%):NEXT              :rem 210
510 RETURN                                   :rem 118
```

Once you enter, save, and run the program, the screen display should include this:

```
49152 = DEC/INT FLAG
49153 = OUTPUT LENGTH
49154 = DEC. PLACES
49155 = LEADING CHARACTERS
49250 = +VE LEAD CHR
SET UP NOW WITH LENGTH 11, 2 DEC. PLACES, & LEADING SPACES.
```

Executing a SYS 49305(X) will print the current value of X, formatted (where possible) in accordance with the values in the five locations listed.

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, this 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 BASIC's unmodified PRINT statement; this routine permits a substitute for the space character to be printed (for example, $), so all positive numbers will appear preceded by the substitute character.

Note that X is truncated; if you wish to round the output value to two decimal places, use SYS (7667) X + .005.

Using PRINT USING. Program 6-21 prints formatted columns of figures. Lines 20, 30–31, and 40 print the first, second, and third columns, respectively. Meaningful variable names should help to make the POKEs more understandable.

Program 6-21. Print Using Demo

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 PRNT=49305:SWITCH=49152:LNGTH=49153:DECPTS=4915
   4:CHAR=49155:LDGCHR=49250
15 FORJ=-10 TO 100 STEP 10:PRINT
20 POKE SWITCH,0:POKE LNGTH,4:POKE CHAR,42:POKE LD
   GCHAR,42:SYS (PR) J
30 POKE SWITCH,1:POKE LNGTH,7:POKE CHAR,32:POKE LD
   GCHAR,32
31 POKE DECPTS,4: SYS (PR) 1/(1+J)
40 POKE LDGCHAR,ASC("$"):POKE DECPTS,2: SYS (PR) 1
   00 + J
50 NEXT
100 REM SYS 49305 (400.00) ETC.
```

The central piece of machine language code in this routine follows:

```
JSR   $AD9E  ; INPUT AND EVALUATE A BASIC NUMERIC EXPRESSION
JSR   $BDDD  ; CONVERT BYTE IN ACCUMULATOR 1 INTO A STRING
JSR   $C012  ; SPECIAL ROUTINE (ADDRESSING MAY VARY) TO PROCESS
             ; NUMBER OUTPUT AT $100-$10C
JSR   $AB1E  ; PRINT THE STRING USING A (LOW), Y (HIGH) POINTERS
RTS          ; RETURN TO BASIC WITHOUT ANY OTHER ACTION
```

The idea is to print normally, except that the number, after being prepared for printing as a string, is edited. Most of this is identical to the 64 ROM routines, but the inserted subroutine processes the number as it is held in memory just before being printed. The program is designed to allow relocation of ML by altering the parameter T; it can, for example, be stored at the top of BASIC. Remember to protect it from BASIC by lowering the top-of-BASIC pointer.

RECONFIGURE

Chapter 5 explains how BASIC configures itself on switch-on. However, there are many ways memory can be allocated on the 64. The pointers at 43 and 44, and 55 and 56, show the entire BASIC area is normally $0800–$A000. To lower the top of BASIC memory to $8000, POKE 55,0: POKE 56,128. Now, CLR will reset all the string pointers correctly, but stored variable values will be lost. POKE 51,0: POKE 52,128: POKE 53,0: POKE 54,128: POKE 55,0: POKE 56,128 has the same effect, but retains variables, and is therefore sometimes better.

Program 6-22 allows the start or end of BASIC (or both) to be changed, so that PRINT FRE(0) returns different values from usual. The screen RAM can also be moved, within 1K boundaries; if it's moved to overlap BASIC, a program or its variables may be displayed in the screen, generally with odd side effects.

Program protection methods sometimes make use of this feature. For example, you can move the screen to $C000 and write ML starting at the normal screen area of $0400. When the ML is loaded, the screen fills with what is apparently garbage, but which is necessary to run the program. This makes a program relatively safe from being copied.

Another use is to simulate other machines, mainly the VIC-20 and CBM/PET. For example, the CBM/PET simulator in the Appendices moves BASIC and the screen to the CBM/PET positions and adds some other CBM-like features. All these examples keep BASIC in ROM; Chapter 8 explains how BASIC in RAM can be used to reconfigure BASIC more fundamentally.

Note that BASIC must have a zero byte at the position immediately before that indicated by the pointers 43 and 44. If it does not, NEW or RUN will cause a *?SYNTAX ERROR.*

Program 6-22. Reconfigure

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
800 S=2049:INPUT " START OF BASIC";S        :rem 24
802 E=40960:INPUT "{3 SPACES}END OF BASIC";E
                                            :rem 123
804 SC=1024:INPUT "START OF SCREEN";SC      :rem 248
999 VB=INT(SC/16384):VB=(NOT VB) AND 3      :rem 118
1000 POKE 648,SC/256:POKE 53272,(PEEK(53272) AND 1
     5) OR ((SC/64) AND 240)                :rem 150
1002 POKE 56576,(PEEK(56576) AND 252)OR VB:rem 164
1010 POKE 55,E-INT(E/256)*256                :rem 13
1015 POKE 56,E/256                          :rem 158
1020 POKE 43,S-INT(S/256)*256                :rem 39
1030 POKE 44,S/256                          :rem 166
1040 POKE S-1,0                               :rem 1
1050 PRINT "{CLR}"                           :rem 42
```

To make BASIC start at $1200 and end at $1400, with the screen at $2000, run Program 6-22, enter 4609, 5120, and 8192 at the prompts, and then enter NEW. Following this PRINT FRE(0) shows 509 free bytes, and POKE 8192,6: POKE 55296,5 prints a green *F* at the home position, showing that the POKEs have worked correctly.

Boots. The ability of a computer to load and run a program automatically is called *booting*. Some microsystems require the disk operating system (DOS) to be loaded in when the computer is turned on. The term came about because the computer is said to be "pulling itself up by its own bootstraps." The Commodore 1541 disk drive, however, uses proprietary software (the DOS is contained in the disk drive's ROM chips). And the 1541's built-in software does not autoboot. Nevertheless, the keyboard buffer and input buffer can be used to solve this deficiency. For example, the commands can be printed to the screen and the keyboard can be filled with the characters needed to input them; this, however, assumes that the position of screen memory doesn't change.

Tape boot. When using tape, autobooting is easy. Simply press SHIFT–RUN/STOP. If you want to see how to do the same from within a program, add line 1045 to Program 6-22:

```
1045 POKE 631,131: POKE 198,1
```

These POKEs have the effect of typing SHIFT–RUN/STOP (the ASCII value is 131). Since 198 holds the number of characters in the keyboard queue, POKEing a 1 into that location simulates a single keypress. Now, the program reconfigures BASIC, loads the next tape program, and runs it.

Disk boot. The keyboard queue can't easily hold much more than ten characters, which is insufficient to load a disk program since, unlike tape, a name is usually needed. LOAD"*",8:RUN in its short form just fits. One solution is to use the input buffer as in the following lines:

```
61 CLR :REM NEW NOT NEEDED AT END (AS NEW PROGRAM IS TO BE LOADED)
62 N$="LOAD" + CHR$(34) + "NAME" + CHR$(34) + ",8" + CHR$(0)
63 FOR J = 1 TO LEN(N$) : POKE 511+J, ASC (MID$(N$,J)): NEXT
64 POKE 198,3: POKE 631,82: POKE 632,213: POKE 633,13
65 POKE 781,255: POKE 782,1 :REM POINTER TO $01FF
66 SYS 42118 :REM INPUT LINE
```

Line 62 sets up a string ending with a null byte; this exactly mimics a line input from the keyboard. Line 63 POKEs the characters into the input buffer at 512 ($0200). Line 64 puts R SHIFT-U RETURN in the keyboard buffer, to cause the program to run after loading. Lines 65 and 66 process the line in the buffer, loading the program called "NAME".

REM

REM is, of course, one of the 64's normal statements. It deserves a place here because of the unique status of REM statements outside the normally 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, {CLR}, {HOME}, etc., can be inserted after an opening quotation mark. {DEL} (delete) characters can be used by opening up space inside quotes with the {INST} (insert) key. A hidden line can be created by following it with :REM" ", expanding the space in quotes, and filling the space with {DEL} characters, though this maneuver won't hide the line when it's listed on a printer.

You can use REM statements to produce colorful listings, too. For example, you could list the initialization section of the program in white, the main loop in yellow, and subroutines in other colors. This way you could find the section you wanted to view easily. To change the color of the listing, type REM " " and delete the second quotation mark, then press {RVS} (CTRL-9) followed by SHIFT-M. Next, press {INST} (SHIFT-DEL) once, and select the color by pressing CTRL or Commodore key and the correct numbered key. After this, press the RETURN key to enter the line. REM stores some characters differently inside quotes than outside. Thus, utilities which search for strings may not find them in REM statements.

Inserting characters into REM statements. REM is tokenized as 143 in decimal. The following short routine puts two RETURN characters immediately after REM in a REM line, and also immediately before the end of the REM line, so 100 REM** REMINDER COMMENTS * will list remarks neatly onto new lines.

```
63000 L=43
63010 L=PEEK(L) + 256*PEEK(L+1): IF L=0 THEN END:REM SKIP THROUGH LINKS
63020 IF PEEK(L+4)<>143 GOTO 63010:REM IF REM NOT FOUND TRY NEXT LINE
63030 POKE L+5,13: POKE L+6,13:REM POKE TWO RETURNS
63040 FOR J=L+5 TO 9E9: IF PEEK(J)>0 THEN NEXT:REM FIND END-OF-LINE,
63050 POKE J-1,13: GOTO 63010 :REM AND POKE ONE RETURN
```

Inserting reverse SHIFT-M within quotes adds a SHIFT-RETURN character with a similar effect. Other characters could include printer control characters to enhance REM statements, or color characters to list REMs in a different screen color.

Using REMs to store ML. As Chapter 9 explains in detail, BASIC can hold ML within REM statements. The data can simply be POKEd in. This can be very efficient, but there are two potential problems with the technique.

Zeros should not be used, because they will be treated as end-of-line markers if the program is edited, so the ML will be corrupted—a link address and line number will be inserted. This could be used, with care, as a security device. Generally, instead of LDX #0, use LDX #1:DEX.

The actual position in memory of the ML data must be known. The easiest method is to use a REM statement at the very start of the program, so the sixth byte from the initial zero byte is the start position. The ML routine must be relocatable to work with all BASIC configurations.

RENUMBER

Renumbering a BASIC program has some cosmetic advantages and is valuable where BASIC line numbers are too close to allow more BASIC to be added, or when a program is finished and you want to renumber by ones starting at line number 0 (which causes the program to run slightly faster). Program 6-23 is a short BASIC subroutine that changes line numbers only, between a selected range, by POKEing in new values.

Program 6-23. Renumber

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
60000 INPUT "RENUMBER FROM,TO ";L,H           :rem 40
60005 INPUT "START, INCREMENT ";S,I           :rem 49
60010 DIM L(600,2):A=2049:B=256:J=-1          :rem 94
60100 J=J+1:L(J,0)=PEEK(A+2)+B*PEEK(A+3):L(J,2)=A+
      4                                       :rem 250
60105 IF L(J,0)<L OR L(J,0)>H THEN L(J,1)=L(J,0):G
      OTO 60120                               :rem 31
60110 L(J,1)=S+R*I:R=R+1:NL=L(J,1)           :rem 136
60115 POKE A+2,NL-INT(NL/B)*B:POKE A+3,NL/B
                                             :rem 175
60120 A=PEEK(A)+B*PEEK(A+1):IF A>0 GOTO 60100
                                             :rem 128
60200 FOR K=0 TO J-30:A=L(K,2)                :rem 18
60205 P=PEEK(A):SP=0:IF P=0 THEN NEXT K :END
                                             :rem 219
60210 IFP<>137ANDP<>138ANDP<>141ANDP<>155ANDP<>167
      ANDP<>203THENA=A+1:GOTO 60205          :rem 184
60300 N=0:A=A+1:P=PEEK(A):IF P=32 THEN SP=SP+1:GOT
      O 60300                                :rem 188
60305 IF P=164 GOTO 60300                    :rem 231
60310 IF P<ASC("0") OR P>ASC("9") GOTO 60205
                                             :rem 187
60315 IF P>47 AND P<58 THEN N=10*N + (P-48):A=A+1:
      P=PEEK(A):GOTO 60315                   :rem 144
60320 FOR Q=0 TO J-30:IF N=L(Q,0) GOTO 60330
                                             :rem 240
60325 NEXT Q:PRINT "*** UNREFERENCED LINE IN" L(K,
      1):GOTO 60500                           :rem 1
60330 IF L(Q,0)=L(Q,1) GOTO 60500            :rem 145
60400 N$=STR$(L(Q,1)):O$=STR$(L(Q,0))         :rem 4
60405 D=LEN(N$):D2=LEN(O$):IF D<D2 THEN N$=CHR$(32
      )+N$:GOTO 60405                         :rem 21
60410 IF D>D2+SP THEN PRINT "{RVS}*** PUT" N$ " IN
      TO LINE" L(K,1):GOTO 60500             :rem 164
60415 X=A-D-1:FOR Q=2 TO D:POKE X+Q,ASC(MID$(N$,Q)
      ):NEXT                                 :rem 181
60500 IF P=32 THEN A=A+1:P=PEEK(A):GOTO 60500
                                             :rem 178
60505 IF P=44 OR P=171 THEN SP=0:GOTO 60300
                                             :rem 246
60510 GOTO 60205                              :rem 50
```

This BASIC utility is a four-parameter renumber; it allows a part of a program to be renumbered, leaving the rest alone, so that, for example, a subroutine between 2000 and 2500 can be tidied up, perhaps being renumbered from 2000 in steps of ten.

0800 8906 H 00 FF FCE2
2737

One difficulty with renumbering is that line numbers within programs are stored as ASCII strings, so if a renumbered line is different in length, the program's length may have to be changed. Another difficulty concerns syntax; Program 6-23 simply assumes correct syntax, mainly to use less space.

To use "Renumber," RUN 60000. You may renumber lines 0–59999, but not above. Lines 60000–60120 of the program build an array; L(J,0) holds original line numbers, L(J,1) holds new numbers, and L(J,2) holds pointers to the start of each line. The numbers at the start of each line are renumbered at this stage. J counts the number of lines in the program; not all these are needed, of course, since RENUMBER itself should be left alone.

Lines 60200–60210 scan all the relevant program lines, searching for keyword tokens, which are processed by the lines that follow. Line 60305 looks for TO; this allows GO TO to be renumbered, not just GOTO. Spaces after a keyword are counted, allowing variation in the renumbered line number lengths. Line 60320 searches for lines in the table and signals if they're not found. Lines 60400–60415 POKE in the new line number, where possible. And 60500 processes constructions like ON X GOTO 100,200 and LIST 10–30.

RESET

SYS 64738 resets the 64, giving a result similar to switching on the machine. RAM from $0 to $0400, except for the stack, is completely cleared out, and BASIC is in effect NEWed, but the rest of memory is untouched and BASIC can be recovered with OLD.

SYS 64738 is useful whenever the 64 has been reconfigured or pointers have been set in unusual ways. For example, after loading ML high in memory, RESET will leave it there by return to the normal condition of BASIC on startup. When BASIC is in ROM, a hardware reset (see Chapter 5) has the same effect as this software reset; other CBM machines behave similarly.

However, if BASIC is in RAM, SYS 64738 acts differently from a hardware reset and may show an unusually large number of bytes free, because the software SYS call, unlike hardware, doesn't necessarily switch BASIC into ROM, if the Kernal has been modified. Chapter 8 explains in depth.

Note that some CBM publications contain a wrong SYS call for this feature.

SEARCH

Searching BASIC is reasonably straightforward, given an understanding of the way it is stored in memory. The following ML search hunts for a match with the contents of the first BASIC line.

Program 6-24. Search

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
5 FOR J=830 TO 921:READ X:POKE J,X:NEXT    :rem 219
10 DATA 166,43,165,44,134,251,133,252,160,1,134,25
   3,133,254,177,253,240                     :rem 158
11 DATA 73,72,136,177,253,72,160,4,132,142,132,143
   ,177,251,201,34,208                        :rem 60
```

```
12 DATA 2,230,143,164,143,177,251,240,28,72,164,14
   2,177,253,240,15,104                     :rem 107
13 DATA 209,253,240,6,230,142,160,4,208,222,230,14
   2,208,226,104,104,170                    :rem 136
14 DATA 104,208,193,160,2,177,253,170,200,177,253,
   32,205,189,169,32,32                     :rem 120
15 DATA 210,255,201,0,208,231,96             :rem 25
```

Run Program 6-24, then enter 0 DATA and type SYS 830. All the line numbers of lines with DATA statements will list. You can search for lines containing the number 240 with 0"240; 0"SYS will find SYS as a word, not as a BASIC keyword. The ML relocates, and can be moved to any free RAM area.

SET

SET (and UNSET) are graphics commands in some BASICs which allow a point or small square to be drawn at any specified positions on the screen. Chapter 12 has a lot of information on this, including a high-resolution plotting routine.

SORT

Sorting means arranging a list in order, usually alphabetically or numerically. Many sorting methods exist, but only three major ones are discussed here: two BASIC sorts and one ML sort, which includes a demonstration to illustrate the syntax. The machine language version is far faster than BASIC.

BASIC sorts. The Shell-Metzner Sort is a fast sort, which is also easy to program. The version given in Program 6-25 sorts items 1 to *N* of an array dimensioned with A$(N). The sort is written as a subroutine to be added to your programs, and it assumes that array A$ 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 6-25. Shell-Metzner Sort

```
59010 M=N
59020 M=INT(M/2):IF M=0 THEN END
59030 J=1:K=N-M
59040 I=J
59050 L=I+M
59060 IF A$(I)>A$(L) THEN T$=A$(I):A$(I)=A$(L):A$(
      L)=T$:I=I-M:IF I>0 THEN 59050
59070 J=J+1:IF J>K THEN 59020
59080 GOTO 59040
```

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 (which can otherwise be a problem with BASIC) is not a factor.

Program 6-26 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 6-26. Tournament Sort

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 INPUT "SORT HOW MANY ITEMS";N:B=N-1:DIM N$(B),I
   (2*B)                                    :rem 141
20 FOR J=0 TO B:N$(J)=STR$(RND(1)*100):NEXT
                                            :rem 113
30 PRINT "SORTING:-"                        :rem 193
2000 X=0:FOR J=0 TO B:I(J)=J:NEXT            :rem 98
2005 FOR J=0 TO 2*N-3 STEP 2:B=B+1          :rem 215
2010 I(B)=I(J):IF N$(I(J+1))<N$(I(J)) THEN I(B)=I(
     J+1)                                   :rem 247
2015 NEXT                                     :rem 7
2020 X=X-1:C=I(B):IF C<0 THEN END            :rem 35
2025 PRINT N$(C)                             :rem 92
2030 I(C)=X                                  :rem 55
2035 J=2*INT(C/2):C=INT(C/2)+N:IF C>B GOTO 2020
                                             :rem 41
2040 IF I(J)<0 THEN I(C)=I(J+1):GOTO 2035 :rem 107
2045 IF I(J+1)<0 THEN I(C)=I(J):GOTO 2035 :rem 112
2050 I(C)=I(J):IF N$(I(J+1))<N$(I(J)) THEN I(C)=I(
     J+1)                                   :rem 253
2055 GOTO 2035                              :rem 207
```

Machine language sort. This ML sort is far faster than either of the two BASIC sorts above. Program 6-27 loads the program into free RAM at $C000, 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 64'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 6-27. Machine Language Sort for String Arrays

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
0 DATA 32,115,0,133,97,169,128,133,98,32,115,0,240
  ,7,9,128,133,98,32,115                    :rem 213
1 DATA 0,165,47,133,99,165,48,133,100,160,0,165,97
  ,209,99,208,7,200,165,98                   :rem 79
2 DATA 209,99,240,20,24,160,2,177,99,101,99,72,200
  ,177,99,101,100,133                        :rem 71
3 DATA 100,104,133,99,144,221,160,5,177,99,133,102
  ,200,177,99,133,101,208                     :rem 3
4 DATA 2,198,102,198,101,24,165,99,105,7,133,99,16
  5,100,105,0,133,100,165,101               :rem 192
5 DATA 208,2,198,102,198,101,208,4,165,102,240,18,
  133,105,162,0,134,103,134                  :rem 82
```

```
6 DATA 104,165,99,133,106,165,100,133,107,240,224,
  240,114,24,165,106,105                     :rem 198
7 DATA 3,133,106,165,107,105,0,133,107,230,103,208
  ,2,230,104,160,2,177,106                   :rem 17
8 DATA 153,109,0,136,16,248,160,5,177,106,153,109,
  0,136,192,2,208,246,170                    :rem 7
9 DATA 56,229,109,144,2,166,109,160,255,232,200,20
  2,208,8,165,112,197,109                    :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
100 FOR J=49152 TO 49394:READ X:POKE J,X:NEXT
                                             :rem 18
110 PRINT "USE SYS 49152:X TO SORT ARRAY X$(), FOR
     EXAMPLE:-"                              :rem 163
1000 INPUT "SIZE OF ARRAY";N                 :rem 109
1010 DIM XY$(N)                              :rem 16
1020 FOR J=1 TO N: XY$(J)=STR$(RND(1)*100): NEXT
                                             :rem 66
1030 PRINT "SORTING..."                      :rem 69
1040 SYS 49152:XY                            :rem 180
1050 FOR J=0 TO N:PRINT XY$(J):NEXT           :rem 5
```

Program 6-27 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, but 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. Note that 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 is 2, sorting begins from the third, and so on.

Provided spaces pad out the strings correctly, it's possible to resort an array in different ways. For an example, see the disk directory sorting program in Chapter 15, 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 0–25 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 often 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 1000–1050 in Program 6-27 provide a demonstration of the sort. Lines

1000 and 1010 establish array XY$, and line 1020 fills the array with random numeric characters. Line 1040 calls the ML sort routine, and 1050 prints the values to the screen. Note that you specify XY to sort array XY$—the $ is not used. If you wish to add this sorting routine to your own programs, lines 1000–1050 should not be included.

TRACE with SINGLE STEP

This version of TRACE displays the whole current BASIC line at the top of the screen. The f1 key toggles the trace on and off, f3 changes the speed of TRACE by accepting a number from 0 to 9 (fast), f5 executes a true single-step, and f7 traces as fast as possible through BASIC.

Program 6-28. Trace

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
1Ø DATA 169,76,133,132,169,19,133,133,169,192,133,
   134,96,255,Ø,254,15                         :rem 84
11 DATA Ø,252,72,138,72,152,72,173,136,2,141,148,1
   92,166,197,224,4,2Ø8                        :rem 122
12 DATA 12,228,197,24Ø,252,173,13,192,73,255,141,1
   3,192,173,13,192,24Ø                        :rem 116
13 DATA 38,224,5,2Ø8,61,228,197,24Ø,252,16Ø,Ø,14Ø,
   14,192,132,198,32,66                        :rem 117
14 DATA 241,24Ø,251,24,1Ø5,198,141,15,192,165,57,1
   64,58,2Ø5,16,192,2Ø8                        :rem 126
15 DATA 5,2Ø4,17,192,24Ø,92,173,15,192,141,18,192,
   162,128,16Ø,128,165                         :rem 74
16 DATA 197,2Ø1,3,24Ø,22,2Ø1,5,24Ø,2ØØ,173,14,192,
   2Ø8,162,2Ø8,74,2Ø2                          :rem 25Ø
17 DATA 2Ø8,236,136,2Ø8,233,238,18,192,2Ø8,228,12Ø
   ,162,Ø,181,Ø,157,76                         :rem 75
18 DATA 193,2Ø2,2Ø8,248,162,79,169,16Ø,157,Ø,4,2Ø2
   ,2Ø8,25Ø,32,1Ø2,229                         :rem 71
19 DATA 165,57,164,58,141,16,192,14Ø,17,192,133,2Ø
   ,132,21,32,2Ø7,192                          :rem 18
2Ø DATA 162,Ø,189,76,193,149,Ø,2Ø2,2Ø8,248,32,1Ø8,
   229,88,1Ø4,168,1Ø4                          :rem 31
21 DATA 17Ø,1Ø4,76,179,227,224,6,2Ø8,137,142,14,19
   2,228,197,24Ø,252,2Ø8                       :rem 18Ø
22 DATA 18Ø,32,19,166,16Ø,1,132,15,177,95,24Ø,67,3
   2,44,168,234,234,234                        :rem 122
23 DATA 2ØØ,177,95,17Ø,2ØØ,177,95,197,21,2Ø8,4,228
   ,2Ø,24Ø,2,176,44,132                        :rem 117
24 DATA 73,32,2Ø5,189,169,32,164,73,41,127,32,71,1
   71,2Ø1,34,2Ø8,6,165                         :rem 73
25 DATA 15,73,255,133,15,2ØØ,24Ø,17,177,95,2Ø8,16,
   168,177,95,17Ø,2ØØ                          :rem 3Ø
26 DATA 177,95,134,95,133,96,2Ø8,181,96,234,234,23
   4,234,234,16,215,2Ø1                        :rem 141
```

```
27 DATA 255,240,211,36,15,48,207,56,233,127,170,13
   2,73,160,255,202,240                        :rem 111
28 DATA 8,200,185,158,160,16,250,48,245,200,185,15
   8,160,48,178,32,71                          :rem 36
29 DATA 171,208,245,96                         :rem 70
100 FOR J=49152 TO 49483:READ X:POKE J,X:NEXT
                                               :rem 17
110 SYS 49152                                  :rem 150
```

Program 6-28 puts the ML for TRACE into memory starting at $C000. Load or type in a BASIC program and run it. As stated above, whenever you press f1, the trace begins; f7 traces fast, f5 single-steps, and f3 waits for a keypress from 0 to 9 before continuing. At this stage f1 will turn TRACE off, leaving BASIC running normally, but f1 is still tested for, so tracing can be resumed at any time. SYS 58260 NEWs BASIC and turns off TRACE completely; SYS 49152 reinstates it if desired. This combination of features offers maximum flexibility in examining BASIC programs.

Programs with graphics may list illegibly, with some BASIC characters appearing as graphics; and programs using the function keys, of course, may present problems. These are typical difficulties in designing TRACE routines.

TRACE works by wedging into BASIC. It performs various operations before returning to BASIC, which as far as possible is untouched. First, the key f1 is checked, and if it's pressed, a flag is reversed. If this flag is off, the program control is returned to BASIC. If the trace flag is on, f3 is checked, and, if pressed, a number key from 0 to 9 is awaited. When the number is received, it is inserted into a delay loop. Also, f5 is tested, and if the single-step flag is on, the program loops indefinitely waiting for f5. 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 indefinite loop reentered. If f7 is pressed, the delay loop is bypassed, so BASIC lines are listed as rapidly as possible. In this way, there is maximum keyboard control over the trace.

The program is not relocatable as it stands, but it isn't difficult for an experienced ML programmer to move it. If you disassemble the routine, note the routine at $C083, which lists lines. This routine saves the entire zero page (so LIST can't corrupt any locations), homes the cursor and blanks the first two lines of the screen, lists the line using a modification of LIST, and restores the zero page values and previous cursor position.

UNLIST

This system command prevents LISTing of BASIC to reduce the risk of unauthorized copying or modification. UNLISTing is successful in proportion to the difficulty of acquiring detailed knowledge of a system. No widely sold microcomputer yet has foolproof protection. Nevertheless, temporary and makeshift expedients may be better than nothing. A collection of suggestions follows. Note that disabling RUN/STOP and RUN/STOP–RESTORE is dealt with 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 RUN/STOP and RUN/STOP– RE-

STORE, 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 64. BASIC runs normally but lists as 0 SYSPEEK(44)*256+23 without any further lines. To use this routine, follow these steps:

1. Be sure that the program has no line numbered 0 or 1. Change the numbering if it does.
2. Enter line 0, with no spaces, in exactly this way: 0SYSPEEK(44)*256+23
3. Enter line 1 with exactly 21 asterisks (or any other character) and no spaces, like this: 1*********************
4. List lines 0–1 and check them.
5. Type in X=PEEK(44)*256+23. This is the starting address of the ML you will POKE in, usually 2071.
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 with PRINT PEEK(X) before you run. *All of them must be correct.*

```
POKE X, 169: POKE X+1,45: POKE X+2,133: POKE X+3,43: POKE X+4,169
POKE X+5,234: 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+17,166: POKE X+18,76
POKE X+19,174: POKE X+20,167:POKE X−4,0: POKE X−3,0: POKE X+22,0.
```

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 or she can list it.

Simple ML run. Here's another method, with an explanation of how it works. Enter a program with no line 0 or 1, and add 0SYS2063 and 1********** (ten asterisks). Next, perform the following ten POKEs:

```
POKE 2063,169: POKE 2064,26: POKE 2065,133: POKE 2066,43
POKE 2067,32: POKE 2068,89: POKE 2069,166: POKE 2070,76
POKE 2071,174: POKE 2072,167
```

In addition to the above POKEs, POKE 2059,0: POKE 2060,0 to put end-of-program bytes after line 0. This lists as 0 SYS 2063. It should run as normal. The ten ML bytes disassemble in this way:

```
$100F  LDA  #$1A
$1011  STA  $2B     ; MOVES START-OF-BASIC TO THE TRUE START AFTER ML
$1013  JSR  $A659   ; CLR SETS POINTERS
$1016  JMP  $A7AE   ; 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 RUN/STOP and RUN/STOP–RESTORE to get an effective UNLIST.

Special characters in REM statements. Since characters in the same line after REM don't affect a program's performance, there is plenty of scope for POKEing in or otherwise entering confusing characters. See the discussion of REM earlier in this section for some simple ideas.

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 already taken:

```
50000:::::S=PEEK(43)+256*PEEK(44): FOR J=1 TO 9999
50001:::::IF PEEK(S+4)>0 THEN POKE S+4,0:S=PEEK(S)+256*PEEK(S+1): NEXT
50002:::::END
```

RUN 50000 will put null bytes into the start of each line. Upon trying to LIST, you should see a set of line numbers and nothing else—but the program should work fine. Next, simply delete lines 50000–50002 and the process is complete.

The following lines 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 users of your programs haven't read this book. You can also set traps, like using :::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. All of a line that is longer than about 250 characters cannot be LISTed. LIST expects each line to be pointed to by a single-byte pointer and will loop indefinitely if the line is longer. However, some other commands, like READ, also fail to work.

To combine lines, replace the null byte at the end of each line with a colon (except the last one in the group), 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 it. 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 6-29. Combine Lines

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
0 INPUT "COMBINE LINES";L,U                        :rem 132
1 DEF FN DEEK(C)=PEEK(C) + 256*PEEK(C+1):C=FN DEEK
  (43):E=FN DEEK(45)-4                             :rem 225
2 LT=FN DEEK(C+2)                                  :rem 173
3 IF LT<L THEN C=FN DEEK(C):GOTO 2                 :rem 127
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:GOTO 6                   :rem 227
7 IF PEEK(C+2)=0 GOTO 13                           :rem 230
8 LT=FN DEEK(C+3):IF LT<=U THEN PRINT LT            :rem 71
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
10 C=C+1:POKE S,C-INT(C/256)*256:POKE S+1,C/256:S=
   C:C=C+4                                         :rem 166
```

```
11 IF PEEK(C)>Ø THEN C=C+1:GOTO 11           :rem 59
12 IF PEEK(C+2)>Ø GOTO 1Ø                     :rem 16
13 PRINTC+3:C=C+1:POKE S,C-INT(C/256)*256:POKE S+1
   ,C/256:CLR:END                             :rem 212
```

When this program is run, line numbers are printed, as is a value (see 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 will 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 current one (10000 GOTO 100).

You can make use of this to get another type of UNLIST. Type in some lines of BASIC, PRINT PEEK(2049), and write down the value, then POKE 2049,255 or some other random value. LIST will probably show garbage, but RUN should be satisfactory. Before a GOTO or GOSUB of the sort just described, you'll need to POKE 2049 with the correct value for the program, then afterward 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 6-30 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 6-30. VARPTR

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
1ØØ DATA 32,115,Ø,32,139,176,164,95,165,96,32
                                             :rem 168
11Ø DATA 145,179,32,115,Ø,32,139,176,133,73,132
                                             :rem 2
12Ø DATA 74,165,14,72,165,13,72,76,186,169 :rem 35
13Ø FOR J=83Ø TO 861:READ X:POKE J,X:NEXT  :rem 61
```

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. Below is an example that finds and prints the value of X.

```
200 N=123
210 SYS 830:N:X
220 FOR J=X to X+6: PRINT PEEK(J);: NEXT
```

These lines print the seven bytes which store X. In the same way, pointers or any string can be found, and so on. (Note that arrays move if new simple variables are defined; if you're investigating arrays, be sure not to add variables after VARPTR has found the current array position.)

This routine can't find TI, TI$, or ST, which are not stored as conventional variables. The machine language for the VARPTR routine follows this flow:

```
JSR   $0073   ; JSR CHRGET (IGNORES SEPARATING COLON)
JSR   $B08B   ; WITH JSR PTRGET FINDS THE VARIABLE
LDY   $5F
LDA   $60
JSR   $B391   ; CONVERTS POINTER BYTES TO FLOATING-POINT
JSR   $0073   ; IGNORES COLON
JSR   $B08B   ; FINDS SECOND VARIABLE
STA   $46
STY   $47
LDA   $08
PHA           ; TWO ENTRIES ON STACK NEEDED
LDA   $07     ; TO ASSIGN VALUE TO VARIABLE
PHA
JMP   $A9BA   ; EXIT THROUGH LET
```

Chapter 7

6510 Machine Language

- Introduction to 6510 ML Programming
- Description of the 6510 Chip
- 6510 ML Techniques
- Monitors for the 64
- Monitor Command Dictionary
- Assemblers for the 64

Chapter 7

6510 Machine Language

Machine language (ML) programming is often considered more difficult than programming in BASIC, but by the end of this chapter you should have a good grasp of ML techniques on the 64. This chapter assumes familiarity with hexadecimal notation (explained in Chapter 5) and that you have an ML monitor program available. Readers without a monitor may type in *Supermon* from the Appendix. Note that Chapter 10 is a complete reference guide to all 6510 commands and contains examples which can help you write your own ML programs.

This chapter contains actual examples to teach you the simple techniques, a description of the 6510 microprocessor, a list of problem-solving techniques, and discussions of monitors (notably *Supermon)* and assemblers.

Introduction to 6510 ML Programming

This section presents some short ML programs, using only the simplest instructions. Each example should be entered with a monitor. *Supermon* uses a fairly standard format, and the examples presented here use the *Supermon* syntax. At this stage, only four monitor commands will be discussed: A (Assemble) for writing ML programs using the mnemonic instruction set; D (Disassemble) to decode ML bytes so they appear as they did during assembly; M (Memory display), which displays the contents of consecutive bytes; and G (Go) which executes the program, much as RUN executes BASIC programs in memory. Additional monitor commands are discussed later in this chapter.

Most of the demonstration ML routines end with BRK. This is fine with *Supermon* and other ML monitors, because the BRK instruction returns control to the monitor program. SYS calls from BASIC usually end with RTS, because RTS returns control to BASIC. Therefore, *if you call any of these routines from BASIC, remember to replace BRK with RTS.*

These programs put characters into the screen memory, so the effect of each program is instantly visible; direct feedback like this is helpful in learning. The 64 has a movable screen memory, but these programs assume that the normal $0400 starting place applies. Color RAM starts at $D800.

Example 1. POKEing a Single Character to the Screen

Load and run *Supermon* (or your favorite monitor). The microprocessor's registers will be displayed (don't worry about them for now) on the line above a period followed by a cursor. The period is a prompt showing that the monitor is waiting for you to proceed.

Type in the ML program below exactly as shown, using either method. The two forms of the program are exact equivalents; they are just different ways of showing the same information. For example, the byte $A9 is always treated as the LDA command by the 6510, and the D, or disassemble, command simply expands $A9 into LDA whenever it finds it in the right place, similar to the way that BASIC's LIST expands one-byte tokens into keywords.

The A command lets you enter the program using mnemonics. Simply type A followed by a space, then the address where you want the program to start, and then the first instruction. After you press RETURN, the monitor will print the next free memory location for you.

```
.A  C000  LDA  #$00
.A  C002  STA  $0400
.A  C005  BRK
```

} Prints @ at the top Left Corner of the Screen (no color)

Press RETURN twice after typing BRK, to return to the period prompt. You can enter the same program by typing a colon, followed by eight hexadecimal values.

.:C000 A9 00 8D 00 04 00 —any—

This puts the designated values into the eight memory locations from $C000 to C007. Another way to do this is with the M command. Type M C000 C007 to display the contents of those addresses, then cursor over and type in the new value for each byte.

You'll find that .D C000 C005 disassembles the bytes, translating the contents of memory back into mnemonics. At left is the address where each instruction starts; to the right are the hexadecimal values which make up the instruction, and finally the mnemonic. The *Supermon* D command always prints an entire screen of disassembly; other monitors may display only the specified range of addresses.

Note that, looking at the six bytes of the program, the screen start $0400 is held with the low byte first and the high byte second—with 00 preceding 04. This feature is common to all three-byte commands of the 6510 and other 6500 series chips.

.G C000 executes this short program, then returns to *Supermon.* 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 the background color, making the @ invisible.

This is an easy program to understand, since $0400 is the first screen position, and POKEing 0 to the screen generates the @ symbol. In fact, we can read the ML like this: Load the accumulator with 0 (the *number* zero), store the byte in the accumulator in $0400, then BRK (break) to return to *Supermon.* The *accumulator* is an eight-bit location, and it can be loaded with any value $00–$FF; essentially, it is a one-byte buffer. The above example, therefore, has the same effect as POKEing $0400 with 0, using the BASIC command POKE 1024,0.

Here's another idea. If we cursor-up and alter the first line to LDA #$01, then G C000 has the effect of POKEing a 1 into the screen top, so the letter A appears. To make this change in the *Supermon* disassembly, type over the value 00 shown in the middle of the screen, to the left of the mnemonic. Other monitors may let you change the value in the mnemonic field. You can now put any character into any screen location, after a certain amount of calculation to determine the address, and with the screen POKE value from the Appendices.

From BASIC, FOR J=49152 TO 49157 PRINT PEEK(J): NEXT prints the six bytes of ML in decimal form, much like *Supermon*'s M command. ML programs can be POKEd into memory as well, and Chapter 9 includes a program which converts ML into BASIC DATA statements for that purpose.

To illustrate the fact that BASIC can POKE in and use ML programs, enter:

```
.A  C005  RTS
(press RETURN twice)
.X
```

The X command allows you to leave the monitor. Now that you are in BASIC, enter:

```
FOR J=1 to 255: POKE 49153,J: SYS 49152: NEXT
```

This 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 form. To disassemble the program in its final form, enter:

```
SYS 8
.D C000
```

and you will see this:

```
C000  LDA  #$FF
C002  STA  $0400
C005  RTS
```

This illustrates how the second byte of the six in the sequence contains $FF, or 255, the last value we POKEd in from BASIC. Note that your program is no longer the same one that you first typed in. Beginners are ordinarily discouraged from writing *self-modifying* programs (which change their own instructions as they run), because they can be confusing and difficult to debug. Until you have gained more experience, it is probably best to avoid self-modifying code.

Example 2. POKEing a Character with Its Color

We'll now POKE a character to the screen, and *also* POKE a byte into the corresponding position in color RAM. With *Supermon*, enter the following code (omit everything from the semicolon to the end of each line—these are comments to help you understand the commands):

```
.A  C000  LDA  #$00    ;LOAD ACCUMULATOR WITH 0
.A  C002  STA  $0400   ;STORE ACCUMULATOR IN SCREEN
.A  C005  STA  $D800   ;STORE ACCUMULATOR IN COLOR RAM
.A  C008  BRK          ;BREAK, BACK TO SUPERMON
```

This nine-byte program will disassemble with .D C000 C009 into exactly the same form; try this to confirm that it was entered correctly. Entering .M C000 C009 gives this (the hyphens represent bytes that don't matter):

```
.:C000  A9 00  8D  00  04  8D  00  D8
.:C008  00 --  --  --  --  --  --  --
```

.G C000 executes the program; @ appears, at the top left; it is black because 0 indicates black in the color RAM. Cursor up and replace LDA #$00 with LDA #$02. Now a red B will appear when you enter .G C000.

Example 3. Using an Index

This section introduces the X register and shows how to use it as an *index*. X is an eight-bit register like the accumulator (a one-byte buffer, of sorts), and the instruction TAX (Transfer Accumulator to the X register) simply copies the byte in A into X. The special notation:

```
$0400,X
```

refers not just to address $0400, but to address $0400 *plus the value of the byte in the*

X register. That is, the eight-bit value contained in X is added to the sixteen-bit address $0400, and the result is the address used in the command. Since X has eight bits, the range of addresses must be within $0400 to $04FF in the example, with similar figures applying to the color RAM area.

```
.A  C000  LDA  #$00     ;LOAD A WITH 0
.A  C002  TAX           ;TRANSFER A TO X
.A  C003  STA  $0400,X  ;STORE ACCUMULATOR IN SCREEN + X
.A  C006  LDA  #$00     ;LOAD A WITH 0
.A  C008  STA  $D800,X  ;STORE A IN COLOR RAM + X
.A  C008  BRK           ;BREAK
```

Now .G C000 prints @ in black, exactly like the previous program. The difference only appears on cursoring up, and altering LDA #$00 to LDA #$05, for example. Executing this prints E in black in the fifth screen position past the @ symbol. And any value in place of $00 prints a character offset from the screen start. Change BRK to RTS, type X to exit to BASIC, and enter:

FOR J=0 TO 255: POKE 49153,J: SYS 49152: NEXT

This prints all 256 characters consecutively in black, filling the top part of the screen and showing clearly how the index, X, operates. POKE 49159 with another color value, say, 2 for red, to watch the effect of the ML at $C006.

Example 4. Loops with ML

We've just used BASIC to cause a FOR-NEXT loop and we can do the same in ML. Exactly as in BASIC, we 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 6510, which has increment, decrement, and branching instructions. Type in the following ML program using your monitor:

```
.A  C000  LDX  #$00     ;LOAD X REGISTER WITH 0
.A  C002  TXA           ;TRANSFER X TO A (HAPPENS 256 TIMES IN
                        ;LOOP)
.A  C003  STA  $0400,X  ;STORE A IN SCREEN START + OFFSET X
.A  C006  LDA  #$02     ;SET COLOR RED
.A  C008  STA  $D800,X  ;STORE COLOR IN COLOR RAM + OFFSET X
.A  C00B  INX           ;INCREMENT X REGISTER
.A  C00C  BNE  $C002    ;BRANCH IF X NOT EQUAL TO 0
.A  C00E  BRK           ;BREAK WHEN X CYCLES THROUGH TO 0
```

With this ML in memory, .G C000 prints 256 characters in red in the top half of the screen; it does this far faster than the equivalent BASIC version in Example 3, taking about 1/200 second.

First, X is loaded with 0 and this is copied into A. (The TXA transfer uses one fewer bytes than LDA #$00.) Using TXA insures that the offset X corresponds to the character in A so that after the branch at $C00C, which is taken 255 times, the value in the accumulator depends on the value in the X register. This shortcut depends on the use of INX (INcrement X) to increase the value of the byte in the X register by one. Note that the accumulator (A) value stored in screen memory cycles through $00–$FF, but the A value stored in color RAM is always $02, so the color of each character stays constant. To understand this program fully, note the values in A and X at each stage of the program; X increases until it is as large as eight bits can contain, at which point it increments from $FF to $00, while A alternates between the identical, increasing, value of X and $02. (Incrementing an eight-bit register or memory location past $FF flips the value back to $00; similarly, decrementing below $00 gives you $FF.)

At the point that the X register holds a value of 0, the program stops looping back and executes the BRK instruction. This is because of the BNE (Branch if Not Equal to zero) instruction. As long as X contained a nonzero byte, the program branched back to the code at $C002. As soon as the value flips over to 0, no branch occurs and the next instruction is executed.

Note that the branch command starting at $C00C occupies only two bytes, in spite of looking as though it would take three bytes. It uses *relative* addressing, meaning that if the branch is taken, execution resumes at the address of the following command plus the byte just after the branch command. The example adds the offset value of $F4 to the address $C00E (it treats $F4 as negative, or −$0C, since $F4 + $0C = $00 in a single-byte register). Since $C00E−$0C is $C002, it all works fine. Don't worry if this arithmetic looks confusing; the monitor will calculate the right offset value for you, as soon as you enter the destination address for the branch. Note, however, that such branch commands can reach only 127 bytes forward or 128 back.

Example 5. Comparisons and Subroutines in ML

Just as SYS calls can run an ML program as a subroutine, provided the RTS command ends the ML, you can call ML subroutines from your own ML using the JSR (Jump to SubRoutine) instruction. RTS is therefore analogous to RETURN in BASIC, and JSR is similar to GOSUB. Add the following program steps to Example 4:

```
.A  C00E  RTS          ;CHANGE BRK TO RTS
.A  C00F  JSR   $C000  ;CALL LOOP IN EXAMPLE 4 AS A SUBROUTINE
.A  C012  INC   $C007  ;INCREMENT THE COLOR IN EXAMPLE 4
.A  C015  LDA   $C007  ;LOAD A WITH THE NEW COLOR
.A  C018  CMP   #$10   ;COMPARE THE NEW COLOR WITH 16
.A  C01A  BNE   $C00F  ;BRANCH IF NOT EQUAL TO 16
.A  C01C  BRK          ;BREAK WHEN COLOR = 16
```

Now, .G C00F runs Example 4, cycling through the colors until the last color (light gray) is reached. Because the subroutine is changed by this program, .G C00F behaves differently the second time. However, the point is that, like BASIC, subroutines provide a powerful means of dividing programs into manageable chunks. CMP (CoMPare) tests the byte in the accumulator with $10 (decimal 16), and if the two are equal, a special flag called the *zero flag* is set. The BNE that follows checks that flag, so if the value in the accumulator is not $10, the branch takes effect. Comparisons can be followed by other branches than BNE or BEQ (Branch if EQual to zero—if the zero flag is clear); 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, you could add a delay loop after C00F JSR $C000, using up time without performing significant processing work. Use the X and Y registers; Y is another eight-bit register in the 6510. Construct a loop within a loop, and use DEX (DEcrement X) and DEY (DEcrement Y), each followed by BNE, so that X decrements 256 times for each decrement of Y. Remember that RUN/STOP–RESTORE generally returns you to BASIC if your program doesn't work.

Description of the 6510 Chip

This section describes the 6510 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. The opcodes (machine language instructions) are introduced last because their use depends on prior knowledge of the other 6510 features. Chapter 10 has an annotated guide to all the opcodes; and the Appendices have comprehensive tables, giving concise information on the 6510 for experienced ML programmers.

Addressing Modes

The 6510 has 13 addressing modes. Most are easy to understand, but a few are more difficult. Disassembly treats a given byte in the same way every time, once it has determined the byte is an instruction; 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 numeric values. This is invaluable in dealing with RAM and ROM where the processor often is mainly concerned with arranging blocks of RAM. For instance, in the short programs above, we changed the contents of memory locations beginning at $0400.

All 6510 instructions are either one, two, or three bytes long. The following discussion examines each type.

Single-byte instructions. Single-byte instructions cannot reference either address or data, and operate only on features within the 6510 chip itself. The phrase *addressing mode* doesn't really apply since there is no address, but for consistency these instructions are described as possessing *implied* addressing (the address can be thought of as an eight-bit location in the processor itself). Instructions which shift or rotate bits in the accumulator, like ASL (Arithmetic Shift Left), are sometimes said to use *accumulator* addressing. Nevertheless, 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 (see the examples above). 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 section. That addressing mode is usually called *relative*, because the offset indicates a destination address relative to the current address.

Zero page instructions. Five of the two-byte modes use *zero page* addressing. The zero page is not a feature of the chip itself; it is the section of RAM in the 64 which is wired to addresses $0000–$00FF. However, the chip has the facility of enabling the most significant byte to be ignored (since it is a zero anyway), so that LDA $34 can be written in place of LDA $0034, for example. This saves a byte, which shortens programs and increases execution speed. For this reason, the first 256 bytes are usually in great demand in 6510 programs, and 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. Forgetting this difference is a common source of programming bugs for beginners.

Zero page indexed by X. LDA $A0,X loads into the accumulator the value in the address calculated by adding $A0 to the contents of the X register. 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). LDX,Y is an example.

Indexed indirect. An example of this type of instruction is LDA ($00,X). The parentheses indicate 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 zero plus zero equals zero.

Suppose the first four bytes in zero page are 01 80 84 02. The instruction LDA ($00) would be expected to load 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, would have the same effect as LDA $8001.

However, such pure zero page indirect addressing is not available on the 6510; you must use an index as well. Indexed indirect addressing, as the name implies, allows indexing of the indirect address. Thus, if X has the value $02, then LDA ($00,X) has the effect of loading the accumulator from the indirect address of $00 + $02, or ($02). If the bytes in locations $02 and $03 are 84 02, 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, or when you want to access a table of pointers in the zero page. The pointers to the start and end of BASIC program and variable space provide an example. This instruction is not uniform 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 ($00),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. However, this mode is post-indexed by Y; that is, first 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, suppose again that the four bytes at the very start of RAM contain 01 80 08 24. Now, LDA ($00),Y loads from $8001 + Y, so the 256 bytes from $8001 to $8100 can all be accessed, depending on Y's value.

Indirect indexing can be done only with the Y register. It's used for pure indirect addressing when Y is $00, for such purposes as following the link pointers from one BASIC line to the next; it is also used for processing blocks of data which aren't in the zero page.

The difference between indexed indirect and indirect indexed can be confusing at first. Put simply, indexed indirect—LDA ($00,X)—is often used to access a vector table (a series of indirect addresses which point to special locations). By changing the value of X, you can pick different two-byte addresses from the table, and use them in processing.

Indirect indexed—LDA ($00),Y—is a far more useful addressing mode; it lets you access any memory location from $0000 to $FFFF. Typically, you will place the desired base address in two free zero page locations, and index from there. To use a common example, suppose that you have loaded locations $FB and $FC with 00 04. Your base address is $0400, the first byte of screen memory. When the Y register contains zero, LDA ($FB),Y loads the accumulator with the contents of $0400. If Y is $01, STA ($FB),Y stores the accumulator contents at $0401, and so on.

Three-byte instructions. Three-byte instructions in the 6510 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 base address to give the actual referenced address. Thus, if X holds $50, LDA $8000,X loads the accumulator with the contents of $8050. As with zero page indexing, the maximum value cannot exceed the legitimate address range of $0000–$FFFF, 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. LDA $8000,Y is an example.

Absolute indirect. The 6510 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. Suppose once more that the first four bytes in zero page contain the values 01 80 84 02. 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 memory) exists in a block. For example, the RESET vector at $FFFC–$FFFD can be called by JMP ($FFFC). This addressing mode is not often used, partly because of a bug in the 6502 series chips. If the indirect jump address is located on a page boundary—for example, JMP ($80FF)—program flow will be transferred to an erroneous address.

The Status Register

The status register (or processor status register), denoted as SR in *Supermon*, is another eight-bit register. It contains seven individual status bits, or *flags*, all of which are automatically controlled by the 6510 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-patterns 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 $20 (32), even when all flags are clear. For example, if the register contains $32, then B (the break flag) is set and Z (the zero result flag) is set. These flags don't change unless altered by an instruction. For example, D (the decimal mode flag) typically remains off through all BASIC programs.

Table 7-1. 64 Status Register

7	6	5	4	3	2	1	0
N	V	1	B	D	I	Z	C

High Nybble				
2			1	
3			1	B
6		V	1	
7		V	1	B
A	N		1	
B	N		1	B
E	N	V	1	
F	N	V	1	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

Chapter 10 shows which flags are affected by each instruction. LDA, for instance, affects the N (negative) and Z flags, but no others. This process is automatic; it's part of LDA and happens even if you don't need it to. However, a few instructions are specifically for setting or clearing flags: CLC (CLear Carry) clears the C (carry) 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 (overflow) and N flags are tricky, while Z and I (the interrupt disable flag) 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 hexadecimal 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. This special concept of *negative* is part of twos complement arithmetic, which is discussed below.

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 below.

The 1 flag (bit 5 of SR) is unused. Since it is always set to 1, it is referred to in this book as the 1 flag.

The B, or break, flag (bit 4 of SR) is usually set only when a BRK instruction is encountered. 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 6510, discussed in greater detail later.

The D, or decimal calculation mode, flag (bit 3 of SR) changes the way the processor handles bytes in general and selects the 6510's binary coded decimal (BCD) mode of addition and subtraction, instead of the usual binary. The results resemble ordinary decimal arithmetic. 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 6510 automatically adds 6 to either nybble if a result exceeds 9. For more on BCD representation of numbers, see *Mapping the Commodore 64* and *The Second Book of Machine Language* from COMPUTE! Publications.

The I, or interrupt disable, flag (bit 2 of SR), when set with SEI (SEt Interrupt flag), prevents any IRQ interrupts from taking place—remember, this is the interrupt *disable* flag. Chapter 8 explains these interrupts, with examples, but due to their importance in handling the keyboard, they are mentioned in other places as well. The main reason for disabling interrupts is to prevent them from disturbing ML routines which won't work properly if interrupted; for example, you would not want an interrupt to occur while you were changing the interrupt vector to point to your own ML routine. CLI clears this flag.

The Z, or zero result, flag (bit 1 of SR) is set by most of the instructions which set N. To derive Z, all eight bits of a result are ORed together; if this process gives a value of zero, the Z bit is set to show a zero result. Otherwise, *when the result is nonzero, Z is zero*. 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 in addition, which denotes overflow from a column of figures to a more significant column (it is used as a borrow flag in subtraction). BCC, BCS, CLC, and SEC are other instructions involving this flag; they often follow a comparison (CMP) instruction.

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. A BRK or interrupt causes the PC value to be saved on the stack, as does a JSR. Thus, the value of the PC can be determined after BRK, which is how *Supermon* records the PC. Machine branch and jump instructions operate by loading new values into the PC, thereby 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 6510 instructions can use zero page addressing modes, which are faster and shorter than absolute addressing, this region is the most important area of RAM. A page is a section of 256 (2^8) bytes—the area that can be indexed by a single-byte reference—and the 6510 can address 256 pages.

The stack is a part-RAM, part-hardware feature of the 6510. It uses page 1 of RAM, from $100 to $1FF, and it can be difficult to understand for several reasons. First, although page 1 is used by the processor as the stack, it also doubles as normal RAM. Second, instructions like PHA (PusH Accumulator onto stack) and its opposite PLA (PuLl a byte from the stack into the Accumulator), 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 8-bit register, the stack pointer. The process is explained in Chapter 10.

Note that another complementary pair of instructions, PHP and PLP, operates on the processor status register, allowing it to be stored and examined at will. Four other instructions use the stack: JSR (Jump to SubRoutine), its converse RTS (ReTurn from Subroutine), RTI (ReTurn from Interrupt), and BRK. 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 (see Chapter 10).

NMI, RESET, and IRQ Vectors

The 6510 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 6510 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 64 is turned on, after a short delay, the RESET line of its 6510 is held low, causing the processor to look at locations $FFFC and $FFFD for the address for the standard power-up sequence. If you check these addresses with PRINT PEEK(65532)+256*PEEK(65533), you'll see that the the 64's ROM reset routine begins at location 64738 ($FCE2), as discussed in Chapter 5.

The RESTORE key uses a Non-Maskable Interrupt vector, so NMI can be programmed. RESET is valuable in program recovery, to restore programs which have crashed in otherwise infinite loops. IRQ is used by the 64 to read the keyboard, among other things. Chapters 5, 6, and 8 discuss the software side of these hardware features.

6510 Instructions and Opcodes

An opcode (operation code) is a single-byte value that instructs the microprocessor to perform a particular action. Since humans find it easier to deal with letters rather than binary digits, the opcodes are usually represented by *mnemonics,* character representations intended to make machine language relatively easy to read. All 6510 opcodes are three letters long, which makes for neat assembly and disassembly listings.

Although the mnemonics are standard, there is nothing to stop you from coming up with your own. This may in fact be helpful as a learning aid, although it would be unorthodox.

There are 56 distinct instruction types (and hence 56 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 6510'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 6510 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 clear or set, respectively.

Break. The BRK instruction causes an unconditional jump to the address in locations $FFFE–$FFFF, having first saved both 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 bytes in memory. The byte 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 6510 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. There is no instruction that directly increments or decrements the accumulator; however, you can use ADC or SBC to add or subtract a value from A.

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 pushes the current address plus two onto the stack, for use when the subroutine is finished.

Logical operations. AND, EOR (Exclusive-OR the byte in the accumulator), and ORA (OR the byte in the accumulator) perform binary logical operations on the accumulator using immediate data or a byte in a specified memory location, retaining the result in the accumulator, and setting the N and Z flags. The BIT instruction sets the Z flag just as AND would, but does not affect the contents of the accumulator; it also copies the sixth and seventh bits of the tested value into the V and N flags.

No operation. NOP does nothing but take space. It is useful for testing, because, for example, JSR instructions can be masked by inserting NOPs over the JSR and the two subsequent address bytes.

Return. RTS returns to the instruction following JSR by pulling the stored return address off the stack, and transferring program control to the next byte after the address. This has the effect of jumping to the instruction which follows the two-byte address after JSR. 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 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 0. Flags N, Z, and C are set.

Stack operations. These are PHA, PHP, PLA, and PLP and 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 of a 6510 machine language program consists of 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 6510 clock cycles; the faster the clock, the faster the ML executes. The 64's chip runs at about one million cycles per second. Table 7-2 summarizes timing in the 6510; most instructions are included in the first column, but a few exceptional instructions are listed in the other columns.

Table 7-2. 6510 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 difficult 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 screen full of information.

6510 ML Techniques

This section uses assembler-style listings in the examples. See the section on assemblers for information. The following topics are discussed:

- Two-Byte Operations
- Testing the Range of a Byte
- Loops
- Shift and Rotate Instructions
- Logical Instructions
- Twos Complement Arithmetic
- Decimal Arithmetic
- Debugging ML Programs

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. This is not as simple as incrementing, since there's no test for decrement from #$00 to #$FF. 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 overflow 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 (C) is set before subtraction (if it is left 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. Amazingly enough, the 6510 has no instructions specifically for multiplying and dividing. The example below, however, multiplies the contents of two zero page locations ($FC and $FD), leaving the result in the same two bytes. On average, about 6000 multiplications per second can be performed by this routine, which uses ROR (ROtate Right) to detect bits and to store the result in $FC (low byte) and $FD (high byte).

```
C000   CLC
C001   LDA   #$00
C003   LDX   #$08
C005   ROR
C006   ROR   $FC
C008   BCC   $C00D
C00A   CLC
```

```
C00B   ADC   $FD
C00D   DEX
C00E   BPL   $C005
C010   STA   $FD
C012   RTS
```

After exiting *Supermon,* the following BASIC line can be used to test the ML multiply routine.

```
10 INPUT X,Y: POKE 252,X: POKE 253,Y: SYS 49152: PRINT PEEK(252)+256* PEEK(253)
```

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.

```
C000   CLC
C001   LDX   #$08
C003   LDA   $FD
C005   ROL   $FC
C007   ROL
C008   BCS   $C00E
C00A   CMP   $FE
C00C   BCC   $C011
C00E   SBC   $FE
C010   SEC
C011   DEX
C012   BNE   $C005
C014   ROL   $FC
C016   STA   $FD
C018   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, SYSing to 49152, 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 0
```

Z is set if the contents of the first address equal those of the second; C is clear if the contents of the first are less than the second. Therefore, 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 the Range of a Byte

The following example tests whether the byte in the accumulator is within the range 5–9. 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.

```
      LDA  TESTBYTE
      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.

The short loop below puts the five bytes for the letters of the word *HELLO* on the screen. There are two versions:

```
      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 0, and CPX #$00 is redundant, since in effect the processor does this when it sets the Z flag of the status register. Decrements are often more efficient than increments, because you can eliminate the two-byte comparison instruction. 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, since the loop starts with the highest value of X and indexes backward to zero. You'll need to take that into account when you set up your tables of data. Using decrements also adds an extra difficulty; the LDA instruction cannot be executed with an X value of 0. This explains why the decrementing version uses LDA TABLE−1,X instead of LDA TABLE,X; if this were not done, the program would print a garbage character followed by OLLE.

Note that LDX #$04 : ... : BPL LOOP counts X down from 4 to 0, and the accumulator loads from the expected starting point. However, X values larger than $7F won't cause a branch on BPL (because bit 7 is used to indicate a negative number with the BPL and BMI instructions), 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 $C000. Different approaches are shown below:

```
       (A)                    (B)                      (C)
       LDA  #$00              LDY  #$00                LDY  #$00
       STA  $FB        LOOP   LDA  $C000,Y      LOOP   LDA  $C000,Y
       STA  $FD               STA  $D800,Y             STA  $D800,Y
       LDA  #$C0              LDA  $C100,Y             INY
       STA  $FC               STA  $D900,Y             BNE  LOOP
       LDA  #$D8              INY                      INC  LOOP+2
       STA  $FE               BNE  LOOP                INC  LOOP+5
       LDY  $00                                        LDA  LOOP+5
LOOP   LDA  ($FB),Y                                    CMP  #$DA
       STA  ($FD),Y                                    BNE  LOOP
       INY
       BNE  LOOP
       INC  $FC
       INC  $FE
       LDA  $FE
       CMP  #$DA
       BNE  LOOP
```

Loop B is the shortest and 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. In the example, the loop becomes LDA $C100,Y : STA $D900,Y the second time around, then LDA $C200,Y : STA $DA00,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 (crashing the computer in this case), one reason why beginners are often warned against using self-modifying code.

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 $C000 and $9600. In each case, these examples assume that the loop ends at a page start address like $DA00. 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. TRACE (Chapter 6) does this to allow LIST and BASIC to work together. The routines are simple enough but require 256 bytes of RAM protected from BASIC (usually, the top of BASIC is lowered). Use the following routine to save the area, where STORE is the first byte of your protected area:

```
        LDX   #$00
LOOP    LDA   $00,X      ; LOAD CONTENTS OF ZERO PAGE LOCATION
        STA   STORE,X    ; STORE IN SAFE LOCATION
        INX
        BNE   LOOP       ; FINISH ALL 256 BYTES
```

Use this routine to restore the area later:

```
        LDX   #$00
LOOP    LDA   STORE,X    ; LOAD VALUE FROM STORAGE
        STA   $00,X      ; STORE BACK IN ZERO PAGE
        INX
        BNE   LOOP       ; FINISH
```

Shift and Rotate Instructions

Shifting instructions (ASL, LSR) and rotating instructions (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 either 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 commands are valuable in calculations, because they multiply by 2. This example shows how to multiply by 40; at the start, location $FC holds a Y-coordinate from 0 to 24, and $FD holds 1.

The ML points to the Xth column of the Yth row of the screen. This can be done with a lookup table (which would be faster), but this is shorter: it calculates $400+40*Y (pointing to the beginning of the line), putting the result in ($FC), so an instruction like LDY $FB followed by LDA ($FC),Y references the correct position on the screen.

```
LDA  $FC  ;A HOLDS Y-COORDINATE
ASL       ;A HOLDS 2*Y-COORD (0-48)
ASL       ;A HOLDS 4*Y-COORD (0-96) C=0
ADC  $FC  ;A HOLDS 5*Y-COORD (0-120)
ASL       ;A HOLDS 10*Y-COORD (0-240)
ASL       ;A = 20*Y-COORD (0-480); ANY OVERFLOW IN C
ROL  $FD  ;$FC, $FD HOLDS $0200 + OVERFLOW
ASL       ;A = 40*Y-COORD (0-960); ANY OVERFLOW IN C
ROL  $FD  ;$FD, $FD HOLDS $0400 + OVERFLOW
STA  $FC  ;($FC) HOLDS $0400 + 40*Y-COORDINATE
```

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. Then, AND #$0F to mask off (erase) the leftmost bytes. ORA #$30 forces $30 into the byte to create the ASCII value of a numeral ready for printing. Recover the original byte and repeat. This way, both digits are correctly output.

An example of EOR may be helpful, too. EOR combines bits in repeatable patterns, and you can use this to generate a checksum for BASIC or ML programs, which is helpful in verifying if a program is correct. The following version prints a number 0–255 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  $BDCD  ;USING THIS ROM ROUTINE
```

All logical instructions (like the arithmetic instructions ADC and SBC) use the accumulator. EOR #$FF is the equivalent of NOT A, since all the bits in the accumulator are flipped; every 1 bit is changed to 0, and vice versa.

The BIT instruction is 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 0, and bits 6 and 7 of the result are copied into the V and N flags, respectively.

Twos Complement Arithmetic

It is possible to arrange arithmetic in eight-bit bytes to indicate negative values. Although $00–$FF ordinarily represent the positive numbers 0–255, with a change in interpretation, negatives can be used 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 (highest) bit, can be regarded as a sign bit. It takes one of two values, with 0 designating a positive number and 1 representing a negative sign; the seven lower bits in the byte indicate the actual value of the number. The N flag in the status register is wired to be consistent with this scheme; when N=1, the number is considered negative and BMI's branch is taken. If N=0, BPL is taken. These branches operate whether or not you're using signed arithmetic.

A number and its negative must add to 0. 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 is its own negative 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, the 64'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.

The V flag is also associated with this type of arithmetic, showing that an overflow took place into the sign bit. Consider addition, for example, where V is affected by ADC. Numbers of opposite signs cannot overflow; even extreme values must fall in the correct range. But if the signs are the same, overflow is possible. For example, $44 + $33 gives $77, and V is clear, but $63 + $32 gives $95, which in twos complement arithmetic is considered negative; 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 fully understand the 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 64's screen to print the score of a game. Start by putting the screen code for zero—$30(48)—into the first five screen locations. Then put the score 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  $0400,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  $0400,X  ;UPDATE SCREEN BYTE
       DEX           ;COUNT DOWN TO NEXT BYTE
       BPL  LOOP     ;BRANCH UNTIL X IS FF
```

This is fast and efficient. Change the value of X for more than or fewer than six digits. This method does not use BCD mode.

Debugging ML Programs

Listed here are many errors common in 6510 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. Your program will be simpler to debug if you build it out of distinct modules or subroutines that each perform a clearly defined task. Thus, when bugs appear, you can locate the source of trouble by testing one routine at a time.

Careless errors. Errors of oversight 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 Kernal or other subroutine which alters A, X, or Y.

Addressing mode errors. These stem from confusing the 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 value of X). Indirect jumps may also cause problems; JMP ($03FF) takes its address from $03FF and $0300, due to a bug in the microprocessor.

Calculation errors. With addition, subtraction, and so on, do not forget to use the proper flag 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. Also, be careful to keep track of the carry bit with shifts and rotates. That can be tricky, since C is easy to overwrite.

Status flag errors. The logic behind flags 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 stores the contents of KEY in LOCN, but STA sets no flags:

```
LDA  KEY
CMP  #$3A
BNE  ERROR
STA  LOCN
```

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, and the order should match. For example, PHA:TXA:PHA usually requires PLA:TAX:PLA to retrieve A and X. Stack errors frequently crash the computer by transferring program flow to an address that contains garbage. Advanced programmers often use the stack for temporary storage, but it is usually safer (and about as efficient) to use other RAM locations for that purpose. You may find that you can program for a long time without ever having to use the stack.

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 the BASIC pointers (including utilities like *Supermon),* 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 that 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.

Monitors for the 64

Monitors are programs that allow individual bytes to be inspected and programmed. The simplest and least useful allow little more than a hexadecimal display of bytes; the best allow assembly and disassembly as well as facilities to examine and alter memory freely, to run ML programs in a controlled way, and to convert types of data.

BASIC Monitors

BASIC programs which use PEEK and POKE to program in ML are slow, but they do have advantages, particularly for beginners. They use familiar INPUT commands, and can be loaded, run, stopped, and listed without difficulty. As they are BASIC, they occupy the normal BASIC space in RAM, whereas ML monitors occupy unfamiliar areas. They're easily modified; if you'd like decimal addresses with a disassembly, or nonstandard opcodes, these are easy to put in.

Machine Language Monitors

These are fast and generally better than BASIC. We'll concentrate on *Supermon*, a public domain program which is the work of several people, including Jim Butterfield. *Supermon* is listed in an Appendix with instructions on saving it to tape or disk. If you don't have a monitor, but do have some free time, type it in and use it.

CBM MON, supplied with Commodore's assembler package, and "Micromon-64" (published in *Compute!'s First Book of Commodore 64*) are both better than *Supermon*, but are about twice as long. They are typical good-quality monitors. Each is used in a fairly standard way; the alphabetic list of commands which follows describes typical commands, although there are minor variations. For example, a Save command like .S "ML PROGRAM",08,C000,C100, which saves $C000–$C0FF to disk and names it ML PROGRAM on one monitor, must be entered .S "ML PROGRAM",C000,C100,08 with some monitors.

Before discussing specific monitors, some potential difficulties are worth noting. First, new versions appear from time to time, and you may find that documentation lags behind. Second, there are potential memory problems, because BASIC strings or POKEs may overwrite the monitor, or the monitor may use part of the memory needed for ML. The notes following therefore explain the monitor's memory positions. Finally, it can be quite important that a monitor is compatible with BASIC. One reason is that a SYS call often runs ML in a different way from a monitor's G command. For example, POKEing location 1 to switch out ROMs may not work properly using G, despite being correct in ML.

Supermon is usually supplied with a BASIC loader, which loads like BASIC, and, when run, relocates *Supermon* into memory. Chapter 9 explains how this works. *Supermon* is put into the current top of BASIC, and the pointer ($37) is adjusted down so BASIC will not corrupt it. Its starting address is usually 38893. After exiting to BASIC with X, *Supermon* can be reentered by SYS 38893 or SYS 13, because 13 normally holds a zero byte which acts as a BRK command.

Supermon coexists well with BASIC. Also, it leaves the tape buffer from $033C unused, using page 2 locations for its own work. *Supermon* can be saved as an ML program, needing a forced load back into its correct area. There's no timesaving in doing this, and if you wish to use BASIC including string processing, you must remember to lower the top of BASIC—POKE 56,150: NEW is fine. So the BASIC loader is often the most convenient. Obviously, since loading it will overwrite BASIC in memory, it's better to load *Supermon* at the start of a session. Like most BASIC loaders of its type, repeated running of the BASIC loader will put a series of working versions of the program next to each other in the top of memory, lowering the pointer each time, and reducing memory available to BASIC.

Supermon won't normally load into the area starting at $C000, so it's suitable when you wish to write ML into $C000, which is probably most of the time. It also locates itself below cartridges at $8000, and is therefore easy to use when examining ROM programs.

Supermon has these commands:

A, D, F, G, H, L, M, R, S, T, and X.

(*Note:* All the monitors discussed here have an ASCII table near the end that lists the commands.)

It does not have an intelligent relocater, and its memory display command doesn't print ASCII equivalents of memory, so you cannot scroll through a program looking for keywords, for example. Disassembly clears the screen, then fills the page; the result is tidy, but tiresome to move through, particularly backward. But it's still a useful monitor.

CBM MON is supplied with Commodore's editor/assembler package in two versions; both use forced loads, one at $8000 and the other at $C000. A call to the start address enters the monitor—SYS 8*4096 or SYS 12*4096, respectively. Two versions are supplied so ML can be written into either of the major RAM areas; for example, the version starting at $8000 must be used when programming in $C000. POKE 56,128: POKE 55,0: NEW protects the $8000 version against corruption by BASIC strings.

It's easy to relocate *CBM MON.* For example, you may use these commands to move the $8000 version to $1000:

```
.T  8000  9000  1000              :STRAIGHT TRANSFER OF 4K
.N  1000  2000  9000  8000  9000  :ADJUST ADDRESSES BY ADDING 9000
.N  1E66  1E99  9000  8000  9000  :ADJUST ADDRESS TABLE BY ADDING 9000
```

CBM MON is based on *VICMON,* with a number of changes—references to locations 0 and 1 removed, screen altered from $1E00 to $0400, JMP (C002) replaced by JMP (A002). It fits 4K of memory, but the conversion to the 64 is not very satisfactory, and *CBM MON* versions vary. In particular, the commands B, Q, and W (to set Breakpoints, Quick Trace, and Walk) have been written out, perhaps because these didn't work properly on the 64. As a result, because *CBM MON* freely uses zero page locations (the tape buffer isn't touched), BASIC won't work properly with *CBM MON;* the easiest way to recover BASIC in full seems to be to follow the exit command, X, with:

```
POKE 43,1: POKE 44,8: POKE 45,A: POKE 46,B: CLR
POKE 2,25: POKE 23,24: POKE 24,0
```

where A and B are previous PEEK values of 45 and 46. These POKEs reset BASIC and string handling.

CBM MON has these commands:

A, C, D, F, G, H, I, L, M, N, R, S, T, and X

It has backward and forward scrolling with D, I, and M, which is very handy. (Spacing is sometimes erratic; the current line-links are used. The RUN/STOP key isn't implemented during scrolls, which can be irritating.) Backward disassembly is inherently unreliable; *CBM MON* gives priority to longer opcodes, so results can be different from forward disassembly, but this is unavoidable. Unlike *Supermon,* a disassembled portion of ML can be edited and immediately reassembled by typing over the mnemonic field of the disassembly. There's a possible bug here, because ASCII tables which happen not to be opcodes disassemble as ???. Typing RETURN on such a line converts ??? into #2, the smallest non-opcode, irrespective of its original value.

"Micromon" is very similar to CBM MON. It occupies 4K, typically $7000–$7FFF, and is force-loaded, typically by LOAD "MICROMON",8,1. POKE 56,122: NEW (or POKE 56,122: CLR) protects a version at $7000, for example, from BASIC; SYS 7*4096 would enter it. Like *CBM MON,* this monitor is quite easy to relocate: See the example later in this chapter.

Its major commands are identical to *CBM MON*'s, except that it has a command to calculate branch offsets (O), and its memory display, in effect, includes I. Like *CBM MON,* commands to set Breakpoints, Quick Trace, and so on aren't implemented. (It hangs on Q, for example.) However, Micromon has some valuable additional commands tagged on the end: $, #, and % convert hex, decimal, and binary numbers, " converts ASCII characters, + and − allow hex arithmetic, and the ampersand (&) totals bytes between two addresses into a two-byte checksum.

BASIC coexists successfully with Micromon, though it has a tendency to return to the monitor when the screen scrolls. Micromon uses the tape buffer, so don't try to assemble into the area around $033C.

How to Use a Monitor

Syntax. *Supermon* starts each line by printing a period, and commands are typed after this, appearing as .X, for example. Other prompts are used internally by the monitor—a colon when altering memory with .M, a comma when disassembling, and so on. These are generally handled automatically by the monitor, but sometimes it's useful to alter them manually.

Commands generally consist of a single letter followed by some pattern of bytes relevant to the command. When RETURN is pressed, the table storing valid commands is searched for the letter, and if it's found, a specific address is jumped to. At that point the parameters following the command are evaluated. In fact, experienced ML programmers can add extra commands by modifying the search loop.

As an example, .T 1800 1850 033C transfers the bytes in 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 it isn't, but the command is ignored. Some errors, like attempting to use an invalid command, do indicate there's been a mistake, usually by printing a question mark. Since the parameters to be input are accepted by absolute position, punctuation is irrelevant. Thus, .T 1800,1850,033C and .T 1800 1850 033C have the same effect. Most command inputs are accepted up to, but not beyond, a colon, so .A 033C CLC: GARBAGE is treated as .A 033C CLC and assembled correctly; this is often useful.

Screen scrolling may be erratic. Monitors with this feature generally use whatever linked lines already exist, so spacing may be unpredictable, with occasional skips of a line.

Interaction with BASIC. As we've seen, routines ending with BRK return to the monitor after .G C000 or a similar command runs them. A routine ending with RTS typically returns to BASIC. To run such a routine requires that you exit to BASIC with .X and then enter SYS 12*4096 or something similar. You may find, in fact, that some ML routines run correctly when called with SYS calls, but don't work from within a monitor; JSR $BDCD (a ROM routine which outputs a number) has this effect, because the monitor uses zero page routines used by the ROM routine.

After exit to BASIC, you may like to reset with SYS 64738. This leaves your monitor in RAM, but completely resets pointers and the low part of memory, leaving everything in order. If you use the above SYS, POKEs to lower the top of memory will have to be reentered.

What to do if ML crashes. When the computer is caught in a loop or will not respond for some unknown reason, try the following:

- Try RUN/STOP–RESTORE. If this works, enter a SYS call back to the monitor.
- If RUN/STOP–RESTORE fails (as it will with an X2 crash), the only recovery procedure is a hardware reset switch, not standard to the 64, as explained in Chapter 5. This erases ML from $0 to $102 and from $200 to $400, but leaves ML higher in RAM unaffected. If you have no reset switch, you'll have to turn the computer off and start over.

You may be able to avoid this problem more often if you fill RAM with zero bytes using the .F command, thus increasing the chance that a wrong command will end on BRK and return you safely to the monitor.

Getting started. If you're an absolute beginner, load your monitor and enter it; for example, load, then run *Supermon.* Try assembling the short demonstration programs at the start of this chapter, using the .A command, then run them with .G, and disassemble again with .D—the full syntax is explained in the following list of commands. You'll soon get the feel of it. Note that many monitors output their results using $ to indicate hex, but won't accept a $ as part of the command format.

Don't be discouraged if your first ML programs crash the computer with distressing regularity. ML instructions are very powerful, and work without the automatic error-checking that BASIC provides. Nearly every ML program contains a few bugs at first, and correcting them is a normal part of the programming process, for experts as well as beginners.

Monitor Command Dictionary

Following is a list of commands commonly available when using a machine language monitor. Not all of them are supported by *Supermon.* For more information, see the article and program "Micromon-64" in *COMPUTE!'s First Book of Commodore 64.*

A (Assemble)

The Assemble command converts 6510 mnemonics and data into the correct form, inferring the addressing mode from the command's format and storing ML bytes into memory. Labels and other features of true assemblers aren't accepted. There's often a read-back check in case RAM isn't there; try assembling at $A000 to see this. Typing RETURN with no ML instructions following the address allows you to exit the A mode. An example of the use of the A command follows:

```
.A  C000  LDA  #$00
.A  C002  STA  $0400
.A  C004  BRK
.A  C005
.
```

After you enter a line, many monitors will expand it to show the actual hex bytes which make up the instruction. For instance, after entering the second line above, you would see:

```
.A  C002  8D  00  04  STA  $0400
```

Screen editing can be used to alter addresses, opcodes, and operands already on the screen. Cursor to the appropriate place, make the changes, and press RETURN.

C (Compare Memory)

Compare Memory reports any differences between two areas of memory. Syntax is identical to T.

.C C000 C100 C800, for example, checks whether the bytes in $C000–$C100 match bytes from $C800 to $C900, and prints the addresses of nonmatching bytes.

D (Disassemble)

The Disassemble command translates the contents of memory into standard 6510 mnemonics, using $ and # to denote hex addresses and data. The format is compatible with that of the assembler. It cannot produce labeled disassemblies and lacks some other features of true assemblers.

Supermon always prints as much disassembly as will fit on the screen, whether you specify a single address or a range of addresses. The disassembly begins with the first specified address.

On other monitors, .D A500 then RETURN disassembles a single address; if your monitor allows scrolling, you'll be able to continue disassembly by moving the cursor to the top or bottom of the screen.

.D A46E A471 disassembles between the two limits, producing an output as follows:

```
A46E  C8                INY
A46F  F0  03        BEQ $A474
A471  20  C2  BD JSR    $BDC2
```

Some monitors let you edit a disassembly by typing changes in the mnemonic field. *Supermon* disassemblies can only be edited by typing over the hex bytes between the address and the mnemonic.

F (Fill Memory)

Fill Memory 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 wrongly, nothing will happen and you'll have no warning of this. $EA (NOP) is a useful space filler.

G (Go)

Go runs ML from *Supermon.* The command .G C000, for example, transfers control to a program starting at location $C000. The G execution continues until a BRK occurs (which returns to the monitor) or some other irregular event takes place. For example, RTS may return execution to BASIC or the program may contain a mistake and crash. The G command can also be entered without an address, to transfer control to the current program counter address (see R command below).

H (Hunt Memory)

This command reports all instances of a byte combination or string of characters between two addresses. For example:

```
.H E000 FFFF 00 90
```

prints all Kernal uses of $9000, and

```
.H E000 FFFF "BASIC
```

prints all Kernal uses of the word *BASIC.* H requires some care in interpretation. A Hunt for 20 E4 FF will certainly find all instances of JSR $FFE4, but a Hunt for 21 D0, for example, may yield nothing, even though the address $D021 had been used, because D000,X may have been used to address it. And while JMP $C100 can be found with .H 4C 00 C1, a branch command like BEQ $C100 cannot be located like this.

I (Interpret Memory)

Interpret Memory prints addresses followed by eight ASCII characters per row (or dots, where ASCII doesn't apply) and their hex equivalents. Some monitors include this as part of the M command.

L (Load ML)

Load ML uses this syntax for tape and disk loads, respectively:

```
.L "NAME",01
.L "NAME",08
```

Abbreviations are accepted, so .L " ",01 loads the next tape program, L "N*",08 loads the first disk program beginning with N. The program or data is loaded as a block—after loading, it is not altered in any way.

M (Memory Display)

Memory Display prints addresses followed by eight hex bytes, including, with some monitors, ASCII characters in reverse. Monitors which scroll allow examination of large areas of ROM or RAM. For example, .M A09E A0EE displays 11 lines of BASIC keywords, as they are stored in ROM. Readability is improved in lowercase mode. Nonprinting characters are displayed as reversed periods. The addresses and bytes can be altered, followed by RETURN to enter the new values.

N (Number Adjuster)
Number Adjuster is a command which adjusts absolute addresses, such as subroutine calls, within ML. It is usually used after moving ML as a block with T. "Micromon-64" supports the N command. For a detailed description, including instructions for relocating Micromon-64, see *COMPUTE!'s First Book of Commodore 64.*

P (Printer Disassembly)
Disassembly to a printer is supported by some monitors. When using a monitor that does not support this command, enter OPEN 4,4: CMD 4 from BASIC, then enter the monitor and type commands blind. The output from the monitor will be directed to the printer instead of the screen. This will also work if you wish to dump the bytes in memory with M. To recover the screen display after printing, use X to exit the monitor, then enter PRINT#4: CLOSE 4. If your printer omits the last instruction, specify an ending address a few bytes past the last byte you want to print.

R (Register Display)
R displays the contents of the program counter (PC), IRQ vector, status register, bytes in the A, X, and Y registers, and the stack pointer as they were on entry to the monitor. Typically, any of these can be changed. When .G runs the program, the modified contents are loaded into PC, IRQ, and so on, before actual running. In this way, you can change IRQ to point to your own interrupt routine; try different values of A, X or Y; or experiment with different flag settings in the status register.

S (Save ML)
Save uses the following syntax for tape and disk, respectively:

```
.S "ML",01,C000,C200
.S "ML",08,C000,C200
```

Unlike BASIC SAVEs, it's essential to specify the limits of memory to be saved. These examples save memory from $C000 to $C1FF. The final byte is not saved, due to the way the pointers in the machine execute the SAVE command; saving from $E000 to $FFFF is therefore impossible. In some monitors, the device number defaults to 8 if not explicitly included. Note that there may be no error message if a disk drive is off.

T (Transfer Memory)
Transfer moves a block of memory. The syntax is identical to C. For example, .T 0400 07E6 0401 moves a screen of bytes along one position. The end point of the new block is implicit in the three parameters. See N for relocation of programs.

V (Verify)
Most monitors have no Verify command, but BASIC's VERIFY can be used like this by exiting the monitor and using BASIC's VERIFY. Before using VERIFY from BASIC, you'll need to change the start-of-BASIC and end-of-program pointers to match the beginning and end of your ML program.

X (Exit to BASIC)

X is the command that allows for a safe return to BASIC.

Assemblers for the 64

Opcodes and operands generally improve the readability of ML over the hex bytes they represent. *Assemblers* carry this improvement much further, by allowing a fully algebraic or symbolic notation, with comments, to represent ML. Many of the examples earlier in this chapter are listed using an assembler format. Important instructions can be labeled so that at the end of a loop, you can write BNE INLOOP, and the assembler will calculate the offset using the label, INLOOP. Figure 7-1 is a typical assembly listing:

Figure 7-1. Printed Output from an Assembler

```
 Line   |         |              | [Label/ Opcode or Directive/ Operand/ Comment]
Number  | Address | Object Code  | ------------------ Source Code ------------------

  2                              ;
  4                              ; ROUTINE TO AWAIT A KEY, THEN EXECUTE CORRES-
  6                              ;PONDING CODE, USING TABLED VALUES
  8                              ;
 10                              GETIN=$FFE4                   ;TYPICAL 'EQUATES' DIRECTIVE
 12    2000                      *=$2000                       ;TYPICAL STARTING-POINT DIRECTIVE
 14    2000                      .PAGE                         ;TYPICAL TOP-OF-FORM DIRECTIVE
 16    2000                      ;
 18    2000    20  E4  FF        START    JSR  GETIN           ;STANDARD KERNEL 'GET' INTO ACC'R
 20    2003    F0  FB                     BEQ  START           ; WAIT UNTIL KEY PRESSED
 22    2005    A2  02                     LDX  #2              ;TABLE HAS THREE VALUES ONLY
 24    2007    DD  21  20        LOOP     CMP  CHRLIS,X        ;COMPARE VALUES IN TURN,
 26    200A    F0  05                     BEQ  FOUND           ; UNTIL FOUND OR NOT FOUND
 28    200C    CA                         DEX
 30    200D    10  F8                     BPL  LOOP            ;LOOP FROM X=2 TO X=0 INCLUSIVE
 32    200F    30  EF                     BMI  START           ;KEY NOT IN TABLE; GOTO START
 34    2011    8D  20  20        FOUND    STA  STORCH          ;STORE THE ASCII CHARACTER
 36    2014    8A                         TXA                  ;STANDARD JUMP ROUTINE FOLLOWS,
 38    2015    0A                         ASL  A               ;IN WHICH THE STACK HOLDS BOTH
 40    2016    AA                         TAX                  ;BYTES OF THE DESTINATION, AND
 42    2017    BD  25  20                 LDA  ADRTAB+1,X      ;RTS CAUSES THE JUMP.
 44    201A    48                         PHA                  ;      HIGH BYTE ON STACK ...
 46    201B    BD  24  20                 LDA  ADRTAB,X
 48    201E    48                         PHA                  ;      AND LOW BYTE.
 50    201F    60                         RTS                  ;JUMP TO ADDRESS NOW ON STACK
 52    2020                      STORCH   *=*+1                ;USES ASSEMBLER LOCATION POINTER
 53    2021    41  42  43        CHRLIS   .BYTE 'ABC'          ;SETS UP TABLE OF ASCII BYTES
 54    2024    29  20  6F        ADRTAB   .WORD A-1, B-1, C-1  ;SETS UP TABLE OF ADDRESSES - 1
 56    202A    A0  00            A        LDY  #0              ;START OF PROCESSING FOR ROUTINE A
```

These instructions and labels together make up the *source code.* Around the core of familiar 6510 opcodes is a collection of symbols, some of which are punctuated to resemble addressing modes. This code is stored with line numbers, in RAM or on disk or tape as the *source file.* Source code may include *equates,* like SCREEN= $0400, and may have a comment after each instruction to document the program. Therefore, source code is usually much longer than object code, often 20 times as long.

The job of the assembler is to convert the source code into *object code*—the actual numbers which make up machine language instructions. Note that the object code is a sequence of bytes identical to that produced by a simple disassembler. This is necessary, of course, since the 6510 has precise requirements which any utility program must respect. Object code is often stored on disk as an *object file;* this is a machine language program and can be loaded into a specific place in memory and run with a SYS call.

The versatility of assemblers is illustrated by the *pseudo-opcode* (a special assembler function) which assigns the starting address of the ML program. The starting address pseudo-op is always used at the beginning of the source code. The command *=$2000 at the start of the source code causes the assembler to create the ML program starting at $2000. Simply changing the command to *=$3000, followed by assembly, generates ML identical in its effect, but positioned to start at $3000. Object code, on the other hand, isn't usually relocatable without some effort.

But the great advantage of source code is the fact that it can be edited. Inserting extra instructions in the middle of a program is easy, because assembly simply recalculates all the addresses and branches. In contrast, monitor users have to shift parts of the program, alter addresses, and generally rewrite and recheck.

Assemblers also have the advantage of potentially giving ML a very readable format, provided the reader has a good grasp of ML. Symbols like GETCHR and labels like FOUND make ML easier to follow than the object code; and comments allow the programmer room for thorough explanation of the program.

Figure 7-1 is part of a routine which waits for A, B, or C to be pressed, then jumps to a corresponding address, using the trick of pushing the destination address less 1 onto the stack, then using RTS to jump. The column of comments helps in deciphering the program. Object code is obviously harder to follow than source code, but there are *reverse assemblers* available, which take ML and insert labels. Of course, it is impossible to reconstruct comments or the original labels.

Overview of the Assembly Process

A number of assembler packages are available for the 64, many of which have been converted from PET/CBM programs. Converting source into object code is complex, and full-featured assemblers are naturally longer than those which are more restricted. But they all have some common characteristics. All use two passes or more (an assembler must look through the code at least twice), and all build up a symbol table, which is used on the second pass to fill in the forward addresses. To see why this is necessary, imagine you are assembling the code in Figure 7-1, and have arrived at line 24 for the first time. CHRLIS hasn't yet been reached, so its value can be put into the object code and the symbol table only on the second pass. If the label CHRLIS is never found—it may have been misspelled, for example—the assembler will print an error message.

All assemblers have to build symbol tables, which means part of RAM must be allocated. The assembler itself and the file on which it's working have to coexist in memory. Since the source code is usually much longer than the output object code, programs amounting to only a few K when assembled can be difficult to fit in the 64's memory. For this reason, many assemblers allow one source file to chain or link to another, so a program is assembled in sections. However, the symbol table is usually kept intact. Symbols are often limited in length to conserve space in the symbol table.

If the assembler is directed to send the assembled ML to disk, clearly no RAM is used. Many assembler packages include a monitor to allow disassembly, running under ML control, and other convenient functions. An assembler may also include a *text editor* to facilitate the task of writing source code. Special loader and relocater programs are helpful, too. One key to successful use of assembler packages, therefore, is learning to manage the 64's memory efficiently.

Assembler Features

An assembler either reads a source file into RAM or operates on source text already in RAM, converting it into object code on a command such as A, ASSEMBLE, or OPT OO. Assemblers vary in the way they scan source code; some require precise alignment in columns and signal errors if they don't find them, while others are more tolerant. Line 18, containing START, may be rejected, because it seems to contain the opcode STA. Since standardization is limited, it makes sense to learn to use just one assembler.

First-time assemblies without errors are rare. Unlike BASIC, which can run with SYNTAX ERRORs remaining in the unused portions of the code, assembling is not tolerant of errors. Often, removing errors becomes a goal in itself. The triumph of achieving a no-errors message may cause the programmer to fail to notice that the program doesn't do what it should. Because repeated assemblies are the norm, it's desirable that assemblers and their source files should coexist in RAM; it saves disk access time. Likewise, you can speed up the process of testing and revision by assembling directly to RAM when possible, using the disk only to back up your work and save the final product.

Assemblers for the 6510 typically allow for these features:

Labels. These mark addresses to which branches, jumps, or subroutine calls are made. Often there is a specified maximum length.

Symbols. These are values like GETIN in the example which are explicitly set. The assembler must be able to distinguish zero page symbols from others. In practice, the terms *label* and *symbol* are often used synonymously.

Opcodes. Standard 6510 opcodes, like LDA.

Operands. These are symbols or arithmetic values punctuated in a standard way. Line 22's LDX #2 could be written LDX #$2 (hex) or LDX #%00000010 (binary). Line 42 has a symbol used in indirect addressing, but also shows the use of simple arithmetic; many assemblers allow evaluations like this. Line 53 shows the use of the quote to generate ASCII. LDA "A is equivalent to LDA #$41 on some assemblers and is often more convenient. The constructions LDA #<ADDRESS and LDA #>ADDRESS, loading the low and high bytes of ADDRESS, respectively, are often used.

Comments. Generally these are signaled by a semicolon, which causes the assembler to ignore the rest of the line.

Pseudo-opcodes (directives). These are important and, like symbols, essential to assemblers. Formats vary, so what follows may not apply to your assembler.

Pseudo-ops are commands to the assembler, some of which have housekeeping functions, like diverting output to a printer rather than to the screen. Others ease programming, allowing, for example, easy entry of ASCII bytes. They are called pseudo-opcodes because they appear in the source code in the same place as opcodes; often they begin with a period or some other special symbol, so the assembler's parser looks either for an opcode or a period or some other character on each line. These are typical pseudo-ops:

*= (sometimes .ORG, meaning origin) sets the current address at which object code should start. Line 12 in the example starts the assembly at $2000. Line 52 reserves one byte, by adding 1 to the current address. Similarly *=*+500 reserves 500 bytes, and LDA #*−LABEL loads A with the difference between the then-current address and an earlier label. The term *star* is often used to denote this symbol.

= (sometimes .EQU, meaning equates) assigns values to symbols. Equates are usually collected at the start of source code, where they can be easily checked. Usually, zero page equates must be at the start of the source code.

.BYTE allows bytes to be assembled; this is necessary for any kind of data table. So .BYT 31,$EA,%00010001,"HELLO puts eight bytes, 1F EA 11 48 45 4C 4C 4F, into the object code.

.DISK NAME outputs object code to a disk file called NAME.

.END marks the end of the source code. Anything later is treated as comment.

.FILE NEXT instructs the assembler to load, then assemble the source file called NEXT. This pseudo-op is essential for chaining the components of large ML programs.

.LABELS causes a symbol table (a sorted table of all symbols with their values) to be printed after the assembly.

.MACRO INC causes a macro to be inserted; see below.

.OUTPUT instructs the assembler to put the object code into RAM as it is assembled so that it will be ready to run.

.PRINTER diverts whatever output is requested to a printer.

.SCREEN turns on output to screen. With .NS (for example) a part of the assembly can be selected.

.WORD puts two bytes into the object code, least significant byte first, to conform with 6510 convention. Line 54 has an example.

Conditional assembly, library files, macros, and relocatable object files. These are typical extras of good assemblers. Conditional assembly allows several versions of the object code to be prepared. A simple example follows:

```
.IF TYPE-1 <
*=$8000
SCREEN=$0400 >
.IF TYPE-2 <
*=$C000
SCREEN=$8000 >
```

Depending on how the expression after IF evaluates, the source code is assembled at different locations.

.LIB NAME loads and assembles a file, inserting it into the current assembly. This is not the same as chaining, but permits a source file to be built from a group of separate library source files.

Macros allow space to be saved and readability improved by defining pseudo-ops which correspond to ML. For example:

```
.MAC DBINC
INC ?1
BNE XXX
INC ?1+1
XXX .MND
```

defines a macro called DBINC, which performs a double-byte increment, on Commodore's assembler. So DBINC POINTR in the source code will cause the assembler to expand DBINC into three operations.

As mentioned earlier, object files need not exist in immediately loadable form. They can be stored in an intermediate state, without symbols or comments of any kind, but with sufficient extra information on the file to allow relocation in RAM by a loader. The process is analogous to the N command available on monitors. Addresses requiring relocation must be marked in the object file, then the loader simply has to calculate the actual addresses depending on where in RAM it's asked to put the final ML.

Assembler Packages

Here's a brief look at the different types of assemblers that are available for the 64.

Assemblers written in BASIC. Everything described so far can be carried out by BASIC. For example, symbols can be stored in a string array, object files can be written straight to disk, one byte at a time, and pseudo-ops like * can be implemented. The problem with using BASIC is that such assemblers are painfully slow, and take up more RAM than those written in ML.

Assemblers with BASIC editing. Some cartridge-based assemblers use the BASIC editor. When you turn the computer on with the cartridge in place, routines to intercept BASIC are set up, and commands like &A, &S, and &L are used to assemble, save, and load source files. It may not be possible to write the resulting object code to disk, but a monitor with the S (Save) command may be included to save the RAM image as a normal ML program. The instant availability of the assembler is nice, and some edit features may be present, in fact, like AUTO line numbering. A programmer's aid utility may be compatible with the assembler, and this can help edit, though clearly it couldn't be expected to automatically format its output the way the assembler would like.

Another example of an assembler that uses the BASIC editor is Richard Mansfield's *LADS* assembler (published in *The Second Book of Machine Language,* from COMPUTE! Books). *LADS* loads into memory beginning at $2AF8 (11000). Its symbol table is stored down from the start of its own code, and the BASIC source code is stored, loaded, and saved in the usual BASIC area from $0800. The area after $C000 is free, and this could hold a monitor or RAM object code. The assembler recognizes and expands tokenized BASIC keywords if these occur in the source file. The syntax requirements of the assembler are relatively strict—LOOP LDA #0 needs exactly one space between each item. *LADS* produces ML in nonrelocatable form which can either remain in RAM or be saved to disk or tape as loadable ML. Of course, the source code allows for locating the program at different places in memory, and the text describes how to modify *LADS,* customizing it to your needs.

Editor/Assemblers. Commodore's ASM6440 program package has eight separate programs on disk, but some are simply relocated versions—*CBM MON* can load at $8000 or $C000, for example. The assembler loads into the same area of RAM that the editor stores source code, and the package entails a lot of disk activity because of this. Source code must be saved to disk, then the assembler loaded and run, and errors in the source code can be corrected only by reloading the source file, editing it, saving to disk, reloading and running the assembler, and so on.

The editor is force-loaded into $C000–$C64A. SYS 49152 activates it. Tokenization is disabled. PUT and CPUT (which writes to disk omitting spare spaces) write the source to disk as a sequential file, and GET reads it back. AUTO and NUMBER handle line numbers, and FORMAT is similar to LIST, but formats the source into columns. KILL disables the editor, returning to normal BASIC, while DELETE allows block deletion. FIND and CHANGE allow symbols or other source code features to be altered and allow any delimiter to be used. For example, FIND /GETIN/ searches for GETIN, and CHANGE /LOOP/EXIT/ replaces the characters *LOOP* with the characters *EXIT.* CHANGE ALOOPAEXITA has the same effect.

The assembler loads like BASIC and includes a SYS call, so RUN is all that's needed to initiate assembly. Its output is an object file on disk, written in a relocatable format. This must be loaded into RAM with one of the loaders supplied, then saved with the monitor; this process is more cumbersome than using an assembler that puts pure object code directly into RAM.

Two loaders are supplied; LOLOADER fits the BASIC space and can be run. HILOADER is force-loaded into $C800 and run with SYS 51200. These loaders include a checksum test and read-back check to insure that the ML goes to RAM.

Yet another program included is a DOS wedge, located at $CC00 (52224), which simplifies program LOADs from disk.

Part of the reason for this patchwork of programs is its origin as a PET assembler; several programs have been left so they overwrite each other.

Another assembler is the *MAE Editor/Assembler* by Carl Moser of Eastern House Software. Also based on PET, the default memory management is well designed. The editor, assembler, monitor, assembled ML at $C000, and BASIC all coexist in RAM, allowing writing and testing of ML programs to be very efficient. Other programs include a screen scroller, a disk wedge, and relocaters, which operate on relocatable object files if these are chosen. This assembler supports all the previously explained features.

Chapter 8

ML Methods Specific to the 64

- Kernal Routines
- BASIC ROM Routines
- Using RAM Under ROM
- Modifying BASIC
- Vectors
- Interrupts

ML Methods Specific to the 64

This chapter is a reference to the ROM of the 64, and a guide to using the vectors that point to that ROM effectively. You can include your own ML routines that wedge into the normal operations of the computer if you like, and this chapter will show you how. Other chapters that include specific ML material are Chapter 6 (keyboard, screen, etc.), Chapter 12 (graphics), and Chapters 13–17 (sound, tape, disks, peripherals).

Kernal Routines

The Kernal is the essential core of ML routines that the 64 uses during normal operation, and it has a *jump table* pointing to the routines. While the specific addresses of the routines may differ from one computer to the next—like in the VIC and 64—the addresses of the jump table are supposed to remain constant between machines. In theory, this will allow programs to work on several ROM versions of the same machine and even on different Commodore computer models. In practice, consistency among different models is achievable only to a small extent, because so many hardware and software differences exist between machines. Still, it does serve a useful purpose.

Commodore has upgraded the 64's Kernal ROM in the past, and more changes are possible (see "Upgrading ROMs" below). ML programs that access Kernal routines only through the jump table are likely to work correctly on machines with updated ROMs; programs that jump into ROM routines at other entry points might work differently (or not at all) after a ROM upgrade.

The jump table listed below is arranged in ascending order by memory location. The Kernal appears less formidable if you note that more than half is concerned with opening and closing files and input/output of characters. Table 8-1 lists input/output errors that may be returned by Kernal routines.

Note that values shown in parentheses in "The Kernal Jump Table" (below) *are two-byte vectors which contain addresses in standard 6510 low/high byte form. The value ($0281) can be read as "the vector at $0281."*

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. To see the Kernal error messages at work, run this program:

```
10 POKE 157,64:PRINT#55,X$
```

The Kernal message *I/O ERROR #3* will be printed, as will the BASIC message *?FILE NOT OPEN ERROR IN 10.* Location 157 controls the printing of error messages; in normal BASIC operation it contains 0, which suppresses Kernal messages.

The Kernal Jump Table

Kernal Address	Routine Location	Name	Description
FF81	**FF5B**	**PCINT**	**Initialize Screen and Keyboard** Sets VIC chip; sets keyboard buffer to 10; sets light blue foreground; clears screen; homes cursor. CIA timer 1 (registers $D804–$D805) set to generate 60 Hz interrupts both with PAL and NTSC TVs.
FF84	**FDA3**	**IOINIT**	**Initialize I/O Devices** Sets CIA 1 for keyboard scan, CIA 2 for serial device input/output, and the port at address 1 for standard BASIC memory map and for tape handling. Turns off SID chip volume but does not clear other SID registers.
FF87	**FD50**	**RAMTAS**	**Set and Check RAM** Clears RAM up to $03FF, excluding the stack. Sets tape buffer to start at $033C, sets ($0281) to start of BASIC RAM, $0800, and ($0283) to end of BASIC RAM (varies if plug-in cartridge is present). Sets screen to start at $0400.
FF8A	**FD15**	**RESTOR**	**Set Default Vectors** Sets 16 vectors in ($0314–$0333) from a ROM table; used on power-up and reset. Alters A, X, Y, and SR. No error returns.
FF8D	**FD1A**	**VECTOR**	**Save/Set User Vectors** If C flag is set: Moves table from $0314–$0334 to X (low), Y (high) address, saving current vectors. If C is clear: Moves table from X (low), Y (high) back to $0314–$0334. Alters A, Y, and SR. No error returns.
FF90	**FE18**	**SETMSG**	**Control Screen Messages** Puts A into $9D to control messages. A has bit 7 set for direct mode, off for program mode. Bit 6 (not used by 64) causes I/O errors to appear, as the table above shows. Alters A and SP. No error returns.
FF93	**EDB9**	**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 that data output from the 64 can begin (see Chapter 17). Alters A, SR, and probably X,Y. Errors returned in ST byte at $90.
FF96	**EDC7**	**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	**FE25**	**MEMTOP**	**BASIC RAM Top** If C flag set: Loads X (low), Y (high) from ($0283). If 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 64 when power is applied. Alters X, Y, and SR. No error returns.
FF9C	**FE34**	**MEMBOT**	**BASIC RAM Bottom** Identical to MEMTOP, except that ($0281) is the relevant address.
FF9F	**EA87**	**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	**FE21**	**SETTMO**	**Set Time-out** Not used by the 64. Stores A into $0285, but this location is never used. Intention is to set a time-out value, after which a serial device is assumed not present.
FFA5	**EE13**	**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	**EDDD**	**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	**EDEF**	**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	**EDFE**	**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	**ED0C**	**LISTEN**	**Make Device Listen** Converts a device on the serial bus to a listener. Register A holds the device number (4–30). ATN is held low to send the command byte, which is the device number ORA #$20. Alters A, X, SR, and probably Y. Errors return in status byte $90.

FFB4	ED09	TALK	**Make Device Talk** 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	FE07	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 this routine clears $0297 to zero after reading it.
FFBA	FE00	SETLFS	**Set Logical (File Number), First (Device), Secondary Address** This and the following routine are preliminaries to opening a file. They are, in effect, used by all OPEN statements. There are three routines because the 6510 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	FDF9	SETNAM	**Set Filename** A is the length of the filename; X (low), Y (high) points to the start of the name. If A is 0 (acceptable for tape), X and Y become irrelevant. No error returns.
FFC0	(031A) Usually F34A	OPEN	**Open a File** Opens a file, assuming that the filename and other parameters have been set by using SETNAM and SETLFS. 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 F291	CLOSE	**Close a 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 F20E	CHKIN	**Prepare Open File for Input** Prepares an open channel to receive input, in the way GET# is used. Load X with the file number, call CHKIN with JSR $FFC6. Then you can use GETIN, 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 F250	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, then close files with CLOSE, for example. An error is indicated if C is set on return from CHKOUT; A holds the error number (3, 5, or 7). Alters A, X, Y, and SR.

FFCC	(0322) Usually F333	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 F157	CHRIN	**Input a Character** Gets a single byte from the current input device (indicated 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 F1CA	CHROUT	**Output a Character** Outputs a single character to the current output devices. 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. Register A retains its entry value, X and Y are unaltered. Errors return in status byte $90.
FFD5	F49E	LOAD	**Load to RAM** Kernal LOAD is used by BASIC LOAD to load from tape or disk into RAM. The result is not relinked as it is with BASIC LOAD. Thus, this routine 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 used 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. Provided that the secondary address is 0, X (low) and Y (high) point to the address at which LOAD will start. If it is nonzero, the program will be loaded at the start address stored with the file to be loaded.
	(0330) Usually F4A5		LOAD uses a vector after X and Y are stored. The routine branches to $F4B8 (disk LOAD) or $F533 (tape LOAD). On exit, C set denotes an error. Register A holds the error number (4, 5, 8, or 9); A, X, Y, and SR are all altered by LOAD. X (low) and Y (high) point to the end address plus one following LOAD.

FFD8	F5DD (0332) Usually F5ED	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) and Y (high) define the end address (one byte past the end of the block to be saved). Register A is used as a pointer to a zero page vector that contains the start address. If A holds $2A, for instance, the contents of $2A (low) and $2B (high) define the start address. SAVE uses a vector after the addresses are stored in ($C1) and ($AE). After this, the routine branches to $F5FA (disk) or $F659 (tape). On exit, C set denotes an error. In this case, A holds 5, 8, or 9. However, with disks there may be a disk error which has to be read from the disk drive error channel. Alters A, X, Y, and SR.
FFDB	F6E4	SETTIM	**Set Jiffy Clock** Stores Y (highest), X (high), A (low) into three RAM locations which store the jiffy clock. If TI$ is greater than 240000, the next interrupt resets to a normal time range. This is not the most useful routine since the CIA clocks are generally more reliable.
FFDE	F6DD	RDTIM	**Read Jiffy Clock** The converse of the previous routine, RDTIM loads Y (highest), X (high), and A (low) from the TI clock's bytes. The result usually needs some conversion to be useful.
FFE1	(0328) Usually F6ED	STOP	**Test RUN/STOP Key** This is an easy way to check if RUN/STOP is pressed, so RUN/STOP can be used to break into ML programs as an exit mechanism. JSR $FFE1:BEQ will branch if the RUN/STOP key is pressed. Note that seven other keys—Q, Commodore key, space, 2, CTRL, left-arrow, and 1 return unique values in A (for example, $EF is space). If none of these eight keys is pressed, A returns with $FF. If RUN/STOP is pressed, CLRCHN is called. If you don't want this, use LDA $91:CMP #$7F, which will test for the RUN/STOP key. Alters A and SR and, if CLRCHN is called, Y. No errors returned.
FFE4	(032A) Usually F13E	GETIN	**Get a 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 GETIN; otherwise, errors returned in status byte $90.

FFE7	**(032C) Usually F32F**	**CLALL**	**Abort All I/O** Sets number of open files to 0 and unlistens and untalks all devices, but 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. In simple terms, CLALL makes all files appear closed to the computer, but the disk drive may still treat a file as open and create an unclosed file on the disk (see Chapter 15). It is always best to close each file individually with CLOSE (FFC3).
FFEA	**F69B**	**UDTIM**	**Update Timer, Read RUN/STOP Key** Increments TI clock; if the result is 24 hours, returns to 0. (To keep correct time, increments must be made regularly by interrupt.) Location $91 is updated to hold the RUN/STOP key register, so the Kernal STOP routine can be used after this. Alters A, X, and SR. No error returns.
FFED	**E505**	**SCREEN**	**Check Screen Format** SCREEN returns 22 in X and 23 in Y for VIC, and 40 in X and 25 in Y for the 64, regardless of the true screen dimensions (which can change). It can be used in programs that work on the VIC and 64 to determine which computer is in use. For an example, see "*SpeedScript* Customizer" in *COMPUTE!'s Commodore Collection, Volume* 2. Alters X, Y, and SR. No errors.
FFF0	**E50A**	**PLOT**	**Cursor Position** If C flag set: Reads $D6 into X and $D3 into Y, cursor positions down (0–24) and across (0–39), respectively. If C flag clear: Sets X (down), Y (across). An example of its use is given below. PLOT adjusts the screen links. Alters A, X, and SR. No errors.
FFF3	**E500**	**BASE**	**Return $DC00** Loads X (low), Y (high) with $DC00, the start of CIA 1.
(FFFA)	**FE43**	**NMI**	**Non-Maskable Interrupt**
(FFFC)	**FCE2**	**RESET**	**Reset**
(FFFE)	**FF48**	**IRQ**	**Interrupt Request and Break**

Using the Kernal

Using CHROUT to print to screen. CHROUT ($FFD2) prints to screen somewhat like PRINT does, using the same characters. This makes it an easy command to use, and it produces a notable increase over BASIC's speed. Typically, a table of characters beginning with 147 ($93, which is {CLR}) and ending with 0 (to mark the end) is set up, including color and cursor characters; an ML loop prints these far faster than BASIC. Chapter 12 includes several graphics routines using CHROUT.

Try the following:

```
     033C LDX #$00
LOOP 033E LDA TABLE,X  ;TABLE COULD START AT $034A
     0341 BEQ EXIT
     0343 JSR  $FFD2   ;CHROUT DOESN'T AFFECT X
     0346 INX
     0347 BNE LOOP
EXIT 0349 RTS (or BRK)
```

Using PLOT to position cursor. The following example is typical. It positions the cursor with PLOT, and then prints the letter A with CHROUT.

```
CLC         ; SET CURSOR. EXAMPLE VALUES:
LDX  #$09   ; TENTH LINE DOWN
LDY  #$03   ; FOURTH COLUMN ACROSS
JSR  $FFF0  ; PLOT SETS PARAMETERS
LDA  #$41   ; ASCII LETTER A
JSR  $FFD2  ; CHROUT PRINTS THE CHARACTER
RTS or BRK
```

Using GETIN to fetch keyboard characters. The short example below shows a method for echoing keypresses to the screen. In practice, more constructive uses are likely. Note the loop branching back to JSR $FFE4; this is similar to the GET loop waiting for a character. Note also the test for the * key, which allows an exit from the loop.

```
LOOP JSR  $FFE4
     BEQ  LOOP   ;AWAIT KEY
     CMP  #$2A   ;A HOLDS BYTE. COMPARE WITH *
     BEQ  EXIT   ;EXIT ON *
     JSR  $FFD2  ;CHROUT PRINTS TO SCREEN
     BNE  LOOP   ;BRANCHES ALWAYS
EXIT RTS or BRK
```

GETIN alters X and Y registers, unlike CHROUT. Thus, while you can use X or Y in a loop with CHROUT alone, you must use a temporary storage location as the counter when using GETIN and CHROUT together.

Using CHRIN to fetch characters. The routine below shows how a loop inputs successive characters using CHRIN. 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 program prints the characters at the top of the screen to show how CHRIN works. Note how ANDing the accumulator contents with $3F converts the ASCII value into the correct screen display code. $FE is used as a temporary store for the current offset, since X or Y can be 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 BEFOREHAND
     LDA  #$00
     STA  $FE       ;COUNTER
LOOP JSR  $FFCF     ;CHRIN
     CMP  #$0D      ;RETURN IS LAST CHARACTER
     BEQ  EXIT
     LDX  $FE
     INC  $FE       ;BUMP COUNTER UP
     AND  #$3F      ;CONVERT ASCII TO POKE VALUE
     STA  $0400,X   ;STORE CHARACTER TO SCREEN
     LDA  #$00
     STA  $D800,X   ;SET COLOR RAM
     BEQ  LOOP
EXIT RTS or BRK
```

Using LOAD and SAVE. Examples are in Chapter 6 (BLOCK LOAD and SAVE) and in the chapters on disk and tape. If the precise mechanism of these commands interests you, disassemble the routines, following the branches to tape or disk. Tape LOAD at $F533 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. This simple routine saves you the trouble 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 to some routine other than $EA31, its usual destination. The new routine sets the interrupt disable flag (SEI), so no further interrupts are allowed, and repeatedly reads the keyboard until some predetermined keypress occurs. At that time, interrupts are enabled (CLI), then JMP $EA31 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 RUN/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 $00FF–$0104, as the demo shows by storing it to the screen top. The six bytes can, of course, be edited and printed (for example) as 07:15:00.

```
     JSR  $AF84    ;READ/SET CLOCK
     STY  $5E
     DEY
     STY  $71
     LDY  #$06
     STY  $5D
     LDY  #$24
     JSR  $BE68    ; NOW TI$ IS SET UP IN 00FF-0104
     LDX  #$05     ; POKE SIX BYTES INTO SCREEN
LOOP LDA  $00FF,X  ; NOT $FF,X
     STA  $0400,X  ; STORE TO SCREEN
     LDA  #$00
     STA  $D800,X  ; COLOR RAM
     DEX
     BPL  LOOP
     RTS or BRK
```

BASIC ROM Routines

BASIC obviously 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 a BASIC program is in memory, JMP $A871 (or SYS 43121) will run the program, provided it has a line 0, or generate an *?UNDEF'D STATEMENT ERROR* if line 0 does not exist. Any line of BASIC can be run from ML with this equivalent of RUN:

```
JSR  $A660   ;CLR
LDA  #$LO    ;LOW BYTE OF LINE NUMBER
STA  $14
LDA  #$HI    ;HIGH BYTE OF LINE NUMBER
STA  $15
JSR  $A8A3   ;FIND LINE
JMP  $A7AE   ;GOTO LINE
```

This can be useful when ML calls BASIC; see UNLIST in Chapter 6 for an example. Remember that 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 $A560 prints a flashing cursor, then transfers the screen line into the 88-byte input 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, the 64 tokenizes the buffer and treats it as BASIC. To get the feel of this, load and output characters from $200 onward with CHROUT.

Processing BASIC Variables

VARPTR (Chapter 6) uses JSR $B08B to input a variable name and search for it in BASIC RAM. The address returns in Y and A.

Printing strings and numerals. A cluster of routines around $AB1E outputs strings without the need to repeatedly call $FFD2 to print individual characters. For example, JSR $BDDD, then JSR $AB1E prints the contents of FAC1. JSR $BDDD converts the accumulator to an ASCII string, setting pointers ready for JSR $AB1E to print.

$AB1E is generally useful and will print any ASCII string up to a zero byte (or double quotation 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 $B79B and JSR $AD8A fetch the numbers. There's generally a choice of registers and memory locations for use in transferring data between ROM routines. $B79B returns the value in both $65 and X; $AD8A evaluates numeric expressions (for instance, VAL(X$)+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 how to carry out calculations in ML. With the help of Chapter 11, it will be clear that useful results are relatively easy to achieve, so you should not be held back by problems requiring 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 bytes of data (mantissa), and a sign. If E is set to 0, the number is treated as 0 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 $BBC7. 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, $BD7E adds the contents of A to FAC1, and $BAE2 multiplies FAC1 by 10. Between them, these routines allow ordinary decimal numbers to be input and stored in FAC1 as each digit is entered.

The ROM routine at $B391 is an easy way to put integers from −32768 to +32767 into FAC1 as floating-point numbers. The following routine loads 1 into FAC1; A holds the high byte and Y holds the low byte of the number, in 16-bit signed integer format.

```
LDY  #1
LDA  #0
JSR  $B391
```

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 which is always followed by a value in parentheses, like PEEK. USR lets you pass a value from BASIC to ML by enclosing the value in parentheses after the keyword. You can pass a value from ML back to BASIC by assigning a variable to the function.

For example, consider the statement A = USR(6). When BASIC finds this, the value in parentheses is computed, and the value is put into FAC1. Then BASIC executes JMP $0310. Locations $0310–0312 (784–786) act as a user-defined jump vector, just like the Kernal jump tables at the top of memory. $0310 contains a JMP instruction, and you are responsible for loading the next two bytes with a destination address. If this vector contains 0310 JMP $C000, for example, program flow is transferred to the routine at $C000, where you may process the value in FAC1. When the ML routine ends with RTS, the value then contained in FAC1 is assigned to the BASIC variable A.

Thus, POKE 784,96 puts RTS at $0310, 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 in the legal range and five bytes will be output in MFLPT format.

Program 8-1. USR Demonstration

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 FOR J=828 TO 841:READ X:POKE J,X:NEXT    :rem 15
20 POKE 784,76:POKE 785,60:POKE 786,3      :rem 217
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 32,199,187,162,4,181,92            :rem 45
110 DATA 157,74,3,202,16,248,96            :rem 247
120 PRINT:GOTO 30                          :rem 246
```

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. The program works by directing USR to the following:

```
033C  JSR  $BBC7   ;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 $0310, 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  $BC0C ;COPY FAC1 INTO FAC2
033F  JSR  $BA30 ;MULTIPLY FAC1 BY FAC2; RESULT IN FAC1
0342  JSR  $BFB4 ;NEGATE FAC1
0345  JSR  $BAE2 ;MULTIPLY FAC1 BY 10; RESULT IN FAC1
0348  RTS
```

Return to BASIC, then POKE 784,76:POKE 785,60:POKE 786,3 and PRINT USR(8). You'll get −640, and so on. If you have no ML monitor, POKE the following numbers using BASIC to locations 828–839: 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 $BFED to calculate EXP of FAC1. Alternatively, use temporary storage areas. For instance, the following routine puts FAC1 into MFLPT form beginning at $57, then multiplies FAC1 by the MFLPT number it finds starting at $57. In effect, it is simply another way of multiplying a number by itself.

```
JSR  $BBCA
LDA  #$57
LDY  #$00
JSR  $BA28
```

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

The program below is a longer program example using ML arithmetic that illustrates several points. INIT sets FAC1 to 0 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 RUN/STOP key. PROC is the processing part; each digit is converted from ASCII ($30 to the character 0, for example), added to FAC1, and, if a further digit is wanted, multiplied by 16. PRINT outputs the result.

```
INIT   033C  LDA  #$04
       033E  STA  $FE      ;COUNT FOUR DIGITS
       0340  LDA  #$00
       0342  STA  $61      ;FAC1 NOW ZERO
       0334  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 RUN/STOP KEY
       0358  BNE  $035F
       035A  LDA  #$01     ;IF RUN/STOP PRESSED,
       035C  STA  $CC      ;FLASH CURSOR &
EXIT   035E  RTS           ;RETURN
       035F  JSR  $FFE4    ;GET CHARACTER FROM KEYBD
       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  $BD7E  ;ADD A TO FAC1
        0374  DEC  $FE    ;REDUCE COUNTER
        0376  BEQ  $0383  ;EXIT AFTER FOUR DIGITS
        0378  LDA  #$8B
        037A  LDY  #$00   ;SET POINTERS TO $8B
        037C  JSR  $BA28  ;MULTIPLY FAC1 BY MFLPT AT $8B (IE BY 16)
        037F  BEQ  $0355  ;BRANCH BACK (RELOCATABLE) FOR
        0381  BNE  $0355  ;NEXT DIGITS
PRINT   0383  JSR  $BDDD  ;CONVERT FAC1 INTO STRING AT $100
        0386  JSR  $AB1E  ;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
```

SYS828 accepts four-digit hex numbers and continues until the RUN/STOP key is pressed. The routine is relocatable. For binary-to-decimal conversion, POKE 829,8: POKE 844,130 after running.

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 $E0D3 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.

$E0D3 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 $E0BE 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 $E09E uses CIA 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 $E0BE is used to give a repeatable sequence (this eases debugging). The seed is replaced by $E09E for use.

One- or Two-Byte Random Numbers

These are often more useful in ML. You could use the following routine, which uses all four bytes, excluding the exponent, presumably increasing the result's randomness.

```
JSR  $E0BE  ;NEW RND NUMBER FROM OLD
LDA  $8C
EOR  $8D    ;COMBINE DATA BYTES
EOR  $8E    ;INTO COMPOSITE BYTE
EOR  $8F
```

Suppose you want something to happen 10 times in every 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 simply 48 plus a random number from 0 to 9. Another easy, but slow method is to check a result and go back if it's not in range.

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 (for example, $FB):

```
LDA  LO
ASL
ASL
CLC
ADC  #$odd number
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 $DC04) 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  #$odd number
ADC  LO
STA  LO
LDA  #$any number
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 65,536 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.

Series Calculations

All of the 64's mathematical functions are evaluated by series summation. Briefly, the value to be converted is first put into a smaller range. Trigonometric functions, for instance, repeat regularly, so their input values can be reduced (if large) by subtracting multiples of pi. Then a series evaluation works out the function's value, and finally an allowance is made for the initial scaling-down process.

In the 64, the ROM routine at $E059 sums the series. The following short example shows how:

```
LDY  #$03
LDA  #$40
JSR  $E059
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 Powers of Two

Although conversion of two-byte integers into floating-point form is often useful, the standard ROM routine at $B391 converts A (high) and Y (low) into the range from −32768 to 32767, the range of integer variables.

The following routine puts A (high) and Y (low) into FAC1 in the range 0–32767. Note that the 64 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  $BC49   ;CONVERT TO FAC1
```

Using RAM Under ROM

As we saw in Chapter 5, the normal operating system is stored in two ROMs, one at $A000–$BFFF (BASIC ROM), the other at $E000–$FFFF (Kernal ROM), which work together as the familiar BASIC language. These ROMs can be switched out either by hardware—when an external cartridge is sensed by the 64—or by software. With cartridges, the software controlled lines HIRAM and LORAM have higher priority than the EXROM and GAME lines which control the 64's sensing of plug-in ROMs, so the methods in this section will actually apply to 64s with or without a cartridge. For example, a language like Forth or Logo could exist on a cartridge from $8000 to $BFFF, but RAM BASIC could still be switched in to replace it, though there might be complications. For example, the new language might have no equivalent to POKE, or it could use RAM from $A000 to $BFFF itself. For simplicity, most of this section assumes that your 64 has no plug-in cartridges present.

Moving BASIC into RAM

The 64's software control allows both BASIC and Kernal ROMs to be switched out in favor of RAM. This means the entire BASIC language can be stored in RAM, where it can be modified. This feature alone gives the 64 possibilities that many computers don't have. Of course, there is a potential problem: Programs which write into these RAM areas will corrupt them, something impossible with a ROM-based language.

The process is simple enough. First, note that writing to the ROM area, *whether or not RAM is selected,* always writes to RAM. Second, HIRAM and LORAM are bits 1 and 0, respectively, in the control port. Thus, POKE 1,55 selects ROM BASIC, while POKE 1,54 switches out the BASIC ROM, and POKE 1,53 switches out both BASIC and Kernal ROMs. Note that bits 0 and 1 of location 0 must both be set for this process to work. Normally, this is automatic, but location 0 can sometimes be corrupted—see POKE in Chapter 3. From now on assume POKE 1,53 will allow us to modify either of the two ROMs.

If POKE 1,53 is done without preparation, BASIC will disappear as far as the 64 is concerned, and any BASIC will immediately crash.This program copies BASIC and the Kernal into RAM:

```
3 FOR J=40960 TO 49151: POKE J,PEEK(J): NEXT :REM MOVE BASIC TO RAM
4 FOR J=57344 TO 65535: POKE J,PEEK(J): NEXT :REM MOVE KERNAL TO RAM
5 POKE 1,53 :REM SWITCH TO RAM
```

A single long loop can't be used, because it sets the VIC chip wrongly. The program exploits the fact that POKE puts a byte into the RAM underlying ROM, even when ROM is selected. It's quite slow because of the slowness of BASIC, taking more than a minute to perform 16384 PEEK and POKE combinations. Program 8-2 uses ML to speed things up:

Program 8-2. ROM RAM

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
2 DATA 120,165,1,72,169,55,133,1,169,160   :rem 170
3 DATA 133,3,160,0,132,2,177,2,145,2,136   :rem 149
4 DATA 208,249,230,3,165,3,240,8,201,192,208
                                           :rem 109
5 DATA 239,169,224,208,229,104,133,1,88,96 :rem 36
10 FOR J=49152 TO 49193:READ X:POKE J,X:NEXT
                                           :rem 223
20 SYS 49152:POKE 1,53                     :rem 148
```

At this stage, POKE 1,55 and POKE 1,53 can be alternated with absolutely no visible effect, since the two versions of BASIC are identical.

Making Modifications to RAM BASIC

Small modifications. It makes sense to signal that modified BASIC is present. Type in POKE 41853,33. This replaces READY. with READY! so whenever you see READY! you'll be sure RAM BASIC is in use. Enter POKE 1,55 and POKE 1,53 alternately, to flip from one version of BASIC to the other. Generally, tables are the easiest features of BASIC to alter. For example, BASIC keywords are stored from 41118, starting with END, and these can be changed. It's easiest to keep keywords the same length as their original form, although it's possible (say) to redefine BASIC with single-letter and other short keywords, allowing very long (but hard to read!) lines of BASIC to be entered. Another example is the power-up message; a later program shows how this can be altered as far as is possible.

Larger modifications. Significant adjustments to BASIC require some ML knowledge and the information on BASIC's structure given in Chapter 11. At the simplest level, we can alter locations like $EAEA (delay between repeats) and $EB1D (cursor countdown) to alter cursor control. At a more advanced level, Chapter 6 shows how Computed GOTO and MERGE can be introduced into BASIC, and how the keyboard's tables can be redefined. These small adjustments are known as patches; $E4E0 (Filename) and $E4EC (Color) are two patches added to more recent 64 Kernal ROMs, correcting a tape name bug and a screen color effect. As another example, we can modify RUN to eliminate the test for RUN/STOP, the CONT line updates, and the end-of-program test (so END becomes necessary) quite easily, with a small speed increase of 3-1/2 percent. With BASIC in RAM, use these POKEs:

POKE 42960,160: POKE 42961,0: POKE 42962,177: POKE 42963,122: POKE 42964,208
POKE 42965,49: POKE 42966,24: POKE 42967,169: POKE 42968,4: POKE 42987,208

Upgrading ROMs

Earlier CBM computers had to have ROMs changed, at some expense, when improvements were made to BASIC. With the 64 this is no longer necessary. New versions of BASIC can be used as they become available. If the changes aren't too great, a program with ML to move BASIC to RAM, and a series of values to POKE into RAM to upgrade BASIC will be faster than loading the entire 16K from disk or tape.

Many 64s have a version number 0; PRINT PEEK(65408) in the Kernal to see this. Newer ROMs return 3. These have a few improvements: INPUT with a long prompt string works correctly with wraparound to the next line, and the screen edit bug is removed (where BASIC lines overrunning the bottom screen line, then backspaced, crash). Also, like very early 64s, POKEs to the screen are visible after {CLR} without needing color RAM POKEs.

If you'd like a different version of 64 ROM, an easy way to compare ROMs is with Program 8-3 or a similar comparison routine:

Program 8-3. Compare ROM

```
10 OPEN 1,8,4,"KERNAL 02"
20 OPEN 2,8,5,"KERNAL 03"
30 FOR J=14*4096 TO 65535
40 GET#1,X$:GET#2,Y$
50 IF X$<>Y$ THEN PRINT J; "NEW=" ASC(Y$+CHR$(0))
60 NEXT:CLOSE 1:CLOSE 2
```

Program 8-3 compares Kernals, assuming these to have been saved with a monitor, and commands like .S "KERNAL",08,E000,FFFF, but BASIC ROMs can be compared, too. Program 8-4 can be used at the start of a session; it converts version 0 to version 3 (there are no BASIC ROM differences):

Program 8-4. ROM Upgrade

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
20 FOR I=40960 TO 49151:POKE I,PEEK(I):NEXT
                                          :rem 143
30 FOR I=57334 TO 65535:POKE I,PEEK(I):NEXT
                                          :rem 151
40 POKE 1,53                               :rem 88
50 READ X:IF X>255 THEN A=X:GOTO 50       :rem 177
60 IF X<0 THEN END                        :rem 191
70 POKE A,X:A=A+1:GOTO 50                 :rem 136
100 DATA 58540,129                        :rem 121
110 DATA 58579,133,169,169,1,133,171,96   :rem 153
120 DATA 58587,134,2                      :rem 224
140 DATA 58748,32,240,233,169,39,232,180,217,48,6,
    24,105,40,232,16,246                  :rem 132
142 DATA 133,213,76,36,234,228,201,240,3,76,237,23
    0,96,234                               :rem 39
150 DATA 58914,145,229                     :rem 74
160 DATA 59911,32,218,228,169             :rem 166
170 DATA 59916,145,209,136,16,246,96,234  :rem 206
180 DATA 61332,76,211,228                 :rem 211
190 DATA 65408,3                           :rem 26
1000 DATA -1                               :rem 57
```

New Languages

Languages radically different from BASIC which are supplied on disk rather than cartridge require a forced LOAD into the BASIC area, then POKE 1,54, assuming the Kernal doesn't need to be altered. This takes more time, of course, and is less convenient than a cartridge. As an example, CBM BASIC 4 includes disk commands (CATALOG, DOPEN, etc.). To put BASIC 4 into the 64 as nearly as possible requires that its 20K (BASIC 4 fills $B000–$FFFF) be relocated to fit the 64's space as best it can ($A000–$CFFF, $E000–$FFFF, probably) and that the hardware references be converted from VIAs to CIAs, and to apply to the VIC and SID chips. In this way, some compatibility of software is possible, even down to matching zero page and other locations, but *complete* compatibility is impossible. ML programs which use some CBM entry points cannot work on the 64, for instance.

It's conceivable that a language like Apple BASIC might be transferrable to the 64, but even if this were done, there'd be difficulties over disk drive and other hardware incompatibility.

It's worth noting a potential problem when loading new versions of programs into the Kernal area. The command .S "ML",08,E000,FFFF doesn't save the very last byte, so LOAD "ML",8,1 works perfectly except for the final byte. Since this byte helps determine the IRQ vector, it's important that it should be set correctly.

RUN/STOP–RESTORE and RAM BASIC

RUN/STOP–RESTORE calls the IOINIT routine, which unless changed sets location 1 to 55. This would automatically convert RAM BASIC back to ROM. The easiest way to prevent this is to POKE 64982,53 which changes IOINIT in RAM. To see how this might work, POKE these values as well:

POKE 60634,1: POKE 60633,1: REM BACKGROUND AND BORDER WHITE
POKE 58677,0 : REM CHARACTERS TO BLACK

The first pair alters the 64's default screen color values; the last changes the color in CINT. Now press RUN/STOP–RESTORE; RAM BASIC is retained and the new colors appear. RAM BASIC is made completely secure against RESTORE in this way.

RESET and RAM BASIC

Reset with SYS 64738 uses IOINIT like RUN/STOP–RESTORE. But with RAM BASIC as we've developed it so far, it prints 51,216 bytes free! This happens because another routine, RAMTAS, which checks 64's RAM, detects the first ROM location at $D000, since RAM does in fact now exist up to there. To avoid this (the top-of-BASIC is set too high) RAMTAS must be modified; the easiest method is just to put in the desired top-of-BASIC. Thus, RAM BASIC can be made secure against a software reset.

Hardware resets are different. A reset switch sets the 64 for ROM BASIC, then goes through the usual ROM reset routines, including putting 55 into location 1. You may expect POKE 1,53 to revert to RAM BASIC without any problems, since resetting leaves the area alone, but there's one subtle effect of reset: It tests RAM by POKEing in a value, then reading it back, so it will detect BASIC ROM at $A000. However, it will also corrupt location $A000, POKEing in $55. POKE 40960, PEEK(40960) after any hardware reset so $A000 is correct.

Protection against either type of reset can also be arranged by putting bytes 195, 194, 205, 56, and 48 sequentially from $8004 on, with ($8000) holding the address to be jumped to on reset ($E37B to warm start BASIC, for example). The indirect vector ($8002) should point to RUN/STOP–RESTORE's destination (perhaps $FEBC, which returns from the interrupt generated by RUN/STOP–RESTORE).

Full Example of Modified BASIC in RAM

Program 8-5 shows a variety of features demonstrating everything we've seen so far. It sets RAM top to $8000 without checking RAM (as a result there's no delay before the sign-on message); it moves the start-of-screen to $8000, which means several adjustments to the VIC chip; BASIC starts at $0400; RUN/STOP–RESTORE and both resets are nullified as far as possible. Green characters on black are selected. Because this configuration simulates the PET/CBM, many programs for PET/CBM will run, provided they don't use ML routines which are too machine-specific.

Program 8-5. PET Your 64

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
1 PRINT"{CLR}{2 DOWN}SET 64 TO SIMULATE PET/CBM BA
  SIC"                                     :rem 96
3 PRINT"LINES 400-600 CONTROL COLOR SCHEME":rem 43
```

```
5 PRINT"LINE 12ØØ SETS BASIC TOP; YOU CAN":rem 143
6 PRINT"SELECT OTHER VALUES, E.G. $7ØØØ."  :rem 7Ø
8 PRINT"LINES 13ØØ-14ØØ SET BASIC & SCREEN"
                                           :rem 11Ø
9 PRINT"STARTS; YOU CAN USE OTHER VALUES"  :rem 52
11 PRINT"LINES 15ØØ-16ØØ SELECT VIC BANK; ":rem 97
12 PRINT"OFTEN THIS WON'T BE NEEDED."      :rem 177
14 PRINT"LINE 17ØØ PERSONALIZES THE SIGNON"
                                           :rem 116
16 PRINT"NEWER 64S DON'T NEED LINE 18ØØ"   :rem 223
18 PRINT"NOTE: IF YOU USE A HARDWARE RESET":rem 53
19 PRINT"SWITCH, RESETTING LEAVES MODIFIED"
                                           :rem 239
2Ø PRINT"BASIC INTACT IN RAM, EXCEPT FOR"  :rem 171
21 PRINT"LOCATION $AØØØ! POKE 4Ø96Ø,148"   :rem 158
22 PRINT"CORRECTS THIS."                    :rem Ø
25 PRINT "{3 DOWN}{RVS}PRESS RETURN TO CONTINUE
   {OFF}"                                  :rem 132
26 GET R$:IF R$="" THEN 26                  :rem 21
27 IF R$<>CHR$(13) THEN 26                  :rem 53
1ØØ FOR J=49152 TO 49193:READ X:POKE J,X:NEXT:SYS
    {SPACE}49152                            :rem 77
2ØØ POKE 1,53                              :rem 134
3ØØ POKE 41853,33                           :rem 89
4ØØ POKE 64982,53                          :rem 1ØØ
5ØØ POKE 6Ø634,Ø                            :rem 35
6ØØ POKE 6Ø633,Ø                            :rem 35
7ØØ POKE 58677,5                            :rem 56
1ØØØ POKE 64887,24Ø                        :rem 195
11ØØ POKE 649Ø4,162:POKE 649Ø5,Ø           :rem 138
12ØØ POKE 649Ø6,16Ø:POKE 649Ø7,128         :rem 248
13ØØ POKE 64913,4                           :rem 9Ø
14ØØ POKE 64918,128                        :rem 199
15ØØ POKE 64972,5                           :rem 98
16ØØ POKE 6Ø625,4                           :rem 89
17ØØ FOR J=58494 TO 585Ø5:POKE J,ASC(MID$("REVISED
     {2 SPACES}CBM",J-58493)):NEXT         :rem 149
18ØØ POKE 58587,134:POKE 58588,2           :rem 166
19ØØ SYS 64738                             :rem 213
5ØØØ DATA 12Ø,165,1,72,169,55,133,1,169    :rem 122
5Ø1Ø DATA 16Ø,133,3,16Ø,Ø,132,2,177,2        :rem 1
5Ø2Ø DATA 145,2,136,2Ø8,249,23Ø,3,165,3    :rem 119
5Ø3Ø DATA 24Ø,8,2Ø1,192,2Ø8,239,169,224    :rem 131
5Ø4Ø DATA 2Ø8,229,1Ø4,133,1,88,96           :rem 94
```

In the Appendices is a short PET reconfiguration program, which keeps BASIC in ROM. RUN/STOP–RESTORE and reset therefore destroy the configuration. This is partly cured by disabling RESTORE. These two programs show the greater versatility of RAM over ROM BASIC.

Programming RAM Under BASIC ROM

RAM under BASIC can store programs unrelated to BASIC, as well as graphics information which the VIC chip can access. Graphics can be POKEd in, and the VIC chip reset appropriately. But programs under ROM require the use of ML in some form to be usable. However, this needn't be particularly complex. For example, suppose some ML programs are stored from $A000 on. All that's needed to access them is this set of POKEs in RAM, perhaps located at 830 onward:

```
169, 54, 133, 1, 32, 0, 160, 169, 55, 133, 1, 96
```

Perhaps the easiest method would be to use READ and DATA statements to POKE the bytes in. After this, SYS 830 will carry out the ML routine at $A000 (assuming it ends with an RTS) by putting 54 into 1, jumping to $A000, and putting 55 into 1 on return. In this version, the ML mustn't call BASIC routines, of course. The digits 0 and 160 correspond to $A000 in low/high byte form. A routine at $B055, for example, would need 85, 176 instead of 0, 160. All this, including POKEs or LOADs of ML under BASIC, can be done from BASIC.

The following lines of ML allow access to RAM under BASIC ROM while keeping the Kernal in ROM:

```
LDA  #$36
STA  $01
                ; PERFORM PROCESSING HERE
LDA  #$37
STA  $01
BRK
```

You'll find that some ML monitors will allow you to assemble in the BASIC area after $A000 if you use M 0001 0001 and enter 36, to turn off BASIC ROM. Don't try to exit to BASIC without BASIC present—it'll crash!—and don't turn off the Kernal ROM by putting $35 into $0001, or the keyboard and screen handling will crash.

Modifying BASIC

64 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. Alternatively, 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, in order to avoid tedious retyping, it's important to save programs as they are written or to use a reset switch for emergency recovery. All the methods use ML, but this need not stop you, since the 64 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–$0309) determine where LIST executes.

There are over 20 such vectors. In addition, the CHRGET routine at $0073 (which fetches BASIC characters) 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 64738 or switching off and on will remove them, but otherwise even RUN/STOP–RESTORE leaves them untouched. This is intentional. On the other hand, RUN/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, because tokenization isn't possible, 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. 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. Again, Kernal vectors, like the three interrupt vectors, are all set to normal by RUN/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     ;ADDS 1 TO CURRENT ADDRESS
        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 $E3A2; SYS 58303 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 6510 requires that the return address −1 be pushed on the stack.

CHRGET can be changed. Try the following POKE129,234:POKE128,234: POKE131,234:POKE130,234 (without spaces between POKEs). This deletes the test for space characters, replacing the 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 0.5 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 with $0073 JMP $C000, 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 new commands in BASIC. A wedge, once inserted, is quite durable. As mentioned above, RUN/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.

Many utility programs use wedges. This example puts a 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 8-6 adds the single command ! to BASIC. When it executes, the screen colors are changed. When naming new commands, it's easiest to use a character that 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 8-6. BASIC Wedge Demonstration

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
4 FOR J=49152 TO 49203:READ X:POKE J,X:NEXT:SYS 49
  152                                      :rem 232
5 !:GOTO 5                                 :rem 254
10 DATA 169,76,133,115,169,13,133,116,169,192,133
                                           :rem 116
11 DATA 117,96,230,122,208,2,230,123,32,121,0,201
                                            :rem 69
12 DATA 33,240,3,76,121,0,165,123,201,2,240,247,15
   2                                       :rem 175
13 DATA 72,138,72,238,32,208,238,33,208,104,170,10
   4                                       :rem 203
14 DATA 168,76,115,0                       :rem 214
```

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 DOS 5.1 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 8-6 loads the following ML into the 64:

```
SETUP   LDA  #$4C   ;PUTS JMP $C00D INTO CHRGET
        STA  $73
        LDA  #$0D
        STA  $74
        LDA  #$C0
        STA  $75
        RTS
WEDGE   INC  $7A    ;MIMIC CHRGET
        BNE  INC
        INC  $7B
INC     JSR  $0079  ;A NOW HOLDS BASIC CHARACTER
        CMP  #$21   ;IS IT '!' ?
        BEQ  YES
NO      JMP  $0079  ;NO—JMP BACK TO CHRGOT. WEDGE UNUSED
YES     LDA  $7B    ;YES—CHECK FOR DIRECT MODE
        CMP  #$02
        BEQ  NO     ;DIRECT MODE—DON'T USE WEDGE
        TYA         ;PROGRAM MODE. USE WEDGE—
        PHA         ;SAVE X AND Y
        TXA
        PHA
        INC  $D020  ;EXECUTE '!' COMMAND; HERE WE
        INC  $D021  ;INCREMENT BORDER AND GROUND COLORS.
        PLA         ;PROCESSING OVER. RECOVER A AND X.
        TAX
        PLA
        TAY
        JMP  $0073  ;CONTINUE BASIC
```

SYS 49152 activates the wedge. Note that the entire routine is relocatable, apart from the address in SETUP. Long routines that won't fit the tape buffer can also be put at the top of BASIC memory.

$C021 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 $C021 jumps to CHRGET, there are no SYNTAX ERRORs, but the command becomes difficult to include in BASIC. Note as well that $C031 jumps to $0073. Of course, it immediately jumps back to $C00D, but $0073 always relocates.

Vectors

The main block of vectors starts at $300. ($0300) through ($030A) are vectors to BASIC. ($0314), ($0316), and ($0318) are vectors to IRQ, BRK, and NMI. ($031A) through ($0332) are vectors to Kernal routines, except ($032E) is unused.

Earlier RAM has a sprinkling of vectors, including ($028F), used by SCNKEY, the keyboard-reading routine, which allows keys to be intercepted—see the "Function Keys" definition and other keyboard redefinition programs in Chapter 6. Vectors ($0003) and ($0005) point to floating-to-fixed (and vice versa) number conversion routines, but neither is called by ROM—either through oversight or perhaps because the intention was to allow JMP ($0003) and JMP ($0005) to work on both the VIC and 64 and possibly future machines.

BASIC Vectors

There are six BASIC vectors, each called from the address three bytes before the vector's normal destination. For example, $A437 JMPs to ($0300), which is set normally to $A43A. Although this seems like useless extra execution time, it is the basis for the wedging technique. The vectors are listed here in order:

($300) IERROR: Vector to Error Message Routine

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, allowing an error routine to be specified at some line number.

($302) IMAIN: Vector to Main BASIC Loop

This usually points to $A483, and is called just after READY prints, before input from the keyboard. Try POKEing these values into 828 and the following addresses:

```
169, 42, 32, 210, 255, 76, 131, 164
```

Now POKE 770,60: POKE 771,3. The effect is to move the vector to point to these instructions:

```
LDA  #$2A     ;LOAD WITH *
JSR  $FFD2    ;OUTPUT IT
JMP  $A483    ;CONTINUE AS USUAL
```

Now the cursor expecting input is preceded by *. In fact, you can tell when the routine is called by the presence of the asterisk. IMAIN allows several possibilities: automatic BASIC line numbering, output of some message or prompt, and automatic LOAD and RUN, as Chapter 14 shows.

($304) ICRNCH: Vector to Tokenize Keywords Routine

This tokenizes BASIC, which is scanned while in the BASIC line input buffer at $200, converting keywords to tokens. This vector could be diverted, so new keywords could be recognized and converted into tokens. If this is done, ($308) and ($306) have to be altered, too.

($306) IQPLOP: Vector to Untokenize Keywords Routine

POKE these values from 828 and subsequent addresses:

72, 201, 58, 208, 10, 169, 13, 32, 210, 255, 169, 32, 32, 210, 104, 76, 26, 167

Now POKE 774,60: POKE 775,3. This simple routine compares the character to be listed with the colon; if it finds one, it starts a new line and prints a space, so LIST now puts every statement on a new line. (It doesn't test for colons within strings.) This sort of thing is useful with printers and could include a test for output device number 4.

($308) IGONE: Vector to Execute Next BASIC Program Token Routine

This is somewhat like CHRGET, but points only at tokens—it's 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*. We can intercept the routine and process our own tokens or, as in the next example, redefine a standard token.

Program 8-7. LET Vector Demo

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 DATA 32,115,0,201,136,240,6,32,121,0     :rem 85
11 DATA 76,231,167,32,155,183,142,33,208   :rem 176
12 DATA 76,234,167                         :rem 121
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 46
```

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 that LET 13, for example, sets the background light green. LET 13 now, in effect, POKEs $D021 of the 64 with 13, but does this much faster than POKE. FOR J=0 TO 15 :LET J :NEXT cycles the colors at great speed. The extra ML is this:

```
HERE     JSR  $0073    ;FETCH NEXT BASIC CHR
         CMP  #$88     ;LOOK FOR LET TOKEN
         BEQ  LETFND   ;BRANCH IF FOUND
         JSR  $0079    ;GOTCHR SETS FLAGS
         JMP  $A7E7    ;CONTINUE NORMALLY
LETFND   JSR  $B79B    ;CALCULATE 1-BYTE BASIC PARAMETER
         STX  $D021    ;PUT IT IN VIC REGISTER (BACKGROUND COLOR)
         JMP  $A7EA    ;CONTINUE, AFTER EXECUTION POINT
```

LET (and GO) and many rarely used mathematical functions lend themselves to this treatment, and may be helpful in dealing with some of the more tiresome commands needing POKEs and PEEKs, such as when controlling the SID sound chip.

($30A) IEVAL: Vector to Evaluate Single-Term Arithmetic Expression Routine

Normally set to point to $AE86, this directs execution to the subexpression evaluator routine, which fetches 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 vector table is to allow nonstandard terms, either string or numeric, to be defined. Thus, hex numbers beginning with $ can be introduced into BASIC, or binary numbers beginning with %. Examine the following assembly-style listing:

```
HERE  JSR  $0073  ;GET FIRST CHR OF TERM
      CMP  #$24   ;IS IT $?
      BEQ  YES    ;IF YES, BRANCH
      LDA  #$00   ;IF NO, SIMULATE
      STA  $0D    ;NORMAL BEHAVIOR
      JSR  $0079
      JMP  $AE8D
YES   JSR  $0073  ;GET FIRST CHR AFTER AND
      ......      ;PROCESS. PUT IN FAC1
      JMP  $0073  ;
```

Set ($30A), 778 and 779, to point to the address of HERE.

Program 8-8, is a BASIC program that uses the principles just explained to add hex numbers to BASIC. For example, POKE $D020,1 works correctly.

Program 8-8. Add $ Commands

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
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 240,6,32,121,0,76,141,174,162,2     :rem 104
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 99,162,144,56,32,73,188,76,115,0    :rem 193
100 FOR J=828 TO 905:READ X:POKE J,X:NEXT    :rem 64
110 SYS 828                                  :rem 51
```

To compute results in the range 0–65535, a modified FXFLPT routine has to be used, as explained earlier.

Kernal Vectors

Twelve Kernal routines are vectored through RAM addressed from ($31A) through ($332); all, except ILOAD and ISAVE, are called immediately after entering the Kernal jump table. Open and close routines, routines to fetch and output characters, and the RUN/STOP key test all have indirection and allow modifications to be made to the Kernal even in ROM. LOAD and SAVE store address parameters before entering their vectors. A short list of the vectors is given below:

($31A)	**IOPEN:**	Vector to Kernal OPEN Routine
($31C)	**ICLOSE:**	Vector to Kernal CLOSE Routine
($31E)	**ICHKIN:**	Vector to Kernal CHKIN Routine
($320)	**ICKOUT:**	Vector to Kernal CKOUT Routine
($322)	**ICLRCH:**	Vector to Kernal CLRCHN Routine
($324)	**IBASIN:**	Vector to Kernal CHRIN Routine
($326)	**IBSOUT:**	Vector to Kernal CHROUT Routine

($328)	**ISTOP:**	Vector to Kernal STOP Routine
($32A)	**IGETIN:**	Vector to Kernal GETIN Routine
($32C)	**ICLALL:**	Vector to Kernal CLALL Routine
($32E)	**IUSRCMD:**	Vector to Kernal User-Defined Routine
($330)	**ILOAD:**	Vector to Kernal LOAD Routine
($332)	**ISAVE:**	Vector to Kernal SAVE Routine

As a fairly simple example, using an asterisk as before, the routine below intercepts CHROUT and causes it to output an extra asterisk. POKE these values into 828 and the following:

```
72, 169, 42, 32, 202, 241, 104, 76, 202, 241
```

Now POKE 806,60: POKE 807,3 to alter the vector to $033C. The ML is listed below:

```
PHA          ; SAVE OUTPUT CHARACTER
LDA  #$2A    ; LOAD ASTERISK
JSR  $F1CA   ; OUTPUT ASTERISK, NOT WITH FFD2, OF COURSE
PLA          ; RECOVER CHARACTER
JMP  $F1CA   ; OUTPUT IT, CONTINUE
```

Following SYS 828, any use of CHROUT prints an asterisk—including all BASIC messages like READY, which appears as R*E*A*D*Y*.

Interrupts

Defaults for NMI, BRK, and IRQ vectors (Non-Maskable Interrupt, BReaK, and Interrupt ReQuest vectors) are permanently stored in the top six bytes of the 6510's memory. With the 64, NMI jumps to $FE43, BRK to $FE66, and IRQ to $FF43. Hardware reset (see Chapter 5) has no indirect vectors apart from the optional cartridge's start address.

($318) NMINV: Vector to NMI Routine

NMI processing is complex. One reason is that both RS-232 and the RESTORE key are handled by it. From $FE43, NMI goes through this sequence:

$FE43	Vectors through ($318), normally to $FE47
$FE47	Saves A, X, Y with PHA:TAX:PHA:TAY:PHA
$FE4C	Disables NMIs
$FE51	Checks for RESTORE key or for RS-232 or other CIA interrupts
$FE56	RESTORE key; checks for cartridge; if found JMP ($8002)
$FE5E	Tests for RUN/STOP key; if found, restore BASIC
$FE72	RS-232 or CIA interrupts; processes RS-232
$FEBC	Restores Y, X, A with PLA:TAY:PLA:TAX:PLA
$FEC1	Exits with RTI

To use NMI interrupts successfully requires that ($0318) be redirected with a correct exit, and that the CIA timers be set properly.

A simple example is to redirect ($0318) to $FEC1; RESTORE always now immediately goes to RTI, so RUN/STOP–RESTORE cannot work. Chapter 5 has an example showing how RESTORE generates an interrupt.

($314) CINV: Vector to IRQ Routine
($316) CBINV: Vector to BRK Routine

The IRQ and BRK instruction interrupts are conceptually tricky, though their processing is fairly straightforward. Both an IRQ interrupt from CIA 1 (see Chapter 5) and a BRK instruction jump to $FF43, provided interrupts aren't masked by SEI, and assuming ($FFFE) hasn't been changed in RAM. After $FF43, A, X, and Y are saved on the stack; then the two types of interrupt are separated by checking for the BRK flag in the status register.

BRK. This type of interrupt vectors through ($316), normally set to the restore sequence in NMI, so SYS to a location holding a zero byte typically has the same effect as RUN/STOP–RESTORE. Monitors often change ($316) to point to their own start address, so BRK at the end of the ML returns control to the monitor.

IRQ. Interrupts of this type are vectored through ($314), normally to $EA31. From $EA31, IRQ interrupts go through this process:

$EA31	Update clock
$EA34	Process cursor, tape motor; scan keyboard
$EA7E	Clear interrupt flags
$EA81	Restore A, X, Y with PLA:TAY:PLA:TAX:PLA
$EA86	Exit with RTI.

IRQ interrupts usually occur 60 times every second, unless turned off, or during tape and disk access; CIA 1 controls the frequency. Adding extra ML, to be processed immediately before the usual interrupt ML, must allow for this. Successful use of the vector requires preexisting ML, exiting typically with JMP $EA31, so the TI clock and keyboard and so on work normally. Also, a routine to alter the vector in ($0314) to point to the new ML is needed. Since interrupts are already taking place, they must either be stopped or an ML routine must be used to be sure that the address is changed before another interrupt occurs.

Interrupt-Driven Background Programs

A background program runs with the main program. Interrupts allow programming feats which are otherwise impossible; their common feature is periodicity. Playing music, updating graphics, and printing the current time are examples of background programs which interrupts allow on the 64. Features can be added at will, giving long, complex background programs, if this is required.

NMI-driven background programs. NMI interrupts aren't maskable; they offer interrupts with near-perfect regularity. CIA 2 generates NMIs. The CIA's timers can be used together, allowing repeats to occur over a time range from a few microseconds to an hour or so without special counters for the purpose. Also, the NMI vector is simple to move, and it is easy to turn off NMIs. Disk and tape cannot work with these routines, however. These devices go through the motions, but the NMIs throw their timing off; IRQ interrupts, on the other hand, pause while these operations take place, but are automatically reinstated afterward.

Program 8-9 uses NMIs to drive a short program, which simply changes the background color. The interrupt frequency is chosen to produce narrow bands of color on the screen; BASIC runs completely normally over this. Alter the timer parameters to see the effect of changing the rate at which NMIs are generated:

Program 8-9. NMI Demo

```
1 REM USING NON-MASKABLE INTERRUPTS WITH BASIC
                                        :rem 177
2 REM TO PROVIDE COMPLETELY REGULAR TIMING:rem 170
3 REM E.G. TO PLAY MUSIC NOTES, DISPLAY THE :rem 5
4 REM TIME, DISPLAY GAME SCORES ETC.       :rem 124

6 REM THIS VERSION USES BOTH TIMERS OF CIA#2,
                                        :rem 156
7 REM SO DELAYS CAN VARY FROM A FEW MILLIONTHS
                                        :rem 237
8 REM OF A SECOND UP TO AN HOUR OR MORE.  :rem 197
10 REM THIS DEMO INCREMENTS SCREEN COLOR FAST,
                                         :rem 51
11 REM PRODUCING NORMALLY IMPOSSIBLE STRIPES
                                        :rem 128
100 FOR J=49152 TO 49162: READ X: POKE J,X: NEXT
                                         :rem 11
110 POKE 792,0: POKE 793,192: REM NMI VECTOR TO C0
    00                                  :rem 222
120 POKE 56589,127:{2 SPACES}REM ALL NMIS OFF
                                        :rem 162
130 POKE 56589,130:{2 SPACES}REM TIMER B NMI ENABL
    ED                                   :rem 68
140 POKE 56580,99: POKE 56581,1: REM SET TIMER A
                                          :rem 5
149 REM 16+1 IN TIMER A (1) SETS LATCH, (2) STARTS
     TIMER A:-                          :rem 132
150 POKE 56590,17:{3 SPACES}REM START TIMER A
                                        :rem 208
160 POKE 56582,2: POKE 56583,0:{5 SPACES}REM SET T
    IMER B                              :rem 203
168 REM 64+16+1 IN TIMER B (1) COUNTS TIMER A,
                                        :rem 129
169 REM (2) SETS LATCH, (3) STARTS TIMER B:-
                                        :rem 109
170 POKE 56591,81:{3 SPACES}REM START TIMER B
                                        :rem 213
180 DATA 72,138,72,152,72,238,33,208,76,81,254
                                        :rem 232
181 REM PHA/TXA/PHA/TYA/PHA/INC D021/JMP FE51
                                        :rem 135
182 REM OR TRY EG. 0,4 IN PLACE OF 33,208  :rem 38
183 REM TO INCREMENT TOP LEFT SCREEN CHARACTER
                                         :rem 51
```

RUN/STOP–RESTORE will return the system to normal.

IRQ-driven background programs. IRQ programs are quite popular because early CBM machines couldn't use NMIs. They are a little more complex than NMI programs, but need no special CIA knowledge. Chapter 13's music-playing program is an example; it plays a three-part composition while BASIC runs normally.

To create an IRQ-driven program, you should first choose an area of RAM to store the ML—$C000 onward is the obvious choice on the 64, but there are other locations. Next, you must write the initialization routine, typically something similar to the following, which sets the vector to $C00D:

```
$C000  SEI
$C001  LDA  #$C0
$C003  STA  $0315
$C006  LDA  #$0D
$C008  STA  $0314
$C00B  CLI
$C00C  RTS
$C00D  (background program starts here)
```

Note that you must disable all interrupts with SEI before changing the vector, and then reenable them with CLI. It's also possible to change ($0314) from BASIC. For example, this line sets the IRQ vector to $C000:

```
POKE 56334,0: POKE 788,0: POKE 789,192: POKE 56334,1
```

The next step is to write the background ML, which in our example will start at $C00D and typically exit with JMP $EA31, the normal interrupt routine. Here are a few things to keep in mind when programming with interrupts.

- A, X, and Y can be used independently of BASIC. Because the BASIC values at the time the interrupt happens are saved automatically and restored on return from interrupt, your ML is self-contained.
- The simplest termination of an IRQ routine is JMP $EA31. 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 $EA31, though. Note that RTI restores the status register to its pre-interrupt value, so CLI isn't necessary.
- 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.
- *Polling* means that, during each interrupt, locations are PEEKed to see if action is required. Chapter 16 has an interrupt-driven program to read joysticks; at regular intervals, the joystick hardware addresses are read in ML and transferred to a convenient location. Processing is far faster than the BASIC equivalent. Another example is reading location $C5 (197) to see if a key is held down; unlike GET's once only action, this allows keys to act as long as they are held down.
- POKEs into the background program can be a useful control if a program has several functions. To illustrate, here is a simple example:

```
LDA #0
BNE INTML
JMP $EA31
```

If this is the first ML of an IRQ program, the branch will never be taken and the effect is negligible. But POKEing a value other than 0 into the location following the LDA instruction will activate whatever ML has been put in at the location labeled INTML by the assembler. So completely invisible ML interrupt programs can be activated from BASIC.

- The speed of background programs can, of course, vary. Suppose we have a relatively slow routine, perhaps to fill a screen with graphics. Is there a chance that the interrupt might itself be interrupted and the program crash? Since an IRQ interrupt, in effect, performs SEI whenever it occurs, background programs driven by IRQ can be slower than 1/60 second. If *every* repetition is as slow as this, the normal program will hardly get a chance to run.

  NMIs have no disable flag; if interrupts occur faster than the background program takes to execute, the 64 will lock up.

Chapter 9

Mixing BASIC with Machine Language

- RAM Available for ML Routines
- Combining BASIC and ML
- Relocating ML

Mixing BASIC with Machine Language

While most programmers are happy to use BASIC, machine language (ML) offers increased speed and power. This short chapter explains how ML programs and data can be incorporated into BASIC, and how BASIC can be used to load in and relocate your ML routines.

RAM Available for ML Routines

The 64 has plenty of RAM; the only problem likely to arise is making all of your routines compatible with each other. The best way to achieve this is with the relocating loader technique, discussed later in this chapter. However, with proper planning, there is plenty of room in the 64 for your ML routines and BASIC programs, too.

The 64 has several areas of RAM that are particularly suited for storing ML, the largest of which is $C000–$CFFF (49152–53247), which allows 4K to be stored, and is isolated from BASIC. This is generally the best choice, except in the sense that it is the favorite of most ML programmers using the 64, so careful partitioning will be required to fit in more than one routine. You can calculate the length of your ML, subtract that from the top of this block, and use the result as the LOAD address of your routine. Sometimes this will help you stay out of the way of other programs that use this area.

The RAM under the BASIC ROM ($A000–$BFFF) is an 8K block, which requires the methods explained in Chapter 8 to be usable. This area is relatively unpopular, but access isn't really difficult. ML in this area cannot generally use BASIC subroutines, though. Note that the 8K under the Kernal is also usable, but liable to more difficulties than BASIC, since the machine's input and output operations are controlled by the Kernal.

The RAM normally used for BASIC program storage can usually be reduced without affecting BASIC programs. Either the start of BASIC can be moved up, or the top moved down, or both. The top is often used, since any CBM machine can use this area, while not all of them can raise the bottom of BASIC as easily. *Supermon* is stored at the top of BASIC, and resets end-of-BASIC pointers below itself so BASIC strings won't corrupt it. Autostart plug-in cartridges occupy $8000–$9FFF of this area.

Smaller blocks of RAM are sometimes useful when writing autorun programs (see Chapter 15) or converting VIC-20 subroutines for the 64. The main areas are $2A7–$2FF (679–767) and $334–$3FF (820–1023); part of the latter is used by tape and is secure if tape isn't used after BASIC is loaded. Even if you use tape, $334–$33B (820–827) is free to be used for vectors or flags. The stack, $100–$1FF (256–511), is partly usable—the low end, and only the low end, is safe as long as there aren't many GOSUB or FOR-NEXT calls. Free zero page RAM includes locations 2–6 and $FB–$FE (251–254). Assuming certain ML math calls and RS-232 communications aren't used, $F7–$FE (247–254) is free.

Combining BASIC and ML

DATA statements. The easiest way to combine ML with BASIC is to store the ML as BASIC DATA. When the BASIC loader is run, the numbers are read and POKEd into memory, for use by SYS or USR. Chapter 6 has examples of this technique. This type of program is self-contained and can be loaded and saved like any other BASIC program, and there are generally no problems. The drawback is that each ML byte is stored on average in about four BASIC bytes. For instance, $EA is held as 234 with a comma. Therefore, it is better to store long ML programs another way, usually as object code.

If you've written some ML that works correctly, it's convenient to have a program to read it from memory and write it as DATA statements. Program 9-1 will do this for you, so you will not have to do it by hand.

Program 9-1. DATA Maker

```
100 INPUT "START";A
110 INPUT "{2 SPACES}END";E
120 INPUT "FIRST LINE#";L
130 INPUT "LINE LENGTH";LL
140 PRINT "{CLR}
150 PRINT "{HOME}" L "DATA ";
160 PRINT MID$(STR$(PEEK(A)),2) ",";
170 A=A+1:IF A>E THEN END
180 IF POS(0)<LL THEN GOTO 160
190 PRINT "{LEFT} {HOME}{4 DOWN}L="L"+1:A="A"
    {LEFT}:E="E"{LEFT}:LL="LL":GOTO140"
200 POKE 198,5:POKE 631,19:POKE 632,13:POKE 633,13
    :POKE 634,13:POKE 635,13
```

Suppose you've written ML starting at 49152. To use Program 9-1, load and run it, entering the start (49152) and end addresses of the ML you want DATA statements made for, the line number of the first DATA statements (perhaps 0 or the location of your routine), and a maximum line length, say, 60. If you aren't sure of the end address, put in a figure that is too large, then edit the last few DATA lines.

When the program has finished creating the DATA, delete the lines of the "DATA Maker" program; the DATA lines to store your ML in BASIC will be left. You will need to add some BASIC to POKE in the data:

FOR J=SA TO EA: READ X: POKE J,X: NEXT

The above line (where SA is the starting address and EA is the ending address) or something similar should work fine.

REM statements. ML can be stored in REM statements. To do this, enter a line like 0REMXXXXXXXXXXXX. Now POKE 2054,238: POKE 2055,1: POKE 2056,4: POKE 2057,96. Assuming that BASIC starts at $0801, SYS 2054 calls the following ML, which these POKEs represent:

INC $0401
RTS

This increments the character in the second column of the first row of the screen. A complication is that the ML must not contain any null bytes, or on editing these will be treated as ends of lines and spoil the ML. (You could use LDX #1:DEX in place of LDX #0.) Another complication is that BASIC lines normally have a maximum length of 80 characters. By adjusting the link addresses the limitation is easy to overcome. A line treated in this way LISTs oddly, and mustn't be edited. Using quotes after REM and starting at 2055 will cause the program to LIST without keywords.

Strings. ML can also be stored as a string. The line ML$="4C48D2AAD191D3" illustrates an alternative to DATA that's sometimes encountered. It requires modifications to the DATA writing program to find and print separate hex bytes. This method saves space compared with ML stored as numbers, but is slower to decipher and POKE back in. Since the 64 has a lot of RAM, the earlier method is used more often.

Block LOADs. Chapter 6 explains how to load a block of ML. For an example, see Chapter 12's character editor which allows user-defined characters to be saved to disk for use later. To bypass the 64's attempt to GOTO the first line after a program-mode LOAD, you'll need something like the following line, which loads the ML file only once:

```
0 IF X=0 THEN X=1: LOAD "ML",8,1
```

As we've seen, block LOADs save time compared with DATA READs and POKEs, so this technique (or one of those following) is desirable with ML of any substantial size.

Consolidated BASIC and ML. These programs, sometimes called hybrids, contain ML immediately after BASIC. BASIC LISTs normally, but since the three null bytes marking its end are earlier than the end-of-BASIC pointer, there's space for extra ML which doesn't show on listing. Programs like this can't be edited, or the ML will be moved in memory and probably will not function properly.

An example of this method is *64 Term.* The ML needed to run the 64 modem is loaded as BASIC, but the BASIC is reduced to a single SYS command. In this extreme case, the only use of the BASIC is to run the ML, saving the user from having to force-load and then use a SYS call. Some games include their graphics definitions after BASIC and are hybrids in another sense.

Here is an explanation of how to alter an ML program so that it can be loaded and run. Suppose the ML starts at $0810, just after the start of BASIC. If you have the source code, you can reassemble the program at $0810, and some monitors have an .N relocation feature. We want BASIC to LIST as 0 SYS 2064, our ML to start at $0810 (2064), and both to be loadable simultaneously. The process is shown below (a simple ML routine which changes the screen color serves as an illustration).

Step 1. With a monitor, like *Supermon,* load the ML which is located at $0810. The sample program is shown after being loaded and examined with the M command:

```
.M 0810 0810
.:  0810  EE 21  D0 60  00  00  00  00
```

This is equivalent to INC $D021 / RTS.

Step 2. Add the SYS call. Here we'll put 0 SYS2064:

```
.M 0800 0808
.:  0800 00 0A 08 00 00 9E 32 30
.:  0808 36 34 00 00 00 00 00 00
```

Reading from $0800, this holds a null byte; the link address in low/high form, $080A; the line number 0; SYS2064 (9E is the tokenized form of SYS and it's followed by ASCII numbers); and three end-of-program zero bytes and a few more filler bytes before $0810.

Step 3. Save to disk or tape. The trick is *not* to save the starting null byte. If you do, you'll need to force-load the program to make it work. So use something like this (using your monitor), with the ending address *plus one* for the final parameter:

```
.S "ML AS BASIC",08,0801,0830
```

Step 4. Test the result. It should load normally, LIST as 0 SYS2064, and run correctly.

In practice, it's common to find other characters after SYS. For example, a colon, then backspaces, and a copyright message will make the SYS command invisible when listed to screen.

Relocating ML

Moving relocatable ML. Where several utilities might be required in RAM, it makes sense to write them in such a way that they detect and fill the next highest available space. *Supermon* and some of the utilities in Chapter 6 are written like this; they're not dependent on being put into a fixed place in RAM. The techniques that follow put ML into the top of BASIC, after first lowering the end-of-BASIC pointer by the correct amount. These techniques work with any CBM machine. Since the 64 has RAM available at other places than top of BASIC, it's possible to modify the method, for example, to allocate $C000–$CFFF to ML routines, but some pointer other than top of BASIC has to be chosen to locate the individual ML routines correctly. Loaders to put utility ML into the top of BASIC can start with this:

```
100 T=PEEK(55) + 256*PEEK(56) : REM T=ORIGINAL TOP OF BASIC
110 S=T-N: : REM EG S=T-50 LOWERS BY 50 BYTES
120 POKE 56,S/256: POKE 55,S-INT(S/256)*256: CLR: REM SET NEW TOP
130 S=PEEK(55) + 256* PEEK(56): REM RECOVER VALUE OF S=START OF ML
```

which lowers BASIC's top by an amount specified in line 110 and sets S equal to the start of the new area, ready for a loop of the type:

```
FOR J=S TO S+49: READ X: POKE J,X: NEXT
```

which reads 50 bytes of ML and puts it into protected RAM.

This is fine for ML which is relocatable, using only branches and calls to Kernal or BASIC routines. But ML like this for the 6510 is difficult to write; for example, a block of ML with internal subroutine calls, like C000 JSR C100, cannot work if the ML is simply shifted in RAM. Just as some assemblers (see Chapter 7) use special loader programs to put ML anywhere in RAM, we can write loaders which POKE ML anywhere, modifying it where necessary.

Relocating ML with BASIC. The following technique involves a lot of work, but the versatility of the resulting utility makes it well worthwhile. If you wish to write ML subroutines to be as versatile as possible, bear in mind that it's always simpler for the user if ML is freely relocatable. We'll use the following loader:

```
100 T=PEEK(55)+256*PEEK(56) :REM TOP OF MEMORY
110 L=T-N :REM N=NUMBER OF BYTES OF CODE; L=LOWERED MEM.TOP
120 FOR J=L TO T-1: READ X% :REM ML HELD IN DATA STATEMENTS
130 IF X%<0 THEN Y=X%+T: X%=Y/256: Z=Y-X%*256: POKE J,Z: J=J+1
    :REM Y IS RELOCATED VALUE CALCULATED FROM NEG.X%
140 POKE J,X%: NEXT :REM COMPLETE PROCESS FOR ALL VALUES
150 POKE 55,L-INT(L/256)*256: POKE 56,L/256: CLR:REM RESET TOP-OF-MEMORY
```

To convert code into data which this program can use, enter the code into RAM, then print or write out the disassembled version. (A disassembler giving decimal values of locations is helpful.) Next, mark all the absolute addresses which need changing during relocation, replacing each by its offset from the end of the program; that is, count backward from the end of program *plus one*, the result being a negative number. See the example; this is easier than it might seem.

Convert the bytes into DATA statements and enter them. Note that each new negative value replaces *two* bytes as a rule. Enter the value of N in line 110, then 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 is a short routine which fills the first 256 screen positions with the character stored as the last byte of the routine. It has a subroutine call, a load from an absolute address, a store to an indexed absolute address, and two branches. The branches, because of their relative addressing mode, relocate without modification; so do the implied mode instructions, and the immediate mode instruction. The store to the indexed absolute address does not have to be relocated because the target address is not within the code to be relocated. The only addresses to be relocated are those circled.

Figure 9-1. Relocating ML Example

```
 32  171  2   02A7  JSR   $02AB
 96           02AA  RTS
160    0      02AB  LDY   #$00
173  183  2   02AD  LDA   $02B7
153    0  4   02B0  STA   $0400,Y
200           02B3  INY
208  250      02B4  BNE   $02B0
 96           02B6  RTS
 32           02B7  .BYTE $20
```

Counting back from the end, we find that $02AB is the thirteenth byte and $02B7 is the first; so −13 and −1 respectively replace all occurrences of these two addresses. The DATA statement is therefore:

```
10 DATA 32,-13,96,160,0,173,-1,153,0,4,200,208,250,96,32
```

The number of bytes in the program is 17, so line 110 becomes:

```
110 L=T-17
```

After relocation, the new starting address of the ML can be found with:

```
PRINT PEEK(55)+PEEK(56)*256
```

and the ML can be started with:

```
SYS PEEK(55)+PEEK(56)*256
```

Relocating ML with ML. This technique is similar and much faster. *Supermon* uses this method. ML of course has no out-of-range values in the way BASIC has negatives; instead use null bytes as markers.

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. That is, replace the absolute addresses with the twos complement of the number of bytes from the end of the program. (See Chapter 7 for information on calculating the twos complement.) In the example above, we found that $02AB was 13 ($000D) bytes from the end of the program. Using the twos complement of $000D, you would replace the address with JSR $FFF3. Address $02B7 has a displacement of 1 byte, so we replace LDA $02B7 with LDA $FFFF.

Finally, put in a BASIC call (as shown earlier in this chapter) to the relocator program, the relocator (from Program 9-2, below), and the ML you wish to relocate (preceded by a unique marker byte not found anywhere in the ML to be relocated) together in RAM, and save.

The relocator program works by starting at the end of the ML to be relocated and working backward, moving the bytes one by one to the top of available memory (as indicated by the pointer in locations 55 and 56). If a zero byte is found, the relocator examines the next byte. If it is also a zero, then a zero byte is moved. But if it is not a zero, then an additional byte is retrieved and these two bytes (the displacement you calculated) are added to the top-of-memory address to compute the proper absolute address for the relocated ML.

This continues until the marker byte—which separates the relocator from the ML being relocated—is encountered. As presented in Program 9-2, 222 is used as the marker. If the ML you wish to relocate contains the byte $DE (222), you'll need to change this marker to some other value not found in your code. You can do this by changing the 222 in line 16 to the desired value.

Finally, the relocator program lowers the value in the pointers to the top of BASIC program storage (locations 55 and 56) and string storage (locations 51 and 52) to protect the relocated ML from BASIC. It then executes the ML by jumping to the first byte of the relocated code. As with the BASIC relocator, the starting address of the relocated code can be found with:

```
PRINT PEEK(55)+PEEK(56)*256
```

and the ML can be restarted with:

```
SYS PEEK(55)+PEEK(56)*256
```

Program 9-2. ML Relocator

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
1 REM 222 IN LINE 16 IS MARKER VALUE FOR START OF
  {SPACE}ML                                  :rem 79
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 201,222,208,237,165,55,133,51,165,56,133,5
   2,108,55,0                                :rem 131
```

For an example of how to use this relocator program, add the lines shown in Program 9-3 to Program 9-2. This will create a machine-language relocated version of the example routine from Figure 9-1. Line 30 creates a program file called RELOCATE TEST directly on the disk. Line 40 writes out the data for a BASIC SYS call from line 5, line 50 writes out the ML relocator program from lines 10–16, and line 60 writes the byte that separates the relocator from the code to be relocated. Line 70 reads the ML to be relocated from the DATA statement in line 20. Notice how this data differs from that created for the BASIC relocator in the previous section. When you run the program, it creates a program on disk called RELOCATE TEST, which you can load and run like a BASIC program. The RELOCATE TEST program will move the routine to the top of available memory—adjusting addresses in the process—then lower the top-of-memory pointer and execute the routine.

To use this program for your own ML, replace line 20 with DATA statements containing your ML (modified for relocation as described above), and change the value of L (line 25) to reflect the number of items in your DATA statements.

Program 9-3. Relocating Program Generator

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
5 DATA 11,8,0,0,158,50,48,54,49,0,0,0        :rem 2
20 DATA 32,243,255,0,96,160,0,0,173,255,255,0,153,
   0,0,4,200,208,250,96,32                   :rem 227
25 L=21                                      :rem 83
30 OPEN 1,8,8,"0:RELOCATE TEST,P,W":PRINT#1,CHR$(1
   );CHR$(8);                                :rem 192
40 FOR I=1 TO 12:READ X:PRINT#1,CHR$(X);:NEXT : RE
   M BASIC SYS CALL (5)                      :rem 124
```

```
50 FOR I=1 TO 105:READ X:PRINT#1,CHR$(X);:NEXT : R
   EM ML FOR RELOCATOR (10-16)               :rem 30
60 PRINT#1,CHR$(222);: REM SEPARATOR CHARACTER
                                            :rem 242
70 FOR I=1 TO L:READ X:PRINT#1,CHR$(X);:NEXT : REM
    ML TO BE RELOCATED (20)                 :rem 110
80 CLOSE 1                                   :rem 15
```

Chapter 10

Vocabulary of the 6510 Chip

- The 6510 Instruction Set

Chapter 10

Vocabulary of the 6510 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. An 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 6510 load instructions (like LDA). 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. Each bit of the register serves as an indicator (flag) for a different condition:

bit 7: Negative (N) flag. Matches bit 7 of the result of the operation just completed, which indicates negative numbers in twos complement arithmetic.

bit 6: Overflow (V) flag. Indicates an overflow (result too large for one byte) in twos complement operations.

bit 5: Unused; always set.

bit 4: Break (B) flag. A BRK instruction was encountered.

bit 3: Decimal (D) flag. When set, all math is performed in decimal (BCD) mode.

bit 2: Interrupt disable (I) flag. When set, interrupts are ignored.

bit 1: Zero (Z) flag. Indicates that all bits are zero in the result of the operation just completed.

bit 0: Carry (C) flag. Holds the carry bit for addition, or borrow for subtraction.

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).

The 6510 Instruction Set

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 0 (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 on this basis. This example adds 123 (decimal) to the contents of locations 2 and 3, which are assumed to contain, in ascending order, four binary-coded digits.

```
SED         ; SET THE DECIMAL FLAG
CLC         ; CLEAR CARRY FLAG
LDA  $03    ; WE'VE ASSUMED THE BCD DATA IS STORED IN NORMAL ORDER
ADC  #$23   ; WITH LOW BYTES FOLLOWING HIGHER ONES, NOT 6510 ORDER
STA  $03    ; ADD 23 DECIMAL
LDA  $02
ADC  #$01   ; ADD 01 DECIMAL PLUS POSSIBLY CARRY BIT EQUIVALENT TO 100
STA  $02
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 6510 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 logical 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 0, the result is 0. The resulting byte is stored in A. If A now holds 0—that is, all its bits are 0—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 0:

```
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.

$$\begin{tikzcd} \boxed{C} & \boxed{7\ 6\ 5\ 4\ 3\ 2\ 1\ 0} \arrow[l] & \boxed{0} \arrow[l] \end{tikzcd}$$

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 contents of memory or the accumulator left by one bit position, moving 0 into the 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 0 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 C 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 if the value compared was smaller, 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 0, 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 according to 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 0 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 alternative 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 CHARACTER 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 (Branch if MInus) 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 highest 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 is 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  $0400,X   ;START OF SCREEN (64)
     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:= $FE, PCH:= $FF

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. 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.

   Whenever the 6510 encounters a BRK, it looks to locations $FFFE and $FFFF for the address of the next instruction. In the 64's ROM, locations $FFFE and $FFFF point to a routine beginning at $FF48. 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 RUN/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 to 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 to 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; when the machine is switched 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 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 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 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. Thus, $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  #$27
BCS  EXIT; IF X>39 (#$27)
...
```

   The test routine is part of a graphics plot program; location $FE holds the horizontal coordinate, which is to be in the range 0–39 to fit the screen. The comparison causes exit, without plotting, when X holds 40–255.

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, 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 according to 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 0
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. Since its maximum range, 255 bytes, is often insufficient, several loops may be necessary. This routine moves 28 bytes from ROM to RAM, including the CHRGET routine.

```
     LDX  #$1C
NEXT LDA  $E3A2,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 SECOND BYTE
DEY
ORA  (PTR),Y  ;ORA WITH FIRST BYTE
DEY
ORA  (PTR),Y  ;ORA WITH ZEROTH 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 Exclusive 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 crossed.

Flags:

N	V	—	B	D	I	Z	C
x						x	

Operation: An Exclusive OR (see 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, possibly affecting 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. Either 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 0. These flags may both be 0, or one of them may be 1; it is impossible for both to be set to 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 setting Z=1 if the result is 0. 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 6510 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 one after to highest byte of PC, causing an unconditional jump.

The indirect absolute jump, opcode $6C, is more elaborate and takes longer. PCL and PCH are loaded from the contents of two consecutive locations beginning at the address specified by the two bytes following the JMP opcode. This is the only absolute indirect command available on the 6510.

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 to jumps 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 ($0314) takes place. A look at this region of RAM with a monitor reveals something like this:

   ```
   0314 31 EA 97 FF 47 FE
   ```

   So JMP ($0314) is equivalent to JMP $EA31 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 6510 equivalent of a GOSUB, transferring control to another part of the program until an RTS is found, 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 0 (all bits loaded are 0's). 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 0. 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. Register 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 0. 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 generally 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.

$$\boxed{0} \rightarrow \boxed{7\ 6\ 5\ 4\ 3\ 2\ 1\ 0} \rightarrow \boxed{C}$$

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. Therefore, Z can 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 64'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

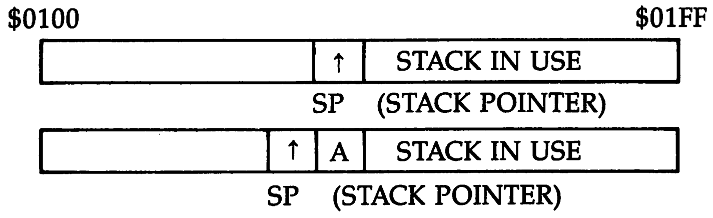
Instruction	Addressing	Bytes	Cycles
$48 (72 %0100 1000)	PHA implied	1	3

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 CHARACTER
     PHA        ;STORE ON STACK
     AND  #$7F  ;REMOVE BIT 7
     JSR  PRINT ;OUTPUT A CHARACTER
     PLA        ;RECOVER WITH BIT 7 INTACT
     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' states can be stored for use later.

PLA

Pull the stack into the accumulator. SP:= SP+1, A:= S

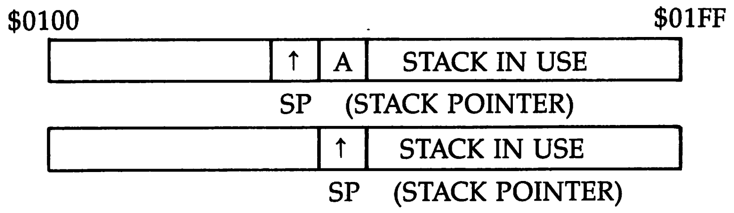
Instruction	Addressing	Bytes	Cycles
$68 (104 %0110 1000)	PLA implied	1	4

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. Figure 10-2 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

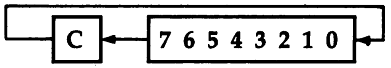
Flags:

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:

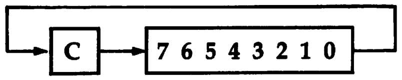
N	V	—	B	D	I	Z	C
x						x	x

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 64, 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 64's RAM IRQ vector into the user's own program at $C00D (where it may play a musical tone or whatever).

```
C000   SEI
C001   LDA  #$C0
C003   STA  $0315
C006   LDA  #$0D
C008   STA  $0314
C00B   CLI
C00C   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. On entry X 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:

$Y \rightleftharpoons A \rightleftharpoons X \rightleftharpoons S.$

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 64 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 onto 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

64 ROM Guide

- 64 Memory Map

Chapter 11

64 ROM Guide

64 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 Commodore 64 BASIC.

Locations are listed for both the Commodore 64 and the VIC, since many locations are the same on the two computers.

Commodore 64 BASIC is stored in ROM from $A000 to $BFFF. 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 a large number 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 much as possible.

A number of ROM routines are vectored through RAM; Chapter 8 explains how to take advantage of this.

Notation

Labels have been included as reference points, and where possible they refer back to well-known labels.

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 convert these. INT or FIX format is the simpler format with bytes in sequence.

A, X, and Y are the 6510/6502's registers. A/Y means the two-byte value 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 64 and VIC addresses; where one address is given, it applies to both computers unless otherwise noted, but where two are given, the 64 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	64/VIC	Descriptions
D6510	$00	6510 on-chip data direction register (Commodore 64 only).
R6510	$01	6510 on-chip input/output register (Commodore 64 only).
	$02	Unused byte (Commodore 64 only).
FACINT	$03-$04	Vector to routine to convert FAC to integer in A/Y (usually $B1AA).
INTFAC	$05-$06	Vector to routine to convert integer in A/Y to floating point in FAC (usually $B391).
CHARAC	$07	Delimiting character used when scanning. Also temporary integer (0–255) used during INT.
INTEGR	$07-$08	Intermediate integer used during OR/AND.
ENDCHR	$08	Delimiter used when scanning strings.
TRMPOS	$09	Temporary location used for calculating TAB and SPC column.
VERCHK	$0A	Flag to indicate LOAD (0) or VERIFY (1).
COUNT	$0B	Temporary pointer used with BASIC input buffer.
DIMFLG	$0C	Flag: default array dimension.
VALTYP	$0D	Flag: current variable data type; 0 means numeric, $FF means string.
INTFLG	$0E	Flag: current variable data type; 0 means floating point; $80 means integer.
GARBLF	$0F	Flag used in garbage collection, LIST, DATA, error messages.
SUBFLG	$10	Flag to indicate integers or array elements, which are forbidden as indexes of FOR/NEXT loops and in function definitions.
INPFLG	$11	Flag used by READ routine; $00 means INPUT, $40 means GET, $98 means READ.
TANSGN	$12	Sign byte used by TAN, SIN. Also set according to any comparison being performed: > sets this location to $01, = sets $02, and < sets $04.
CHANNL	$13	Current I/O device number; prompts suppressed if not 0.
LINNUM	$14-$15	Line number integer (0–63999) or standard two-byte address used by GOTO, GOSUB, POKE, PEEK, WAIT, and SYS.
TEMPPT	$16	Index to next entry on string descriptor stack (may be $19, $1C, $1F, or $22).
LASTPT	$17-$18	Pointer to current entry on string descriptor stack.
TEMPST	$19-$21	Stack for three temporary string descriptors.
INDEX1	$22-$23	General-purpose pointer, for example, for memory moves.
INDEX2	$24-$25	General-purpose pointer, for example, for number movements.
RESHO	$26-$2A	Floating point workspace used by multiply and divide.
TXTTAB	$2B-$2C	Pointer to first byte of BASIC program (2049 for the 64).
VARTAB	$2D-$2E	Pointer to start of program variables; first byte beyond end of program.
ARYTAB	$2F-$30	Pointer to start of arrays; first byte beyond end of variables.
STREND	$31-$32	Pointer to start of free RAM available for strings; first byte beyond end of arrays.
FRETOP	$33-$34	Pointer to current lower boundary of string area. (Set to the contents of MEMSIZ on CLR or RUN.)
FRESPC	$35-$36	Utility pointer used when new string is being added to string area.
MEMSIZ	$37-$38	Pointer to one byte beyond the top of RAM available to BASIC.
CURLIN	$39-$3A	BASIC line number being interpreted ($FF in $003A indicates immediate mode).

OLDLIN	**$3B–$3C**	If STOP, END, or BREAK occurs, this holds the last BASIC line number executed for CONT.
OLDTXT	**$3D–$3E**	Pointer to beginning of current BASIC line for CONT.
DATLIN	**$3F–$40**	Line number of current DATA statement. Initialized to $0000 on RUN.
DATPTR	**$41–$42**	Pointer to one byte beyond the DATA item read by the last READ statement. Initialized to contents of TXTTAB on RUN.
INPTR	**$43–$44**	Temporary storage of DATPTR during READ statement; also pointer within input buffer during INPUT (points to last character entered).
VARNAM	**$45–$46**	Current BASIC variable; two-character name with most significant bit (bit 7) of each byte used to indicate variable type: bit 7 clear in both bytes means floating point, bit 7 set in both means integer, bit 7 set in $46 means string, bit 7 set in $45 means function.
VARPNT	**$47–$48**	Pointer to current variable's address in RAM. Points one byte beyond variable name.
FORPNT	**$49–$4A**	Temporary pointer to variables in memory for INPUT, assignments, etc., and for loop variable in FOR/NEXT loops. Also holds the two parameters for WAIT statements.
OPPTR	**$4B**	Pointer within operator table during expression evaluation in routine FRMEVL.
OPMASK	**$4D**	Comparison mask used in FRMEVL: > sets this location to $01, = sets $02, and < sets $04.
DEFPNT	**$4E–$4F**	Pointer to variable in function definition, within variable table in RAM. Also used by garbage collection routine GARBAG.
TEMPF3	**$4E–$52**	Temporary storage for a MFLPT item.
DSCPNT	**$50–$51**	Pointer to descriptor in variable list or to string in dynamic string area; used during string operations.
SIZE	**$52**	Length of the current BASIC string.
FOUR6	**$53**	Length of string variable during garbage collection.
JMPER	**$54–$56**	Jump vector for function evaluations, JMP ($4C) followed by function address from function vector table.
TEMPF1	**$57–$5B**	Temporary pointers (for example, in memory move); also temporary floating point accumulator.
HIGHDS	**$58–$59**	Pointer used by block transfer routine BLTU.
ARYPNT	**$58–$59**	Pointer used when initializing arrays (when DIM is encountered).
HIGHTR	**$5A–$5B**	Pointer used by block transfer routine BLTU.
TEMPF2	**$5C–$60**	Temporary floating point accumulator.
DECCNT	**$5D**	Number of digits after/before decimal point in ASCII-to-FLPT and FLPT-to-ASCII conversion for the FIN and FOUT routines.
TENEXP	**$5E**	Exponent used in ASCII-to-FLPT and FLPT-to-ASCII conversion in the FIN and FOUTroutines.
DPTFLG	**$5F**	Flag used by the FIN routine ($BCF3) when inputting numbers; set to $80 if string contains decimal point.
LINPTR	**$5F–$60**	Pointer used when searching for line numbers, searching for variables in variable list, doing block transfers.
EXPSGN	**$60**	Sign of exponent of number being input by FIN routine; a value of $80 signifies negative.
FAC1	**$61–$66**	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 are stored in two bytes FAC1+3 and FAC1+4.

SGNFLG	$67	Flag used by FIN when inputting numbers; set to $FF if the number is negative. Also stores count of terms in polynomial series when evaluating trig functions.
BITS	$68	Bit overflow area on normalizing FAC1.
FAC2	$69-$6E	Floating Point Accumulator 2; used with FAC1 in evaluation of products, sums, differences, etc.
ARISGN	$6F	Sign comparison between FAC1 and FAC2; $00 means same sign, $FF means opposite.
FACOV	$70	Rounding/overflow byte for FAC1.
TEMPTX	$71-$72	General pointer used in CRUNCH, VAL, series evaluation, with tape buffer, etc.
CHRGET	$73-$8A	Subroutine to 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	$79	Entry point within CHRGET to re-get current BASIC character and set flags as CHRGET does. Does not increment TXTPTR first.
TXTPTR	$7A-$7B	Pointer into BASIC text used by CHRGET and CHRGOT routines.
RNDX	$8B-$8F	Floating point random number seed and subsequent pseudo-random values.
STATUS	$90	Status ST for serial devices and cassette.
STOPFL	$91	Flag: contains $7F (127) if RUN/STOP key pressed.
TSERVO	$92	Tape timing constant.
VERCK	$93	Flag to indicate LOAD (0) or VERIFY (1).
ICHRFL	$94	Serial flag: a value of $FF indicates a character is awaiting output.
IDATO	$95	Serial character to be output; a value of $FF indicates no character.
TEOB	$96	Flag: end of data block from tape.
TEMPXY	$97	Temporary X,Y storage during cassette read/RS-232 input.
NFILES	$98	Number of files open (maximum of ten); index to file table.
DFLTI	$99	Current input device number; default value is 0 (keyboard).
DFLTO	$9A	Current output device number; default value is 3 (screen).
TPARIT	$9B	Parity of byte written to tape.
TBYTFL	$9C	Flag: byte read from tape is complete.
MSGFLG	$9D	Flag: $00 means program mode; $80 means direct mode.
HDRTYP	$9E	Tape buffer header ID.
PTR1	$9E	Cassette pass 1 read errors.
PTR2	$9F	Cassette pass 2 read errors.
TIME	$A0-$A2	Three-byte jiffy clock for TI, updated 60 times per second. Bytes arranged in order of decreasing significance.
TSFCNT	$A3	Tape read/write bit counter.
TBTCNT	$A4	Tape read/write pulse counter.
CNTDN	$A5	Tape synchronization write countdown.
BUFPNT	$A6	Count of bytes in tape I/O buffer.
INBIT	$A7	RS-232 temporary storage for received bits.
PASNUM	$A7	General temporary store for cassette read/write.
BITCI	$A8	RS-232 received bit count. Also temporary store for cassette read/write.
RINONE	$A9	RS-232 receive: check for start bit.
TBITER	$A9	Write start bit/read bit sequence error.
RIDATA	$AA	Tape read mode; 0 means scan, 1-15 means count, $40 means LOAD, $80 means end-of-tape marker.
RIDATA	$AA	RS-232 received byte buffer.

TCKS	**$AB**	Counter of seconds before tape write. Also checksum.
RPRTY	**$AB**	RS-232 received byte parity.
SAL	**$AC-$AD**	Start address for LOAD/SAVE. Pointer also used by screen scrolling and INSert routines.
EAL	**$AE-$AF**	End address for LOAD/SAVE. Also used as pointer to color RAM used by the INSert routine.
CMP0	**$B0-$B1**	Timing constants for tape.
TAPE1	**$B2-$B3**	Pointer to start of cassette buffer, usually $033C.
BITTS	**$B4**	RS-232 transmit bit count.
TTIX	**$B4**	Tape read timer flag.
NXTBIT	**$B5**	RS-232 transmit: next bit to send.
TEOT	**$B5**	End of tape read.
RODATA	**$B6**	RS-232 transmit: byte to be sent.
TERRR	**$B6**	Tape read error flag.
FNLEN	**$B7**	Number of characters in filename; a value of 0 means no name.
LA	**$B8**	Current logical file number.
SA	**$B9**	Current secondary address.
FA	**$BA**	Current device number; for example, 3 means screen, 4 means printer, etc.
FNADR	**$BB-$BC**	Pointer to start of current filename.
ROPRTY	**$BD**	RS-232 output parity.
TCHR	**$BD**	Byte to be written to/read from tape.
FSBLK	**$BE**	Number of blocks remaining to read/write.
MYCH	**$BF**	Serial word buffer where byte is assembled.
CAS1	**$C0**	Cassette motor control flag.
STAL	**$C1-$C2**	Start address for LOAD and cassette write.
MEMUSS	**$C3-$C4**	Pointer for general use, for example, calculating LOAD address.
LSTX	**$C5**	Matrix value of key pressed during last keyboard scan; a value of $40 means no key pressed.
NDX	**$C6**	Number of characters in keyboard buffer.
RVS	**$C7**	Flag: print reverse characters; 0 means normal, $12 means reverse.
INDX	**$C8**	Count of characters in line input from screen.
LXSP	**$C9**	Cursor Y value (row) at start of input.
LYSP	**$CA**	Cursor X value (column) at start of input.
KEYVAL	**$CB**	Copy of keypress LSTX checked by interrupt so that a held key registers only once.
BLNSW	**$CC**	Flag: cursor blink mode; a value of 0 means enabled, 1 means disabled.
BLNCT	**$CD**	Countdown to next cursor toggle (from $14).
GDBLN	**$CE**	Character (screen code) at cursor position.
BLNON	**$CF**	Flag: 1 means cursor in blink phase, 0 means not in blink phase.
CRSW	**$D0**	Flag: 3 means input from screen, 0 means input from keyboard.
PNT	**$D1-$D2**	Address of start of current line on the screen.
PNTR	**$D3**	Cursor position (X value) along current logical line (0–$4F).
QTSW	**$D4**	Quote mode flag: flips each time quotes are encountered; 0 means move cursor, etc.; 1 means print reverse characters.
LNMX	**$D5**	Length of current logical screen line.
TBLX	**$D6**	Row of cursor.
TMPD7	**$D7**	CHR$ value of last character input/output to screen; tape temporary I/O storage and checksum.
INSRT	**$D8**	Number of keyboard inserts outstanding.

LDTB1	$D9-$F2	Table of 25 high bytes of pointers to the start of screen lines in RAM. (The low bytes are held in ROM from $ECF0.) Lines with wraparound have bit 7 set to 0; otherwise, bit 7 is 1.
USER	$F3-$F4	Pointer to byte in color RAM corresponding to beginning of current line on the screen.
KEYTAB	$F5-$F6	Address of current keyboard decoding table.
RIBUF	$F7-$F8	RS-232: pointer to start of receive buffer.
ROBUF	$F9-$FA	RS-232: pointer to start of transmit buffer.
	$FB-$FE	Unused; available for user programs.
BASZPT	$FF	Temporary storage area for FLPT-to-ASCII conversion.

Page 1 (Stack Area): RAM $0100-$01FF

Label	64/VIC	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	64/VIC	Descriptions
BUF	$200-$258	System input buffer; all keyboard input is read into here.
LAT	$259-$262	Table of up to ten active logical file numbers.
FAT	$263-$26C	Table of up to ten corresponding device numbers.
SAT	$26D-$276	Table of ten corresponding secondary addresses as used by system.
KEYD	$277-$280	Keyboard buffer: maximum of ten characters are read from keyboard and placed here by the interrupt routine.
LORAM	$281-$282	Pointer to lowest available byte of RAM for BASIC program storage (initialized on power-up; normally 2048).
HIRAM	$283-$284	Pointer to highest available BASIC RAM byte (initialized on power-up).
TIMOUT	$285	Serial bus time-out 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; a value of $00 means repeat cursor move and space keys; $80 means repeat all keys; $40 means no keys repeat. Default is $00.
KOUNT	$28B	Delay before repeat operates (system resets this).
DELAY	$28C	Delay between repeats.
SHFLAG	$28D	Detect SHIFT, Commodore key, CTRL keypress: a value of $01 means SHIFT is pressed, $02 means Commodore key, $04 means CTRL. These are additive: $05 means SHIFT and CTRL keys are both pressed, etc.
LSTSHF	$28E	Last SHFLAG pattern; used for debouncing.

KEYLOG	**$28F–$290**	Vector to routine to check SHIFT pattern; used by SCNKEY Kernal routine.
MODE	**$291**	Flag: $00 means enable upper/lowercase toggle using SHIFT and Commodore key; $80 means disable the toggle.
AUTODN	**$292**	Flag: autoscroll down during input; $00 means 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 storage for IRQ vector during tape operations.
	$2A1–$2A5	Temporary storage during tape operations.
	$2A6	PAL/NTSC Flag (0 means NTSC, 1 means PAL)
	$2A7–$2FF	Free RAM available to user

Page 3: RAM $0300–$03FF

Label	**64/VIC**	**Descriptions**
IERROR	**$300–$301**	Vector to BASIC print error message (normally $E38B); X register holds error message number.
IMAIN	**$302–$303**	Vector to routine to input or execute line of BASIC (normally $A483).
ICRNCH	**$304–$305**	Vector to BASIC tokenizing routine (normally $A57C).
IQPLOP	**$306–$307**	Vector to BASIC LIST routine (normally $A71A).
IGONE	**$308–$309**	Vector to BASIC RUN routine (normally $A7E4).
IEVAL	**$30A–$30B**	Vector to BASIC single-expression evaluation routine (normally $AE86).
SAREG	**$30C**	6510/6502 Accumulator storage for SYS; A is loaded from this location on SYS call and stored back into it when the SYS call ends.
SXREG	**$30D**	6510/6502 X register storage for SYS; handling as above.
SYREG	**$30E**	6510/6502 Y register storage for SYS; handling as above.
SPREG	**$30F**	6510/6502 Status register storage for SYS; handling as above.
USRPOK	**$310/$00**	USR function JMP instruction ($4C).
USRADD	**$311–$312/ $01–$02**	USR function address, low/high byte form, initialized to point to BASIC error message routine ($B284).
	$313	Unused byte.

(Note that the vectors from $314–$333 are initialized each time RUN/STOP–RESTORE is pressed, assuming NMINV is normal.)

CINV	**$314–$315**	Vector for IRQ interrupt (normally $EA31). Called from $FF58.
CBINV	**$316–$317**	Vector for BRK (normally $FE66). Called from $FF55.
NMINV	**$318–$319**	Vector for NMI (normally $FE47).

IOPEN	**$31A-$31B**	Vector to Kernal OPEN routine (normally $F34A). Called from $FFC0.
ICLOSE	**$31C-$31D**	Vector to Kernal CLOSE routine (normally $F291). Called from $FFC3.
ICHKIN	**$31E-$31F**	Vector to Kernal CHKIN routine (normally $F20E). Called from $FFC6.
ICKOUT	**$320-$321**	Vector to Kernal CHKOUT routine (normally $F250). Called from $FFC9.
ICLRCH	**$322-$323**	Vector to Kernal CLRCHN routine (normally $F333). Called from $FFCC.
IBASIN	**$324-$325**	Vector to Kernal CHRIN routine (normally $F157). Called from $FFCF.
IBSOUT	**$326-$327**	Vector to Kernal CHROUT routine (normally $F1CA). Called from $FFD2.
ISTOP	**$328-$329**	Vector to Kernal STOP routine (normally $F6ED). Called from $FFE1.
IGETIN	**$32A-$32B**	Vector to Kernal GETIN routine (normally $F13E). Called from $FFE4.
ICLALL	**$32C-$32D**	Vector to Kernal CLALL routine (normally $F32F). Called from $FFE7.
USRCMD	**$32E-$32F**	Unused vector: May be defined by user; initialized to BRK vector ($FE66).
ILOAD	**$330-$331**	Vector to Kernal LOAD routine (normally $F4A5).
ISAVE	**$332-$333**	Vector to Kernal SAVE routine (normally $F5ED).
	$334-$33B	Eight unused bytes.
TBUFFR	**$33C-$3FB**	Tape I/O buffer (192 bytes long). Can be used for ML programs but tape use will overwrite.
	$3FC-$3FF	Four unused bytes.

Note: The following summary of the memory map applies to the 64 only, since there are considerable hardware differences between the 64 and other CBM machines.

In the Commodore 64, locations $8000 and above are subject to memory management by both hardware (EXROM and GAME lines) and software (locations 0 and 1), and therefore can contain different things at different times. All this is explained in Chapter 5. Note that a plug-in cartridge is assumed to be ROM in what follows.

Hex	**Decimal**	**Description**
$0400-$07E7	**1024-2023**	Usual screen memory area: 25 lines with 40 columns each.
$07F8-$07FF	**2040-2047**	Pointers to sprite data blocks (assuming screen starts at $0400).
$0800-$9FFF	**2048-40959**	Space normally occupied by BASIC programs and associated variables, arrays, and strings.
$8000-$9FFF	**32768-40959**	RAM or cartridge ROM (usually with autorun feature).
$A000-$BFFF	**40960-49151**	BASIC ROM or RAM or cartridge ROM (may have autostart).
$C000-$CFFF	**49152-53247**	RAM.
$D000-$DFFF	**53248-57343**	I/O chips and color RAM or cartridge ROM (without autostart).

The region $D000–$DFFF is configured as follows (note that I/O chips have repeat images):

$D000-$D02E	**53248-53294**	VIC chip (see Chapter 12).
$D400-$D41C	**54272-54300**	SID chip (see Chapter 13).

$D800-$DBE7	**55296-56295**	Color RAM (low nybbles store character colors 0–15).
$DC00-$DC0F	**56320-56335**	CIA 1 (see Chapter 5).
$DD00-$DD0F	**56576-56591**	CIA 2 (see Chapter 5).

This region also includes the character generator ROM:

$D000-$D7FF	**53248-55295**	Uppercase/graphics character set.
$D800-$DFFF	**55296-57343**	Lower/uppercase character set.

BASIC and Kernal ROM

Commodore 64 and VIC-20 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	64/VIC	Descriptions
BCOLD	$A000/$C000	BASIC cold start vector ($E394). NEWs BASIC, prints BYTES FREE and READY. Part of the reset sequence; see routines at $E394 and $FCE2.
BWARM	$A002/$C002	BASIC warm start vector ($E37B). CLRs BASIC, prints READY. Part of the NMI sequence; see routines at $E37B and $FE43.
	$A004/$C004	CBM BASIC message.
	$A00C/$C00C	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.)
	$A052/$C052	Table of true addresses of routines for handling numeric and string functions (FRE, POS, SQR, etc.).
	$A080/$C080	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.
	$A09E/$C09E	BASIC keywords as CBM ASCII strings with bit 7 of final character of each keyword set high.
	$A129/$C129	Table of miscellaneous keywords (TAB, STEP, etc., with no action address) with bit 7 of final character of each keyword set high.
	$A140/$C140	Table of operator tokens; also AND, OR as strings with bit 7 of final character of each operator set high.
	$A14D/$C14D	Table of function keywords (SGN, INT, ABS, etc.) with bit 7 of final character of each keyword set high.
	$A19E/$C19E	Table of 28 error messages (TOO MANY FILES, FILE OPEN, etc.) with bit 7 of final character of each message set high.
	$A328/$C328	Table of pointers to error messages.
	$A364/$C364	Table of other messages: OK, ERROR IN, READY, BREAK.
FNDFOR	$A38A/$C38A	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	$A3B8/$C3B8	Open up a gap in BASIC text to allow insertion of new BASIC line. Check whether there is enough room.

BLTUC	**$A3BF/$C3BF**	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–$59 − 1.
GETSTK	**$A3FB/$C3FB**	Test to see whether stack will accommodate A*2 bytes: ?OUT OF MEMORY if not.
REASON	**$A408/$C408**	Check whether address pointed to by A/Y is below FRETOP (current bottom of string area). If yes, exit; otherwise, do garbage collection and check again. If still not, then print ?OUT OF MEMORY.
ERROR	**$A437/$C437**	Print error message; X holds error number (half of offset within error message address table). Vectored via ($0300) to $E38B. Then set keyboard input and screen output, reset stack, and print IN with line number if in program mode.
READY	**$A474/$C474**	Restart BASIC; print READY, set direct mode.
MAIN	**$A480/$C480**	Receive a line into input buffer and add a terminating zero byte. Check for program line or immediate mode command; if immediate mode command, execute it. MAIN is vectored via ($0302) to $A483.
MAIN1	**$A49C/$C49C**	If program line, tokenize it.
INSLIN	**$A4A4/$C4A4**	If the line number already exists, replace it. If it's new, insert it. Line number is in $14–$15 on entry, length + 4 is in Y. If the first byte in buffer is 0, the line is null; delete it.
FINI	**$A52A/$C52A**	Having inserted a new line, do RUNC (thus, variables are lost on editing, and you cannot CONT after editing) and LNKPRG; then jump to MAIN.
LNKPRG	**$A533/$C533**	Chain link pointers in BASIC program using end-of-line zero markers.
INLIN	**$A560/$C560**	Input a screen line into the BASIC text buffer at $200, and add a zero terminating byte.
CRUNCH	**$A579/$C579**	Tokenize keywords in input buffer. Vectored via ($0304) to $A57C.
FNDLIN	**$A613/$C613**	Search BASIC text from beginning for line number in $14–$15. Carry bit set if line found. Locations $5F–$60 point to link address.
FNDLNC	**$A617/$C617**	Search BASIC text from address in A (low byte) and X (high byte) for line number in $14–$15.
NEW	**$A642/$C642**	NEW routine enters here; check syntax, and continue with SCRTCH.
SCRTCH	**$A644/$C644**	Reset first two bytes of text (first link pointer) to 0; load start-of-variables pointer $2D–$2E with start-of-BASIC + 2, and continue with RUNC.
RUNC	**$A659/$C659**	Set pointer within CHRGET to start of BASIC text, using STXPT, then continue with CLEAR.
CLEAR	**$A65E/$C65E**	BASIC CLR routine; erase variables by resetting end-of-variables pointers to coincide with end-of-program pointer; appropriate string variable pointers are also reset. Abort I/O activity and reset stack.
STXPT	**$A68E/$C68E**	Reset pointer within CHRGET routine to beginning of BASIC text ($2B–$2C − 1 is loaded into $7A–$7B).
LIST	**$A69C/$C69C**	Entry point of routine to process LIST command.
LIST1	**$A6C9/$C6C9**	List one line of BASIC; line number, then text.

QPLOP	**$A717/$C717**	Handle character to be listed; if ordinary character or control character in quotes, print it; expand and print tokens. Vectored via ($0306) to $A71A.
FOR	**$A742/$C742**	Entry point for routine to handle FOR statement. Push 18 bytes onto stack: pointer to following statement, current line number, upper-loop value, step value (defaults to 1), loop variable name, and FOR token.
NEWSTT	**$A7AE/$C7AE**	Execute BASIC; test for RUN/STOP key and check for end-of-line zero byte or colon.
CKEDL	**$A7C4/$C7C4**	If at end of text, stop; otherwise, set pointer within CHRGET to beginning of next line.
GONE	**$A7E1/$C7E1**	Handle the BASIC statement in the current line. Vectored via ($0308) to $A7E4, loop back to NEWSTT.
EXCC	**$A7ED/$C7ED**	Execute a BASIC keyword. Uses address for start of routine from table at $A00C. Assumes LET if a token is not the first byte in the statement. Address pushed on stack so RTS of GETCHR jumps to it.
RESTOR	**$A81D/$C81D**	Entry point for routine to handle RESTORE; set the data pointer at $41–$42 to start of BASIC text.
STOP	**$A82C/$C82C**	Entry point for routine to handle STOP; also END and break in program. Information for CONT (pointer in BASIC text, line number) is stored. STOP prints BREAK IN *nnn* while END skips this to READY. The RUN/STOP key invokes STOP. Reaching the end-of-BASIC program text calls END.
CONT	**$A857/$C857**	Entry point for routine to handle CONT; performs this by setting current line number (stored in $39–$3A) and the pointer within CHRGET to values stored by STOP. ?CANNOT CONTINUE ERROR occurs if the high byte of the pointer has been set to 0 on syntax error.
RUN	**$A871/$C871**	Entry point for routine to handle RUN; if RUN is encountered alone, then CLR variables and reset stack, set CHRGET to start of BASIC, and begin execution. If RUN *nnn*, CLR variables and reset stack, then do GOTO *nnn*.
GOSUB	**$A883/$C883**	Entry point for routine to handle GOSUB; push five bytes onto stack: pointer within CHRGET (two bytes), current line number (two bytes), and the GOSUB token. The GOTO routine is then called.
GOTO	**$A8A0/$C8A0**	Entry point for routine to handle GOTO; fetch the line number following the GOTO command and search BASIC text for this line. If high byte of destination is higher than high byte of current line number, search from position of current line onward to shorten search time; otherwise, search from beginning. Put pointer to found line into CHRGET pointer.
RETURN	**$A8D2/$C8D2**	Entry point for routine to handle RETURN; stack is cleared up to GOSUB token (?RETURN WITHOUT GOSUB if not found); then the calling line number and pointer are reinstated, and execution continues.
DATA	**$A8F8/$C8F8**	Entry point for routine to handle DATA statements; routine to let CHRGET skip DATA statement up to terminating byte or colon.
DATAN	**$A906/$C906**	Search for statement terminator; exits with Y containing displacement to end of line from CHRGET's pointer.

REMN	$A909/$C909	Search for end-of-BASIC line.
IF	$A928/$C928	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	$A93B/$C93B	Entry point for routine to handle REM; scan for end of line and update pointer in CHRGET, to ignore contents of REM statement.
DOCOND	$A940/$C940	Continue IF; if expression true, then execute next command, or do GOTO if digit follows.
ONGOTO	$A94B/$C94B	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	$A96B/$C96B	Read an integer (usually a line number) from the BASIC text into locations $14 and $15; must be in range 0–63999.
LET	$A9A5/$C9A5	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	$A9C4/$C9C4	Round FAC1 and put, as integer, into variable list at current variable position, pointed to by $49–$4A.
PTFLPT	$A9D6/$C9D6	Put FAC1 into variable list at location pointed to by $49–$4A.
PUTTIM	$A9E3/$C9E3	Assign the system variable TI$.
ASCADD	$AA27/$CA27	Add ASCII digit to FAC1.
GETSPT	$AA2C/$CA2C	LET for strings; put string descriptor pointed to by FAC1+3–FAC1+4 into variable list at location pointed to by $49–$4A.
PRINTN	$AA80/$CA80	Entry point for routine to handle PRINT# statement; call CMD, then clear I/O channels and restore default I/O device numbers.
CMD	$AA86/$CA86	Entry point for routine to handle CMD; set output device from file table using Kernal CHKOUT routine, then call PRINT.
STRDON	$AA9A/$CA9A	Part of PRINT routine; print string and continue with punctuation of PRINT.
PRINT	$AAA0/$CAA0	Entry point for routine to handle PRINT statement; identify PRINT parameters (TAB, SPC, comma, semicolon, etc.), and evaluate expression.
VAROP	$AAB8/$CAB8	Print variable; if numeral, convert to string before printing.
CRDO	$AAD7/$CAD7	Print carriage return (ASCII 13) followed (if channel > 128) by linefeed (ASCII 10).
STROUT	$AB1E/$CB1E	Print string beginning at address specified in A/Y, and terminated by a zero byte or quotes.
STRPRT	$AB21/$CB21	Print string; FAC1+3–FAC1+4 points to string descriptor.
OUTSTR	$AB24/$CB24	Output string; locations $22–$23 point to string, length in A.
OUTSPC	$AB3B/$CB3B	Output cursor-right (or space if the screen is not the current output device).
PRTSPC	$AB3F/$CB3F	Output space.
OUTSKP	$AB42/$CB42	Output cursor-right.
OUTQST	$AB45/$CB45	Output question mark for error messages.
OUTDO	$AB47/$CB47	Output the character in A.
TRMNOK	$AB4D/$CB4D	Output appropriate error messages for GET, READ, and INPUT.

GET	**$AB7B/$CB7B**	Entry point for routine to handle GET and GET# statements; test for direct mode (illegal) and fetch one character from keyboard or file.
INPUTN	**$ABA5/$CBA5**	Entry point for routine to handle INPUT# statement; fetch file number, turn the device on, call INPUT, and then turn the device off.
INPUT	**$ABBF/$CBBF**	Entry point for routine to handle INPUT statement; output user's prompt string if present, then continue with QINLIN routine.
QINLIN	**$ABF9/$CBF9**	Print ? prompt and receive line of text (terminated by RETURN) into input buffer.
READ	**$AC06/$CC06**	Entry point for routine to handle the READ statement. GET and INPUT also share this routine, but are distinguished by a flag in location $11.
INPCON	**$AC0D/$CC0D**	Entry point into READ routine for INPUT; set flag and call READ, with buffer at the address specified in X (low byte) and Y (high byte).
INPCO1	**$AC0F/$CC0F**	Entry point into READ routine for GET; set flag and call READ, with buffer at the address specified in X (low byte) and Y (high byte).
DATLOP	**$ACB8/$CCB8**	Scan text and read DATA statements.
VAREND	**$ACDF/$CCDF**	Tests for 0 at end of input buffer; if not found, print ?EXTRA IGNORED.
EXINT	**$ACFC/$CCFC**	Messages ?EXTRA IGNORED and ?REDO FROM START.
NEXT	**$AD1E/$CD1E**	Entry point for routine to handle NEXT; check for FOR token and matching variable on stack, and print ?NEXT WITHOUT FOR if not found; calculate next value. If the loop increment is still valid, reset current line number and the pointer in CHRGET and continue.
FRMNUM	**$AD8A/$CD8A**	Evaluate a numeric expression for BASIC by calling FRMEVL, then CHKNUM.
CHKNUM	**$AD8D/$CD8D**	Check that FRMEVL has returned a number by testing flag at location $0D. If a number was not returned, issue a ?TYPE MISMATCH ERROR message.
CHKSTR	**$AD8F/$CD8F**	Check that FRMEVL has returned a string by testing flag at location $0D. If a string was not returned, issue a ?TYPE MISMATCH ERROR message.
FRMEVL	**$AD9E/$CD9E**	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 $80, but leave the result in floating point format in FAC1. Complicated expressions may need simplifying to retain stack space and prevent ?OUT OF MEMORY.
EVAL	**$AE83/$CE83**	Evaluate a single term in an expression; look for ASCII numeral strings, variables, pi, NOT, arithmetic functions, etc.
PIVAL	**$AEA8/$CEA8**	Value of pi in five-byte floating point format.
PARCHK	**$AEF1/$CEF1**	Evaluate expression within parentheses.
CHKCLS	**$AEF7/$CEF7**	Check whether CHRGET points to a) character; issue a ?SYNTAX ERROR message if not.

CHKOPN	**$AEFA/$CEFA**	Check whether CHRGET points to a (character; issue a ?SYNTAX ERROR message if not.
CHKCOM	**$AEFD/$CEFD**	Check whether CHRGET points to a comma; issue a ?SYNTAX ERROR message if not.
SYNCHR	**$AEFF/$CEFF**	Check whether CHRGET points to a byte identical to that in A; if it does, routine exits with next byte in A; otherwise, a ?SYNTAX ERROR message is issued.
SYNERR	**$AF08/$CF08**	Output a ?SYNTAX ERROR message and return to READY.
DOMIN	**$AF0D/$CF0D**	Evaluate NOT.
TSTROM	**$AF14/$CF14**	Set carry flag to 1 if FAC1+3–FAC1+4 point to the ROM area indicating reserved variables TI$, TI, ST.
ISVAR	**$AF28/$CF28**	Search variable list for variable named in locations $45–$46; on exit FAC1 will hold numeric value in FLPT format (whether integer or floating point variable); FAC1+3–FAC1+4 will point to the descriptor if it's a string variable.
TISASC	**$AF48/$CF48**	Read clock and set up string containing TI$.
ISFUN	**$AFA7/$CFA7**	Identify function type and evaluate it.
OROP	**$AFE6/$CFE6**	Entry point for routine to handle the OR function; set flag and do OR between two two-byte integers in FAC1 and FAC2.
ANDOP	**$AFE9/$CFE9**	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, $00 for AND. Convert FLPT to integer (and give an error message if the result is out of range). The result in FLPT format is left in FAC1.
DOREL	**$B016/$D016**	Entry point for routine to handle string and numeric comparisons (< = >). Check variable types, then continue with NUMREL or STRREL, as appropriate.
NUMREL	**$B01B/$D01B**	Perform numeric comparison, using FCOMP at $BC5B.
STRREL	**$B02E/$D02E**	Perform string comparison; exit with X holding $00 if strings equal, $01 if the first string is greater than the second, and $FF if the second is greater than the first.
DIM	**$B081/$D081**	Entry point for routine to handle the DIM statement; set up each array element using the PTRGET routine.
PTRGET	**$B08B/$D08B**	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–$46) with high bits set to indicate the variable type, as described in Chapter 5.
ORDVAR	**$B0E7/$D0E7**	Search variable list for variable whose name is in VARNAM (locations $45–$46) and set VARPNT (locations $47–$48) to point to it. Create new variable if the name is not currently in the list.
ISLETC	**$B113/$D113**	Set the carry flag if the accumulator holds A–Z.
NOTFNS	**$B11D/$D11D**	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	**$B194/$D194**	Calculate pointer value in $5F–$60, to be used when setting up space for arrays.
N32768	**$B1A5/$D1A5**	Holds −32768 as a five-byte floating point number.
FACINX	**$B1AA/$D1AA**	Convert contents of FAC1 to two-byte integer (−32768 to +32767) in A/Y.
INTIDX	**$B1B2/$D1B2**	Fetch and evaluate a positive integer expression from the next part of BASIC text; if result is 0–32767, store in FAC1+3 and FAC1+4.
AYINT	**$B1BF/$D1BF**	Convert the contents of FAC1 to integer in range 0–32767; leave the result in FAC1+3–FAC1+4.
ISARY	**$B1D1/$D1D1**	Get array parameters from BASIC text (number of dimensions and number of elements) and push the values onto the stack.
FNDARY	**$B218/$D218**	Find array named in VARNAM ($45–$46), with other details of the array stored on the stack.
BSERR	**$B245/$D245**	?BAD SUBSCRIPT error. BSERR+3 will print ?ILLEGAL QUANTITY error message.
NOTFDD	**$B261/$D261**	If the specified array is not found, create it using details on stack with DIMension 10.
INPLN2	**$B30E/$D30E**	Locate specified element within array and point VARPNT ($47–$48) to it.
UMULT	**$B34C/$D34C**	Compute offset of specified array element relative to array pointed at by VARPNT ($47–$48); put in X/Y.
FRE	**$B37D/$D37D**	Entry point for routine to handle FRE function; perform garbage collection and set Y/A to point to lowest string minus pointer to end of arrays; then place in FAC1 and continue with GIVAYF.
GIVAYF	**$B391/$D391**	Convert two-byte integer in Y/A (range −32768 to +32767) to FLPT in FAC1.
POS	**$B39E/$D39E**	Entry point for routine to handle POS function; calls Kernal routine PLOT to fetch cursor position, then loads it into FAC1 using SNGET.
SNGET	**$B3A2/$D3A2**	Convert byte in Y to FLPT in FAC1 (0–255).
ERRDIR	**$B3A6/$D3A6**	Test that command was not entered in direct mode; CURLIN+1 ($3A) containing $FF indicates direct mode. ?ILLEGAL DIRECT ERROR if it was. Called by routines that may not be used in direct mode (for example, GET).
DEF	**$B3B3/$D3B3**	Entry point for routine to handle DEF statement; create function definition and find or set up dependent variable. When an FN is invoked, the pointer within CHRGET is set to the beginning of the FN definition in the BASIC text and the expression found there is evaluated; it is then switched back. Information to enable it to do this is stored within the function variable set up in GETFNM.
GETFNM	**$B3E1/$D3E1**	Check syntax of FN; find or set up variable with function name and set DEFPNT ($4E–$4F) to point to it (must be numeric, not string, variable).
FNDOER	**$B3F4/$D3F4**	Evaluate function; evaluate expression within parentheses in statement invoking function, leaving it in FAC1, then evaluate the FN expression (see DEF).
STRD	**$B465/$D465**	Entry point for routine to handle STR$ function; evaluate expression and convert to ASCII string.

STRINI	**$B475/$D475**	Make room in string space for a string to be inserted: A contains length and FAC1+3–FAC1+4 points to the string. On exit, $61–$63 contains descriptor for new string. CHR$, LEFT$, and so on all use this routine.
STRLIT	**$B487/$D487**	Copy a string into string space at top of memory; A/Y points to the start of the string. Scans for quotation mark ("), colon (:), or zero byte as terminator to determine length. Exit with descriptor in $61–$63.
GETSPA	**$B4F4/$D4F4**	Allocate space for string, length in A, in dynamic string space at top of memory; do garbage collection if space exhausted. Called by STRINI.
GARBA2	**$B526/$D526**	Do garbage collection; eliminate unwanted strings in string area and collect together valid strings. The garbage collection routine is slow for large numbers of strings.
DVARS	**$B606/$D606**	Search variables and arrays for next string to be saved by garbage collection.
CAT	**$B63D/$D63D**	Concatenate two strings.
MOVINS	**$B67A/$D67A**	Move string to string area high in RAM; entered with $6F–$70 pointing at the descriptor of the string to be stored.
FRESTR	**$B6A3/$D6A3**	Discard string; entered with pointer to string descriptor in FAC1+3–FAC1+4, exits with new string length and pointer in INDEX1.
FRETMS	**$B6DB/$D6DB**	Clean the descriptor stack.
CHRD	**$B6EC/$D6EC**	Entry point for routine to handle CHR$ function; sets up a one-byte string.
LEFTD	**$B700/$D700**	Entry point for routine to handle LEFT$.
RIGHTD	**$B72C/$D72C**	Entry point for routine to handle RIGHT$.
MIDD	**$B737/$D737**	Entry point for routine to handle MID$.
PREAM	**$B761/$D761**	Pull string descriptor pointer to $50–$51, length to A (also in X).
LEN	**$B77C/$D77C**	Entry point for routine to handle LEN function; floating point value of string length parameter placed in FAC1.
LEN1	**$B782/$D782**	Extract length of string, put in Y, leave string mode, and enter numeric mode. Called by LEN, VAL.
ASC	**$B78B/$D78B**	ASC function; get first character of string and convert to floating point in FAC1. String of length 0 gives ?SYNTAX ERROR.
GTBYTC	**$B79B/$D79B**	Read and evaluate an expression from BASIC text; must evaluate to a one-byte value; value left in X and FAC1+4.
VAL	**$B7AD/$D7AD**	Entry point for routine to handle VAL function; convert value to floating point value in FAC1.
GETNUM	**$B7EB/$D7EB**	Read parameters for WAIT and POKE from BASIC text; put first (two-byte integer) in $14–$15, second in X.
GETADR	**$B7F7/$D7F7**	Convert FAC1 to two-byte integer (range 0–65535) in $14–$15 and Y/A.
PEEK	**$B80D/$D80D**	Entry point for routine to handle PEEK function; on entry FAC1 contains address to be PEEKed in FLPT form; exit with PEEKed value in Y.
POKE	**$B824/$D824**	Entry point for routine to handle POKE statement; fetch two parameters from BASIC text; do POKE.
WAIT	**$B82D/$D82D**	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	**$B849/$D849**	Add 0.5 to contents of FAC1; used when rounding.
FSUB	**$B850/$D850**	Floating point subtraction; FAC1 is replaced by MFLPT value pointed to by A/Y, minus FAC1.
FSUBT	**$B853/$D853**	Entry point for routine to handle floating point subtraction; FAC1 is replaced by FAC2 minus FAC1.
FADD	**$B867/$D867**	Floating point addition; FAC1 is replaced by MFLPT value pointed to by A/Y, plus FAC1.
FADDT	**$B86F/$D86F**	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 $61) to speed the addition in the event that FAC1 contains 0.
COMPLT	**$B947/$D947**	Replace FAC1 with twos complement of the value currently there.
OVERR	**$B97E/$D97E**	Output ?OVERFLOW ERROR message, then READY.
MULSHF	**$B983/$D983**	Multiply by a byte.
FONE	**$B9BC/$D9BC**	Table of constants in MFLPT format: first 1, then constants for LOG evaluation; SQR(0.5), SQR(2), −0.5, and LOG(2).
LOG	**$B9EA/$D9EA**	Entry point for routine to handle the LOG function; compute logarithm to the base e of FAC1.
FMULT	**$BA28/$DA28**	Floating point multiply; FAC1 is replaced by MFLPT value pointed to by A/Y times FAC1.
FMULTT	**$BA30/$DA30**	Entry point for routine to handle floating point multiplication; FAC1 is replaced by FAC1 times FAC2.
MLTPLY	**$BA59/$DA59**	Multiply FAC1 by a byte and store in $26–$2A.
CONUPK	**$BA8C/$DA8C**	Load FAC2 from MFPLT value pointed to by A/Y, unpacking sign bit and storing it separately, forming FLPT format. On exit, A holds FAC1's first byte.
MULDIV	**$BAB7/$DAB7**	Test floating point accumulators for multiply and divide; if FAC2 is 0, set FAC1 to 0; if exponents together are too large then ?OVERFLOW ERROR. If they are too small, force the result to 0 without an underflow message.
MUL10	**$BAE2/$DAE2**	Multiply FAC1 by 10 and put result in FAC1.
TENC	**$BAF9/$DAF9**	The value 10 in MFLPT format.
DIV10	**$BAFE/$DAFE**	Divide FAC1 by 10 and put result in FAC1.
FDIVF	**$BB07/$DB07**	Floating point division; FAC1 is replaced by FAC2 divided by MFLPT value pointed at by A/Y; on entry, X contains sign of result.
FDIV	**$BB0F/$DB0F**	Floating point division; FAC1 is replaced by MFLPT divided by FAC1.
FDIVT	**$BB14/$DB14**	Entry point for routine to handle floating point division; FAC1 is replaced by FAC2 divided by FAC1. On entry, A holds FAC1's first byte.
MOVFM	**$BBA2/$DAB2**	Load FAC1 from MFLPT value pointed to by A/Y, unpacking sign bit and storing it separately, forming FLPT format.
MOV2F	**$BBC7/$DBC7**	Convert FAC1 to MFLPT format and store at $5C–$60, TEMPFP2.
MOV1F	**$BBCA/$DBCA**	Convert FAC1 to MFLPT format and store at $57–$5B, TEMPFP1.
MOVVF	**$BBD0/$DBD0**	Convert FAC1 to MFLPT format and store at address pointed to by $49–$4A.
MOVMF	**$BBD4/$DBD4**	Convert FAC1 to MFLPT format and store at address pointed to by A/Y .

MOVFA	**$BBFC/$DBFC**	Copy FAC2 into FAC1.
MOVAF	**$BC0C/$DC0C**	Round FAC1 by calling ROUND, then copy into FAC2.
ROUND	**$BC1B/$DC1B**	Round FAC1.
SIGN	**$BC2B/$DC2B**	Get sign of FAC1; on exit A holds 0 if value is 0, 1 if value is positive, or $FF if value is negative.
SGN	**$BC39/$DC39**	Entry point for routine to handle SGN function; calls SIGN, then converts A into floating point form in FAC1.
ABS	**$BC58/$DC58**	Entry point for routine to handle ABS function; replace FAC1 with the absolute value of the current contents of FAC1.
FCOMP	**$BC5B/$DC5B**	Compare FAC1 with MFLPT value pointed to by A/Y; on exit, A holds 0 if values were equal, 1 if FAC1>MFLPT, or $FF if FAC1<MFLPT.
QINT	**$BC9B/$DC9B**	Convert FAC1 to four-byte integer in FAC1+1–FAC1+4, highest byte first.
INT	**$BCCC/$DCCC**	Entry point for routine to handle INT function; round down FAC1 but leave it in FAC1 in FLPT form.
FIN	**$BCF3/$DCF3**	Convert an ASCII string (for example, "−99.375") to a floating point value in FAC1. On entry, TXTPTR points to the start of the string, then JSR GETCHR/JSR FIN accomplishes the conversion.
AADD	**$BD7E/$DD7E**	Add contents of A to FAC1.
STCONS	**$BDB3/$DDB3**	Three constants used in string conversions, in MFLPT form: 99999999.9, 999999999, and 1000000000.
INPRT	**$BDC2/$DDC2**	Print IN followed by current line number in CURLIN ($39–$3A).
LINPRT	**$BDCD/$DDCD**	Output integer in A/Y, range 0–65535.
FOUT	**$BDDD/$DDDD**	Convert contents of FAC1 to ASCII string starting at location $100 and ending with zero byte. On exit, A/Y holds start address, so STROUT can print string.
FOUTIM	**$BE68/$DE68**	Convert TI to ASCII string starting at $100 and ending with zero byte.
TICONS	**$BF11/$DF11**	String and TI conversion constants: 0.5 in MFLPT form, then 15 four-byte integer constants.
SQR	**$BF71/$DF71**	Entry point for routine to handle SQR function; FAC1 is replaced by square root of FAC1.
FPWRT	**$BF7B/$DF7B**	Entry point for routine to perform power calculation; FAC1 is replaced by FAC2 raised to the power of FAC1. On entry, A must hold contents of FAC2 so powers of 0 are correct.
NEGOP	**$BFB4/$DFB4**	Negate FAC1.
EXCONS	**$BFBF/$DFBF**	Table of eight constants for evaluating EXP series.
EXP	**$BFED/$DFED**	Entry point for routine to handle EXP function; FAC1 is replaced by e raised to FAC1.
POLYX	**$E059/$E056**	Series evaluation routine. Entered with A/Y pointing to the counter at the beginning of the table of constants used in the power series evaluation.
RMULC	**$E08D/$E08A**	The value 11879546.4 in MFLPT format; multiplicative constant for RND evaluation.
RADDC	**$E092/$E08F**	The value 3.92767778E−8 in MFLPT format; additive constant for RND evaluation.

RND	**$E097/$E094**	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	**$E09E/$E09B**	If FAC1=0, load FAC1 from VIA timer registers; a simple way of reseeding it with a random number.
QSETNR	**$E0BE/$E0BB**	If FAC1>0, load FAC1 with the result of multiplying the stored random number (in $88–$8C) generated by previous calls, by RMULC, and adding RADDC.
RND1	**$E0D3/$E0D0**	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–$8C.
RNDRNG	**$E0E3/$E0E0**	Force the value in FAC1 into the range 0–1 excluding 0 and 1.
BIOERR	**$E0F9/$E0F6**	I/O error message routine if any of the following calls return error flags:.
BCHOUT	**$E10C/$E109**	Output character; uses CHROUT.
BCHIN	**$E112/$E10F**	Input character; uses CHRIN.
BCKOUT	**$E118/$E115**	Set up for output; uses CHKOUT.
BCKIN	**$E11E/$E11B**	Set up for input; uses CHKIN.
BGETIN	**$E124/$E121**	Get one character; uses GETIN.
SYS	**$E12A/$E127**	Entry point for routine to handle SYS statement; load A, X, Y, and SR from locations $30C–$30F, call machine language routine at address specified by the argument, then reload the register contents into $30C–$30F on return from the routine.
SAVET	**$E156/$E153**	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 $FFD8.
VERFYT	**$E165/$E162**	Entry point for routine to handle VERIFY; set flag in A to indicate VERIFY operation, enter LOADT and check for errors.
LOADT	**$E168/$E165**	Entry point for routine to handle LOAD; fetch parameters from BASIC text and set them up, call Kernal routine LOAD via vector at $FFD5 .
LOADR	**$E16F/$E177**	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–$2E) to address of last byte loaded. This step is omitted if the routine is called from within a program, so variable list is preserved. Finally, reset pointer in CHRGET and warm start BASIC to run the new program.
OPENT	**$E1BE/$E1BB**	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 $FFC0.
CLOSET	**$E1C7/$E1C4**	Entry point for routine to handle CLOSE; read parameters from text and set them up; call Kernal CLOSE routine via vector at $FFC3.
SLPARA	**$E1D4/$E1D1**	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	**$E200/$E1FD**	Check for comma and evaluate the following one-byte parameter, which is put in X.
CMMERR	**$E20E/$E20B**	Check for comma followed by anything other than end of statement; otherwise, issue a ?SYNTAX ERROR message.
OCPARA	**$E219/$E216**	Get parameters from BASIC text for OPEN or CLOSE calls; set defaults if not supplied.
COS	**$E264/$E261**	Entry point for routine to handle the COS function; the value in FAC1 is replaced by the cosine of that value.
SIN	**$E26B/$E268**	Entry point for routine to handle the SIN function; the value in FAC1 is replaced by the sine of that value.
TAN	**$E2B4/$E2B1**	Entry point for routine to handle the TAN function; the value in FAC1 is replaced by the tangent of that value.
	$E2E0/$E2DD	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	**$E30E/$E30B**	Entry point for routine to handle ATN; the value in FAC1 is replaced by the arctangent of that value.
	$E33E/$E33B	A counter value (11) and table of 12 constants in MFLPT format for ATN evaluation.
BASSFT	**$E37B/$E467**	BASIC warm start routine, entered on JMP ($A002); part of the break sequence performed if BRK instruction encountered or RUN/STOP–RESTORE keys are pressed. Close all I/0 I/O channels, initialize stack, output ?BREAK ERROR, and jump to READY.
INIT	**$E394/$E378**	BASIC cold start routine, entered on JMP ($A000); part of the reset sequence. Performs INITV, INITCZ, INITMS; sets stack and jumps to READY.
CHRCPY	**$E3A2/$E387**	CHRGET routine and RND seed in ROM for relocation into RAM.
INITCZ	**$E3BF/$E3A4**	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–$285 initialized on power-up. Set end-of-program zero byte at 2048.
INITMS	**$E422/$E404**	Output start-up message: **** CBM BASIC V2 ****, then number of free bytes, then BYTES FREE.
INITV	**$E453/$E45B**	Initialize vectors for ERROR, MAIN, etc., at $0300–$030B.
CPATCH	**$E4DA**	Patch to diminish screen sparkle; called from $EA0B (used by CLR). Commodore 64 only.
IOBASK	**$E500/$E500**	Returns base address of CIA in X/Y (used by SCNKEY).
SCRENK	**$E505/$E505**	Returns screen columns (40) in X, lines (25) in Y.
PLOTK	**$E50A/$E50A**	Set/Read cursor row (X), column (Y).
CINT	**$E518/$E518**	General screen and VIC chip initialization; set up screen editing tables at $D9–$F2, initialize VIC chip, set character color to light blue, do CLR and HOME, reset default I/O device numbers at $99 and $9A.
HOME	**$E566/$E581**	Home the cursor.
INITVC	**$E5A0/$E5C3**	Initialize the VIC chip from table of values at $ECB9–$ECE6 (international variations).

GETKBC	**$E5B4/$E5CF**	Get character from keyboard queue and move remaining characters along; queue must contain at least one character on entry (number of characters in queue is stored in $C6). On exit, the character is in A.
INPPRO	**$E5CA/$E5E5**	Input and process SHIFT–RUN/STOP, RETURN, etc.
QTSWC	**$E684/$E6B8**	Flip quotes flag ($D4) if A contains quotes on entry.
PRT	**$E716/$E742**	Print character in A to screen, like PRINT CHR$; handles such characters as home cursor, clear screen, delete, etc.
CHKCOL	**$E8CB/$E912**	Test A for character color code; change color in $286 if one is found.
COLTAB	**$E8DA/$E921**	Table of color-change codes, arranged Black, White, Red, Cyan, etc.
SCROL	**$E8EA/$E975**	Scroll screen up. If the top line is more than 40 characters long, the routine scrolls up appropriate number of lines to completely remove it. The CTRL key is tested for by directly interrogating the CIA chip, and a slight delay is performed if it is held down.
DSPP	**$EA13/$EAA1**	Put the character in A onto the screen at the current cursor position; no checking for control characters, etc., is performed. The color for the character is held in X.
KEY	**$EA31/$EABF**	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 interrupt register at $DC0D in the CIA is cleared; the A, X, and Y registers are pulled from the stack and restored; and a return from interrupt instruction (RTI) continues processing the main program.
KBDTBL	**$EB81/$EC46**	Tables to convert keyboard matrix values to CBM ASCII values.
VICINT	**$ECB9/$EDE4**	Table of values from which VIC chip is initialized (the exact values vary internationally, depending on the local television standards).
LDRUN	**$ECE7/$EDF4**	The characters LOAD <RETURN> RUN <RETURN>, transferred to the keyboard queue when SHIFT–RUN/STOP is pressed.
RSTRAB	**$EEBB/$EFA3**	Part of the routine used by NMI when servicing RS-232 output.
	$EF2E/$F09F	Flag RS-232 errors into ST byte.
	$F014/$F0ED	Output RS-232 character.
	$F086/$F14F	Get RS-232 character.
	$F0BD/$F17F	Tape messages.
SPMSG	**$F12B/$F1E2**	Output Kernal message from table starting at $F0BD if flag at $9D is set.
	$F179/$F230	Get character from tape.
	$F1DD/$F290	Output character to tape.
	$F38B/$F44B	Open tape file.
	$F3D5/$F495	Open serial device (printer, disk) file.

	$F409/$F4C7	Open RS-232 file.
	$F4BF/$F563	Load from disk.
	$F539/$F5D1	Load from tape.
	$F5FA/$F692	Save to disk.
	$F65F/$F6F8	Save to tape.
	$F6FB/$F77E	Table of I/O error numbers (1–9) and messages.
FAH	**$F72C/$F7AF**	Load next tape header.
	$F76A/$F7EF	Write tape header.
	$F7EA/$F867	Load named tape header.
	$F84A/$F8C9	Load tape.
	$F867/$F8E6	Write tape.
READ	**$F92C/$F98E**	Routines for tape reading.
WRITE	**$FBA6/$FBEA**	Routines for tape writing.
START	**$FCE2/$FD22**	Reset routine; entered from the 6510/6502 RESET vector at $FFFC. If a ROM cartridge is present, JMP ($8000) runs it. Otherwise, the routine calls RAMTAS, RESTOR, IOINIT, CINT, and NEW. Note that all other RAM is unaltered, so BASIC programs can be recovered after reset.
RAMTAS	**$FD50/$FD8D**	Fill low RAM (except for the stack area) with zeros, find the start and end of contiguous RAM for BASIC, and set the appropriate screen position according to the amount of memory present.
IOINIT	**$FDA3/$FDF9**	Initialize CIA chips on power-up.
NMI	**$FE43/$FEA9**	NMI routine; entered from the 6510/6502 NMI vector at $FFFA. The JMP ($318) at $FE44 routes control back to $FE47; altering this vector is one way to modify RUN/STOP–RESTORE. If the RUN/STOP key is down, Kernal routines RESTOR, IOINIT, and CINT are called, and a warm start of BASIC is performed by doing a JMP ($A002). This sequence is also performed on BRK. Otherwise, the interrupt is the result of RS-232 activity.
	$FEC2/$FF5C	RS-232 baud rate table (22 bytes; varies internationally).
PULS	**$FF48/$FF72**	IRQ or BRK routine; entered from the 6510/6502 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 $EA31; if it was a BRK operation, JMP ($316), usually to part of the NMI sequence at $FE66, 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.

For example, 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 64s; moreover, that jump table entry would also work on the Commodore VIC-20 and PET/CBM computers. On the 64, JMP $FFD2 arrives at $F1CA via an indirect jump through the RAM vector in locations $326–$327.

The 64's Kernal jump table begins at location $FF81. Note that the table entries have their own labels, which may be different from the labels of the routines they point to.

Label	**64/VIC**	**Jump Table Entry**	**Descriptions**
PCINT	$FF5B/$E518	$FF81 CINT	Initialize screen editor and video chip, set interrupt frequency.
IOINIT	$FDA3/$FDF9	$FF84 IOINIT	Initialize input/output chips.
RAMTAS	$FD50/$FD8D	$FF87 RAMTAS	Test and initialize RAM.
RESTOR	$FD15/$FD52	$FF8A RESTOR	Restore standard input/output vectors.
VECTOR	$FD1A/$FD57	$FF8D VECTOR	Store/set input/output vectors.
SETMSG	$FE18/$FE66	$FF90 SETMSG	Enable/disable Kernal control message output to screen.
SECNDK	$EDB9/$EEC0	$FF93 SECOND	Send secondary address for LISTEN command on serial bus; LISTEN must be called before using this routine.
TKSAK	$EDC7/$EECE	$FF96 TKSA	Send secondary address for TALK command on serial bus; TALK must be called before using this routine.
MEMTOP	$FE25/$FE73	$FF99 MEMTOP	Read/set BASIC top-of-memory limit.
MEMBOT	$FE34/$FE82	$FF9C MEMBOT	Read/set BASIC bottom-of-memory limit.
SCNKK	$EA87/$EB1E	$FF9F SCNKEY	Scan keyboard.
SETTMO	$FE21/$FE6F	$FFA2 SETTMO	Set serial bus time-out.
ACPTRK	$EE13/$EF19	$FFA5 ACPTR	Get a byte from a serial device (usually disk).
CIOUTK	$EDDD/$EEE4	$FFA8 CIOUT	Output a byte to a serial device (usually a printer or disk).
UNTLKK	$EDEF/$EEF6	$FFAB UNTALK	Send an UNTALK command to devices on the serial bus.
UNLSNK	$EDFE/$EF04	$FFAE UNLSN	Send an UNLISTEN command to devices on the serial bus.
LISTNK	$ED0C/$EE17	$FFB1 LISTN	Cause a device on the serial bus (usually a printer or disk) to listen.
TALKK	$ED09/$EE14	$FFB4 TALK	Cause a device on the serial bus (usually a disk drive) to talk.
READSS	$FE07/$FE57	$FFB7 READST	Read status byte into A.
SETLFS	$FE00/$FE50	$FFBA SETLFS	Set file number, device number, and secondary address.
SETNAM	$FDF9/$FE49	$FFBD SETNAM	Set filename.
NOPEN	$F34A/$F40A	$FFC0 OPEN	Open a file for reading or writing. Uses RAM vector at $031A.
NCLOSE	$F291/$F34A	$FFC3 CLOSE	Close a file. Uses RAM vector at $031C.
NCHKIN	$F20E/$F2C7	$FFC6 CHKIN	Prepare a file for input. Uses RAM vector at $031E.

NCKOUT	**$F250/$F309**	**$FFC9 CHKOUT**	Prepare a file for output. Uses RAM vector at $0320.
NCLRCH	**$F333/$F3F3**	**$FFCC CLRCHN**	Restore default I/O devices. Uses RAM vector at $0322.
NBASIN	**$F157/$F20E**	**$FFCF CHRIN**	Get a character from the designated input device. Uses RAM vector at $0324.
NBSOUT	**$F1CA/$F27A**	**$FFD2 CHROUT**	Send a character to the designated output device. Uses RAM vector at $0326.
LOADSP	**$F49E/$F542**	**$FFD5 LOAD**	Load data into memory from disk or tape.
SAVESP	**$F5DD/$F675**	**$FFD8 SAVE**	Save memory block to disk or tape.
SETTMK	**$F6E4/$F767**	**$FFDB SETTIM**	Set TI clock.
RDTIMK	**$F6DD/$F760**	**$FFDE RDTIM**	Read TI clock.
NSTOP	**$F6ED/$F770**	**$FFE1 STOP**	Test whether RUN/STOP key is pressed. Uses RAM vector at $0328.
NGETIN	**$F13E/$F1F5**	**$FFE4 GETIN**	Get a character, usually from the keyboard. Uses RAM vector at $032A.
NCLALL	**$F32F/$F3EF**	**$FFE7 CLALL**	Abort all I/O and close all files. Uses RAM vector at $032C.
UDTIMK	**$F69B/$F734**	**$FFEA UDTIM**	Add 1 to TI clock; reset to 0 if the count reaches 240000.
SCRENK	**$E505/$E505**	**$FFED SCREEN**	Return the maximum number of screen columns and rows in X and Y (40 and 25, respectively, for the Commodore 64).
PLOTK	**$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 CIA chip registers.

6510/6502 Hardware Vectors

The 6510/6502 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 6510/6502'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 64 and VIC.

Label	Vector	64/VIC	Descriptions
NMI	**$FFFA**	**$FE43/$FEA9**	When the 6510/6502 receives an NMI (Non-Maskable Interrupt) signal, it causes a jump to the address held here.
RESET	**$FFFC**	**$FCE2/$FD22**	When the 6510/6502 receives a RESET signal, it causes a jump to the address held here.
IRQ	**$FFFE**	**$FF48/$FF72**	When the 6510/6502 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

- Graphics with BASIC
- Graphics with Machine Language
- The VIC-II Chip
- User-Defined Characters
- Bitmapped Graphics
- Sprites

Graphics

This chapter starts with the simplest types of graphics using only ordinary BASIC and progresses to full-screen graphics and motion. All the special graphics effects of the 64 are covered.

Graphics with Basic

Effective graphics can be obtained with ordinary BASIC. Before going into programming details, here's a look at the way the 64 stores its standard characters.

Each character is made up of 8 dots by 8 dots on the screen. (Commodore's printers use different dot layouts and cannot easily give an identical copy of the screen—apart from the difficulty with color.) The actual pattern of dots making up each character is stored in ROM at $D000 (53248) to $DFFF (57343), a total of 4K bytes of memory. There are four subdivisions of this ROM:

$D000–$D3FF	Uppercase plus extended graphics
$D400–$D7FF	Reversed uppercase plus extended graphics
$D800–$DBFF	Lowercase with uppercase and some graphics
$DC00–$DFFF	Reversed lowercase with uppercase and some graphics

This is where the character ROM is placed from the point of view of the 6510; as you'll see in the section on user-defined graphics, the VIC-II chip is wired to "see" the character ROM elsewhere in memory.

Each byte is made up of eight bits, which correspond neatly to a single row of dots in a character definition. So every character takes eight bytes to define. For example, the first definition, for the @ sign, from 53248, is stored like this:

Byte Value	Bit Equivalent Which Defines the Character
28 ($1C)	00011100
34 ($22)	00100010
74 ($4A)	01001010
86 ($56)	01010110
76 ($4C)	01001100
32 ($20)	00100000
30 ($1E)	00011110
0 ($00)	00000000

Each 1 in the character definition appears on the screen as the foreground color and each 0 as the background color. Elsewhere in the ROM, similar patterns appear. The eight bytes from 54272, for example, correspond to the reverse-video @ and have exactly the opposite bit pattern; dots which were foreground with @ are now background and vice versa.

Since the total amount of memory dedicated to character definitions is 4K, it provides room for 4096/8 = 512 characters. Because the screen memory area holds only ordinary bytes, it is possible for each position in the screen RAM to select only 1 of 256 possibilities; thus, 256 characters at most can be displayed simultaneously. In BASIC, much of the screen usually contains spaces, of course, so the display is made from some of the 254 nonspace characters. There are two distinct sets of characters; SHIFT–Commodore key switches between them by directing the VIC-II to fetch its character definitions either from $1000, which is set when the 64 is switched on and is the uppercase mode, or from $1800, lowercase mode. Lowercase can be selected by PRINT CHR$(14), and uppercase by PRINT CHR$(142).

The two sets are identical to those in Commodore's other machines; the idea is that one can be used for word-processing applications, where the distinction between capital and lowercase letters is important, and the other can be used for pictorial applications, for example, using playing-card symbols. Because of this they are often called *text* and *uppercase/graphics* modes.

This simple program displays all 256 characters of either mode in white at the top of the screen. Press SHIFT–Commodore key to flip between modes; note how many characters are present in both modes:

```
10 FOR J=0 TO 255: REM 256 CHARACTERS NEED 256 SCREEN LOCATIONS
20 POKE 1024+J, J: REM SCREEN STARTS AT 1024: POKE 0,1,2 ETC
30 POKE 55296+J,1: REM SET COLOR RAM TO WHITE
40 NEXT
```

Tables of these characters are available for reference in the Appendices. Apart from space and SHIFT-space, which PEEK as 32 and 96, there is no duplication of character definitions. There is a rather confusing distinction between characters as they are POKEd into the screen (Appendix I) and character codes that are printed (Appendix H). PRINT translates many characters in special ways—changing color, clearing the screen, moving the cursor up and down or to the top of the screen, and so on. Some, like RETURN, are fairly standard, while others are peculiar to the 64.

Appendix G lists the control functions associated with certain ASCII codes. True ASCII reserves the first 32 character codes for control information, and Commodore has borrowed this idea. The displayed characters corresponding to PRINTed codes are in fact closer to true ASCII than is the case in earlier CBM machines, so conversion to true ASCII is easier. However, the upper- and lowercase alphabets are interchanged in relation to true ASCII.

Only some of the 256 screen characters can be displayed by using statements of the form PRINT CHR$(N). Since some CHR$ codes are for control purposes, like cursor-move commands, there are only 128 ordinary printing characters; all are obtainable by typing key combinations on the keyboard. The reverse feature allows any of the 256 screen characters to be displayed using PRINT; the ordinary character is preceded by a {RVS} character.

Within both blocks of 256 characters, reverse characters are arranged in step with the unreverse characters, but displaced by 128. An easy way to reverse characters in the screen RAM is to set bit 7, or in BASIC terms, add 128 (or, OR 128). Try POKE 55296,1 (to set the color RAM location for the top left of the screen) then POKE 1024,128. The fact that this flag or {RVS} is necessary to print a complete graphics set can be irritating if you have laboriously designed a large graphic display on the screen. It is impossible to save reverse characters in strings by inserting a line number and quotation marks before the characters, and then pressing RETURN. Instead, the strings need embedded {RVS} and {OFF} characters to flip between modes. Block saving of the relevant part of memory may be best. See Chapter 6.

There is no simple translation between unSHIFTed and SHIFTed keys, but usually setting bit 6 of the screen code to 1 displays the SHIFTed version. In BASIC terms, add 64 (or, OR with 64). Try POKE 55296,1: POKE 55297,1 to set the color RAM, then POKE 1024,1, and POKE 1025,65 in lowercase mode.

Note that the pairs of characters on the front right of most keys apply only in uppercase/graphics mode, the mode selected when the machine is switched on. After SHIFT–Commodore key puts the machine into lowercase, only the left-hand graphics symbol can be displayed on the screen, and a SHIFTed key gives the uppercase version, except for a few keys with no SHIFTed version, like @ and *. So the right-hand set of graphics is unobtainable in lowercase mode. Fortunately, some very useful graphics are retained; for example, boxes can be ruled on the screen, in either mode, using Commodore key–A, Commodore key–S, Commodore key–Z, Commodore key–X, SHIFT-* and SHIFT-–.

Toggling between the two modes with SHIFT–Commodore key can be disabled by PRINT CHR$(8) and reenabled with PRINT CHR$(9), or with POKE 657,128 and POKE 657,0 which set the relevant flags. If programs with user-defined characters fail to disable this toggle, SHIFT–Commodore key can produce odd results as character definitions are looked for in a region $800 bytes away from that intended by the programmer.

Some graphics symbols are missing from the keys. Thirty-one keys have a pair of symbols, making 62. Adding pi and SHIFT-space gives 64 graphics characters. But four characters, only accessible in lowercase mode, also exist and are listed in the cross-reference table of graphics: they are Commodore key–up arrow (checkerboard characters), Commodore key–* and SHIFT-£ (sloping diagonal lines), and SHIFT-@ (a square root or check mark).

Printing BASIC Graphics

This is certainly the easiest way to produce graphics effects. First, though, let's examine what PRINT actually does. PRINT has to interpret the information it's given and, in the case of printing characters, convert them into POKEs into the correct part of screen and color RAM. In the case of special, nonprinting characters, PRINT performs operations like selecting uppercase mode, changing foreground color, moving the cursor around on the screen, and so on. This is complicated and relatively slow. It uses memory locations to store current color (646/$286), status of the reverse flag (199/$C7), and the position on the screen at which the next character is to be printed (row is 214/$D6, column is 211/$D3), among other things. Try POKE 646,7. The 64 now prints in yellow, as though you'd typed PRINT "{YEL}" or CTRL-YEL. POKE 199,1: PRINT "HELLO" prints HELLO in reverse. The reverse flag is, however, turned off when RETURN is printed.

PRINT uses the Kernal output routine $FFD2 to put characters in the screen. Try POKE 780,65 : SYS 65490. This uses $FFD2 and has the same effect as PRINT CHR$(65). ML programmers can trace the routine to $E716; program control goes to $E7D4 for characters above 128, while characters from 0 to 127 are processed from $E72A. Comparison instructions look for RETURN, space, SHIFT-space, and so on. The actual routine which POKEs the character is at $EA1C. The accumulator holds the character and the X register holds its color code.

Examples using PRINT. Programmers unused to the graphics set, or looking for new ideas on graphics, could experiment with a short program like Program 12-1, which fills the screen with repeats of whatever string is entered:

Program 12-1. Simple Print Demo

```
10 INPUT "GRAPHICS";G$
20 PRINT "{CLR}";
30 PRINT G$;:IF PEEK(214)<23 THEN 30
40 PRINT:GOTO 10
```

Try, for example, Commodore key–* and SHIFT-£, or Commodore key–A, Commodore key–S, Commodore key–Z, and Commodore key–X, or other combinations of characters, generally in uppercase/graphics mode.

If the string is preceded by quotes, {RVS} and color-change characters may be included as in this example:

?"{RVS}{BLK}{SPACE}{WHT}{SPACE}{RED}{SPACE}{CYN}{SPACE}{PUR}{SPACE}
{GRN}{SPACE}"

Program 12-2 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. Try also SHIFT-E, Commodore key–*, and Commodore key–+ characters. (We'll see more impressive examples when we deal with user-defined characters.)

Program 12-2. Simple Design

```
10 A$(0)="N":A$(1)="M":A$(2)="V"
20 X$="":L=RND(1)*20+1
30 FOR J=1 TO L
40 X$=X$+A$(RND(1)*3):NEXT
50 FOR J=1 TO 999/L:PRINT X$;:NEXT
60 GET X$:IF X$="" THEN 60
70 RUN
```

Program 12-3 shows how strings can be overprinted to produce the effect of movement. The colored message is continuously scrolled to the left:

Program 12-3. Simple Motion

```
10 S$="{WHT}THIS IS {CYN}A {PUR} {RVS}MOVING{OFF}
   {SPACE}{GRN}MESSAGE! "
20 FOR J=1 TO LEN(S$)
30 PRINT MID$(S$,J)LEFT$(S$,J)
40 C$=MID$(S$,J,1)
50 IF C$>=" " AND C$<="Z" THEN FOR K=1 TO 50: NEXT
60 PRINT "{UP}";: NEXT: RUN
```

In Program 12-4, four substrings are extracted from the graphics string B$ and are printed to give four rows of characters, alternate rows moving in opposite directions across the screen:

Program 12-4. Froggie Graphics

```
10 POKE 53281,0:POKE 53280,0:PRINT "{WHT}{CLR}"
20 A$="{7 SPACES}"
30 B$=A$+"Q"+A$+"A"+A$+"Z"+A$+"S"+A$+"X"
40 B$=B$+B$+B$+B$
100 FOR J=1 TO 40
110 D1$=MID$(B$,J,40)
120 D2$=MID$(B$,80-J,40)
130 D3$=MID$(B$,J+20,40)
140 D4$=MID$(B$,100-J,40)
150 PRINT "{HOME}"+D1$+"{HOME}{3 DOWN}"+D2$+"
    {HOME}{6 DOWN}"+D3$+"{HOME}{9 DOWN}"+D4$
160 NEXT:FOR D=1 TO 50:NEXT:GOTO 100
```

Lines 110–140 extract the substrings from a different position in B$ with each pass through the loop; each substring is like a window moving along the string. Line 150 prints them, first positioning the cursor at the top-left position of the screen and spacing them out vertically using {DOWN} characters. B$ contains a repeating cycle of 40 characters (graphics characters separated by spaces) to produce continuity in the movement of the graphics characters.

In Program 12-5, colors are randomly selected and mirrored, using string arrays, to give an attractive symmetry when used to color reverse spaces (see line 60). Each display takes about 15 seconds to generate. Because of the large number of strings and despite the fact that most are very short, there is a potential problem with garbage collection. The CLR in line 70 discards all the strings that have just been used, allowing the next display to be constructed with a completely uncluttered string memory, and avoids the problem.

Program 12-5. Kaleidoscope

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
5 POKE 53281,0:POKE 53280,0:PRINT"{CLR}"    :rem 40
10 DIM M$(38,19),A$(19),RN$(38)            :rem 246
15 CO$="{BLK}{WHT}{RED}{CYN}{PUR}{GRN}{BLU}{YEL}
   Ɛ1ƷƐ2ƷƐ3ƷƐ4ƷƐ5ƷƐ6ƷƐ7ƷƐ8Ʒ"                  :rem 13
20 FOR J=0 TO 38:RN$(J)=MID$(CO$,RND(1)*16+1,1)+"
   {SPACE}":NEXT                            :rem 173
25 FOR J=8 TO 19:FOR K=0 TO J:C$=RN$(J-K) :rem 149
30 M$(K,J)=C$:M$(J,K)=C$                   :rem 235
35 M$(38-J,K)=C$:M$(38-K,J)=C$              :rem 32
40 NEXT:NEXT                                :rem 28
45 FOR J=8 TO 19:A$(J)="":FOR K=0 TO 38    :rem 190
50 A$(J)=A$(J)+M$(K,J)                      :rem 80
55 NEXT:NEXT                                :rem 34
60 PRINT "{HOME}":FOR J=8 TO 19:PRINT "{RVS}" A$(J
   ):NEXT                                  :rem 205
65 FOR J=18 TO 8 STEP-1:PRINT "{RVS}" A$(J):NEXT
                                            :rem 77
70 CLR:GOTO 10                              :rem 28
```

POKEing BASIC Graphics

PRINTing to the screen involves the 64 in time-consuming calculations, but POKEing to the screen can be even slower. POKE itself is not a fast command in BASIC. Machine language "pokes" that write characters directly to the screen RAM are much faster than BASIC ones, because the 6510 processor has fundamental commands which perform this function of transferring data from one memory location to another. But the BASIC POKE command spends a lot of time in calculations, so there is no great speed advantage. Color RAM will also need to be POKEd, unless the screen background color has been selected to make this unnecessary or the color RAM is already set satisfactorily. FOR I=0 TO 999: POKE 1024+I,1:POKE 55296+I,1: NEXT fills the whole screen. Note that it's slower, in fact, than PRINTing.

POKE has some advantages over PRINT. Any part of the screen can be changed without the need to keep track of the cursor position. PRINT is also liable to produce unwanted effects, like scrolling the screen when the bottom-right character is printed, and linking two lines together which makes the effect of RETURN unpredictable. POKEing has none of these side effects. The complete character set is available using POKE, too, without the need to use {RVS}.

To POKE to the screen, you need to know where the screen is. For now, the discussion will concern itself with the default position. Later you'll see how to move the screen around in the 64's memory. The screen can display 25 lines, each containing 40 characters, so its RAM consists of 1000 memory locations. It normally lies between 1024 ($0400) and 2023 ($07E7); the color RAM, each byte of which corresponds to a byte in the screen RAM, is at 55296 ($D800) to 56295 ($DBE7). Note that the color RAM is *always* here, irrespective of which bank the VIC-II chip is currently looking at. If you are unused to color RAM, try POKE 1024+40*3,1: POKE 55296+40*3,0 to plot a black *A* at the start of the third line. It is essential to know what value to POKE; consult Appendix I for a table of the 256 characters available by POKEing, in both lowercase and uppercase mode.

As an example, Program 12-6 puts solid squares of random color onto random locations in the 64's screen:

Program 12-6. Simple Poke

```
10 L=RND(1)*1001
20 POKE 1024+L,160
30 POKE 55296+L,RND(1)*16
40 GOTO 10
```

Finding the offset from the start of the screen for any given line and column is simple if you take some care in numbering: it is easiest to start at 0, so the horizontal position is 0–39, and the vertical position is 0–24, with 0 being the top of the screen. The offset is then 40*vertical position + horizontal position. The subroutine below, Program 12-7, POKEs the character X with color C into the screen at position H across and V down.

Program 12-7. Simple Subroutine Poke

```
30000 POKE 1024+40*V+H,X
30010 POKE 55296+40*V+H,C
30020 RETURN
```

The next example program draws a maze. This example (based on the work of C. Bond in *COMPUTE!'s First Book of Commodore 64 Games*—which includes an ML translation) draws a *simply connected* maze (a maze that is basically a contorted tube with no isolated islands within it). The algorithm uses space characters to mark boundaries, so there's an unused border of space characters. This version selects a random start point, and on finishing, POKEs A and B into the two points furthest removed from each other in the maze. Line 114 records the current longest path and can be deleted. Conversion to ML is needed to make it run faster; white-on-white plotting, for example, followed by color RAM POKEs is necessary if the plotting process is to be invisible.

Program 12-8. Maze Demo

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 POKE 53280,PEEK(53281):PRINT "{CLR}"     :rem 213
20 A(0)=-2:A(1)=-80:A(2)=2:A(3)=80          :rem 160
30 SC=1024:A=SC+81+80*INT(10*RND(1))+2*INT(10*RND(
   1))                                       :rem 125
40 FOR J=1 TO 23:PRINT"{RVS}{39 SPACES}":NEXT
                                             :rem 161
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
114 PRINT "{HOME}" SMAX;FIN                  :rem 203
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 120             :rem 110
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 GOTO 160                                 :rem 103
```

Graphics with Machine Language

BASIC is likely to be slow when dealing with graphics. This section discusses typical ML methods, which are much faster. Knowledge of machine language is not necessary to run these examples.

Printing Characters to the Screen

The routine at $FFD2, the Kernal's CHROUT routine, behaves like PRINT, except that it's your responsibility to store the characters you want to be printed in RAM. The speed advantage is considerable, but the price to be paid is the need to organize memory. Type in the following short BASIC program, which loads the machine language routine. The last line of Program 12-9 holds the ASCII values for the letters HELLO preceded by a space and followed by a zero byte which signals the end of the string.

Program 12-9. Simple ML Output

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 FOR J=49152 TO 49171:READ X:POKE J,X:NEXT
                                           :rem 219
20 DATA 162,0,189,14,192,240,6             :rem 188
30 DATA 32,210,255,232,208,245,96           :rem 84
40 DATA 72,69,76,76,79,0                   :rem 172
```

This is the ML:

```
RPT    LDX  #$00      ; X TO 0. USED AS POINTER
LOOP   LDA  TABLE,X   ; GET NEXT CHARACTER
       BEQ  EXIT      ; 0 SIGNALS END OF STRING
       JSR  $FFD2     ; OUTPUT CHARACTER
       INX            ; INCREMENT POINTER
       BNE  LOOP      ; CONTINUE
EXIT   RTS
TABLE  .BYT "HELLO",0 ; STRING TERMINATED BY 0: MAX LENGTH 255
```

Run Program 12-9 to load the ML, then type SYS 49152 to print HELLO. Although this example uses only straightforward lettering, the technique can easily be extended to include color change or cursor-move characters so that a 3 × 3 colored block of characters, say, can be printed at the current cursor position with a SYS call.

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 obtain. The high bit of each character code in screen RAM determines whether it's reverse or not; all we need to do is replace each character by its equivalent with the high bit reversed. LDA from an address, EOR #$80 to flip the high bit, then STA back into the address does this in ML. The program given here has been made user-friendly by incorporating parameters. It is relocatable, but the loader places it at 49152. When it's been loaded SYS 49152,1024,40 reverses 40 characters starting at 1024, the topmost line of the 64's screen.

Program 12-10. ML Reverse

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 FOR J=49152 TO 49178:READ X:POKE J,X:NEXT
                                            :rem 226
20 DATA 32,178,177,165,100,133,253,165,101 :rem 12
30 DATA 133,252,32,155,183,138,168,136,177 :rem 29
40 DATA 252,73,128,145,252,136,16,247,96   :rem 188
```

Here is the ML routine this loads:

```
START JSR    $B1B2   ; FETCH START ADDRESS FOR REVERSAL OPERATION
      LDA    $64
      STA    $FD
      LDA    $65
      STA    $FC     ; ($FC) POINTS TO START ADDRESS IN SCREEN
      JSR    $B79B   ; FETCH NUMBER OF CHARACTERS TO BE REVERSED
      TXA
      TAY            ; Y HOLDS THIS VALUE
      DEY
LOOP  LDA    ($FC), Y
      EOR    #$80
      STA    ($FC), Y
      DEY
      BPL    LOOP
      RTS
```

Other related effects include reversing all characters, with ORA #$80, and unreversing all characters with AND #$7F. Flashing the whole screen is more easily done by altering the background color rather than the characters—a couple of POKEs to 53281 ($D021) are all you need.

Plotting Rows or Columns

Table 12-1 groups similar graphics characters together. From the layout it is clear that the completeness of the graphics set enables some progress to be made toward accurate graphics. To show the approach, we'll write a routine which plots vertical columns on the screen, to the nearest 1/8 square (with 0–7 rows of dots on top of each column of solid characters).

Program 12-11. Histogram Demo

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
1 DATA 32,178,177,165,1ØØ,133,252,165,1Ø1,133,251,
  32,155,183,134,143                         :rem 9
2 DATA 32,155,183,138,16Ø,Ø,2Ø1,8,144,38,233,8,72,
  169,16Ø,145,251,165                        :rem 68
3 DATA 251,133,253,165,252,41,3,9,216,133,254,165,
  143,145,253,165,251                        :rem 67
4 DATA 233,4Ø,133,251,165,252,233,Ø,133,252,1Ø4,17
  6,214,17Ø,24Ø,21,189                       :rem 99
5 DATA 88,192,145,251,165,251,133,253,165,252,41,3
  ,9,216,133,254,165                         :rem 31
6 DATA 143,145,253,96,1ØØ,111,121,98,248,247,227,1
  6Ø                                          :rem 5
1Ø FOR J=49152 TO 49248:READ X:POKE J,X:NEXT
                                             :rem 224
1ØØ FOR J=Ø TO 39                            :rem 62
11Ø SYS 49152,1Ø24+24*4Ø+J,J,J+64:NEXT       :rem 23
15Ø FOR D=1 TO 1ØØØ:NEXT                     :rem 12
2ØØ FOR J=Ø TO 39                            :rem 63
21Ø SYS 49152,1Ø24+2Ø*4Ø+J,2,SIN(J*↑/2Ø)*8Ø+81:NEX
    T                                        :rem 13Ø
```

Line 10 loads the ML routine into RAM. To make the routine easy to use, three parameters are input as part of the SYS call. The syntax is:

SYS 49152,*bottom of column, color, height*

Lines 100–120 draw columns with increasing heights across the screen; lines 200–210 draw a histogram derived from a sine wave. Finding the offset from the start of the screen is simple with some care in numbering: it is easiest to start at 0, so the horizontal position is 0–39 and the vertical position is 0–24. Now the offset is 40*vertical position + horizontal position. The parameters to the SYS call in line 110 illustrate this.

Table 12-1. Quick Cross Reference to 64 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
	▚	▧	▨	✓

Double-Density Graphics

This program exploits the fact that all 16 possible graphics with internal quadrants exist on the 64.

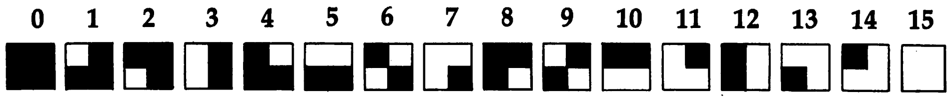
Program 12-12. Double Density

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 DATA 32,155,183,134,5,32,155,183,134,6,32,155
                                        :rem 50
```

```
11 DATA 183,134,2,165,5,48,89,201,80,176,85,165,6
                                          :rem 120
12 DATA 48,81,201,50,176,77,169,0,133,4,169,50,229
                                          :rem 163
13 DATA 6,70,5,38,4,106,38,4,133,6,166,4,169,0,133
                                          :rem 146
14 DATA 4,133,210,56,38,4,202,16,251,165,6,10,10
                                          :rem 30
15 DATA 101,6,10,38,210,10,38,210,10,38,210,133,20
   9                                      :rem 167
16 DATA 169,4,101,210,133,210,164,5,177,209,162,15
                                          :rem 145
17 DATA 221,117,192,240,4,202,16,248,96,138,5,4,17
   0                                      :rem 203
18 DATA 189,117,192,145,209,32,36,234,165,2,145,24
   3                                      :rem 219
19 DATA 96,32,126,123,97,124,226,255,236,108,127
                                          :rem 74
20 DATA 98,252,225,251,254,160            :rem 197
100 FOR J=49152 TO 49284:READ X:POKE J,X:NEXT
                                          :rem 16
```

Figure 12-1. Sixteen Quadrant Pictures



The ML routine replaces one of these graphics symbols with another, depending on the position of the "dot" to be plotted, so the whole screen in effect has a resolution of 80 × 50 small squares. The color can be set in any pair of squares. Graphics of this type don't compare to full hi-res pictures, but they have the advantage of being completely compatible with BASIC, needing no special POKEs or bitmap calculations. Note that the algorithm is designed to ignore text and other characters which aren't among the 16 quadrants, so BASIC text can be intermingled with dot diagrams.

The syntax is SYS 49152,*X,Y,COLOR* where X=0–79, Y=0–49, COLOR=0–15. All parameters are evaluated, so using variables (SYS *P,X,Y,C*) is accepted. The point X=0, Y=0 starts at the bottom left of the screen.

Changing Color RAM with ML

Color RAM occupies 1000 bytes from 55296 ($D800) on. Unlike, for example, the background color, which can be changed with one single POKE, each byte in the color RAM has to be changed if an entire screen of characters' colors is to be changed at once. Program 12-13 alters color RAM only, leaving the characters unchanged. POKE 842 with the color number (1–15), and SYS 830 to run the routine.

Program 12-13. Change Color RAM

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 FOR J=830 TO 853:READ X:POKE J,X:NEXT    :rem 11
20 DATA 169,216,133,3,169,0,133,2,168,162,4,169
                                             :rem 8
30 DATA 7,145,2,200,208,251,230,3,202,208,246,96
                                            :rem 41
```

When the screen is cleared, screen RAM is filled with space characters. The foreground color is now irrelevant; only the background color shows. The original ROM version of the 64 fills color RAM with white after clearing the screen, so that characters POKEd to screen RAM show up in white. Some newer 64s fill color RAM with the background color, stored in 53281, so that POKEs to a cleared screen are invisible. The most recent version of the 64 fills color RAM with the current character color (stored in 646), so POKEs will be visible.

Because of these ROM changes, the effect of a simple screen POKE depends on when your 64 was built; the POKEd character may appear white, invisible, or the same color as the cursor. To make screen POKEs work correctly on all 64s, POKE color RAM with the desired value whenever you POKE a character into screen memory.

ML modifications of color RAM can give many effects. The following program gives the effect of motion by drawing colored bars, then cycling through RAM colors:

Program 12-14. Color RAM Motion

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 FOR J=49152 TO 49177:READ X:POKE J,X:NEXT
                                           :rem 225
20 B1=55296:B2=56296                        :rem 20
30 FOR J=1024 TO 2023:POKE J,160:NEXT      :rem 241
40 FOR Y=0 TO 24:FOR X=0 TO Y              :rem 252
50 P1=X+40*Y:P2=Y+40*X                     :rem 240
60 POKE B1+P1,Y:POKE B2-P1-1,Y             :rem 167
70 IF Y<20 THEN POKE B1+P2,Y: POKE B2-P2-1,Y
                                            :rem 95
80 NEXT:NEXT                                :rem 32
90 FOR J=0 TO 50:NEXT:WAIT 53266,128:SYS 49152:GOT
   O 90                                    :rem 222
100 DATA 160,0,132,253,162,4,169,216,133   :rem 159
110 DATA 254,177,253,24,105,1,145,253,136  :rem 219
120 DATA 208,246,230,254,202,208,241,96    :rem 123
```

The ML adds 1 to each color location. Line 90 delays, waits for the raster-line to be offscreen, then updates color RAM. Raster scanning is discussed in detail later.

Scrolling the Entire Screen

This section shows how to scroll the entire screen left, right, up, or down, moving one character's width or height. Color RAM must be moved to match so that the characters keep their colors, unless a deliberate effect of static color is desired. Because the movement is in whole characters, scrolling is somewhat jerky. Smooth scrolling is discussed later.

These routines have to move 960 or 975 characters to new positions in screen memory and repeat the process for color RAM. Not 1000, because in each direction 25 or 40 characters are lost when the scrolling effect overwrites them. A new row or column of characters has to be POKEd to complete the scroll, or the displayed characters can be stored and reused, giving an indefinitely repeating scroll effect.

Two thousand BASIC PEEKs and POKEs, with calculations, are slow, but repetitive loads and moves are easily written in ML, so this is a good application. ML programmers can alter the routines, for example, to scroll strips of background across the screen at different speeds to simulate motion.

The following four routines are each placed in RAM by a BASIC loader. When loaded, they can be called by a simple SYS command from BASIC and are therefore easy to use. All are relocatable, so if the start address is altered from 49152 they will run correctly, provided all the bytes are POKEd to RAM, and the SYS calls the correct starting address. There are limitations to the programs, but they do illustrate the basic concept of scrolling.

Program 12-15. Scroll Down

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 FOR J=49152 TO 49206:READ X:POKE J,X:NEXT
                                            :rem 218
30 DATA 169,7,133,253,169,191,133,252,169 :rem 240
40 DATA 219,133,255,169,191,133,254,160,0 :rem 225
50 DATA 177,252,160,40,145,252,160,0,177  :rem 171
60 DATA 254,160,40,145,254,165,252,208,2  :rem 172
70 DATA 198,253,198,252,165,254,208,2,198 :rem 253
80 DATA 255,198,254,165,253,201,3,208,218,96
                                            :rem 135
```

Program 12-16. Scroll Up

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 FOR J=49152 TO 49209:READ X:POKE J,X:NEXT
                                            :rem 221
30 DATA 169,4,133,253,169,0,133,252,169,216:rem 71
40 DATA 216,133,255,169,0,133,254,160,40,177
                                            :rem 114
50 DATA 252,160,0,145,252,160,40,177,254,160
                                            :rem 106
60 DATA 0,145,254,230,252,208,2,230,253,230:rem 47
70 DATA 254,208,2,230,255,165,252,201,192 :rem 221
80 DATA 208,222,165,253,201,7,208,216,96  :rem 180
```

Program 12-17. Scroll Right

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 FOR J=49152 TO 49215:READ X:POKE J,X:NEXT
                                         :rem 218
30 DATA 169,4,133,253,169,0,133,252,169,216:rem 71
40 DATA 133,255,169,0,133,254,162,24,160,39:rem 65
50 DATA 177,252,200,145,252,136,177,254,200:rem 66
60 DATA 145,254,136,136,16,241,24,165,252 :rem 226
70 DATA 105,40,133,252,165,253,105,0,133,253
                                         :rem 100
80 DATA 165,254,105,40,133,254,165,255,105 :rem 19
90 DATA 0,133,255,202,16,211,96          :rem 236
```

Program 12-18. Scroll Left

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 FOR J=49152 TO 49214:READ X:POKE J,X:NEXT
                                         :rem 217
20 DATA 162,22,189,235,3,157,234,3,189,235 :rem 25
30 DATA 215,157,234,215,232,208,241,189,234:rem 70
40 DATA 4,157,233,4,189,234,216,157,233,216:rem 73
50 DATA 232,208,241,189,233,5,157,232,5,189:rem 77
60 DATA 233,217,157,232,217,232,208,241,189:rem 74
70 DATA 232,6,157,231,6,189,232,218,157   :rem 136
80 DATA 231,218,232,208,241,96            :rem 198
100 FOR J=1063 TO 2024 STEP 40:POKE J,160:NEXT
                                         :rem 195
110 R=RND(1)                             :rem 135
120 IF R<.4 THEN X=X+1                   :rem 138
130 IF R>.6 THEN X=X-1                   :rem 145
140 IF X>15 THEN X=15                     :rem 74
150 IF X<0 THEN X=0                      :rem 221
160 H=X*40                                 :rem 2
170 FOR J=39 TO 39+H STEP 40:POKE 55296+J,5:NEXT
                                          :rem 87
180 FOR J=79+H TO 1000 STEP 40:POKE 55296+J,6:NEXT
                                         :rem 178
190 SYS 49152:GOTO 100                   :rem 162
```

How the Screen Is Scrolled

Thanks to the systematic memory mapping of the 64's screen, scrolling in any direction is quite easy. Figures 12-2 and 12-3 show scrolling in two directions (down and left) on a simplified 4 $\times$ 4 screen, labeled to show the effects of each scroll. The cells marked with question marks 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

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	?

It is easy to deduce that for the 64, scrolling up or down requires that the entire screenful of characters, excluding a set of 40 at the end, be moved 40 places along. Scrolling right or left, in the simplest case, only requires every screen character but one to be shifted one place along. This is what the left-scrolling program does. Simply enter the line 0 SYS 49152: GOTO 0 and run it to see the screen roll sideways.

With scroll down, the characters at the bottom-right must be moved first; otherwise, because there is no temporary storage and characters are moved directly into the screen, characters will be overwritten before they have been moved.

Speedy ML routines help. If screen RAM and color RAM are both moved, the fastest methods use LDA $0401,X:STA $0400,X:LDA $D801,X:STA $D800,X or similar commands to move each character; this takes about 1/50 second on the 64.

The VIC-II Chip

Color TVs and monitors display color by electronically causing small patches of color, called *phosphor*, to glow on the screen. This is *additive* color. Each extra color adds to the brightness, unlike painting or printing, which uses subtractive color. The primary colors for additive color are red, green, and blue. All three colors combined equally produce 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, too. 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, the 64's choice being yellow, cyan, and purple. Each is made of about equal amounts of two primary colors: yellow is red and green, cyan is green and blue, and purple is blue and red. On the 64, adjacent pairs of color-control keys are complementary: They add to white or gray. For example, red and cyan contain between them red, green, and blue, in the right proportions to mix to give white. The eight foreground colors obtainable with CTRL and a color key are simply the three primaries, each of which is either on or off, giving eight combinations. Of the further eight colors, three are light versions of the primary colors, but the three secondary colors are turned to shades of gray. Brown and orange have also been added.

There are three perceptual effects worth mentioning. One is that the weaker, less saturated colors are very influenced by stronger, adjacent colors, so the 64's gray tones can be visually stabilizing.

Another is the advancing effect of red and the receding effect of blue, which can be startling with solid blocks of these colors. Blue can look a long way behind red. There are related problems in getting a bright color to look bright when compared with, say, white, which has three times as many phosphors lighted. The VIC-II chip doesn't allow control over the luminance of screen pixels, so many color combinations haven't enough contrast to be distinct. Red lettering on blue is unreadable, for example.

Color RAM

The border and screen background colors are straightforward. Color RAM is a more difficult concept, but seems natural enough after a time. It is a block of RAM paralleling the screen RAM; each screen location has a corresponding color RAM location, whose lower nybble (four bits) determines the character's foreground color; thus, each screen character may have its own foreground color. (All characters have a common background color determined by the register at 53281.)

The colors that correspond to numbers in the foreground and background registers, and in the color RAM locations, follow:

Color Nybble	Color (Descriptions Vary Somewhat)
0	Black
1	White
2	Red
3	Cyan
4	Purple
5	Green
6	Blue
7	Yellow
8	Orange
9	Brown
10	Light Red
11	Gray 1, Dark Gray
12	Gray 2, Med. Gray
13	Light Green
14	Light Blue
15	Gray 3, Light Gray

Color RAM occupies 55296–56295 ($D800–$DBE7), a total of 1000 bytes. N that the VIC-II chip always fetches its foreground color information from here, ii respective of which VIC-II bank is in use.

Multicolor Mode

Understanding user-definable characters is essential to getting the most from mu color mode. However, the general idea is fairly easy to grasp. It is another Commodore compromise: In order to get more color into the screen, resolution is cut half. Below is a discussion of how this works with ordinary graphics; the principl the same in high-resolution mode.

Normally, a one in a character definition shows up in the foreground color, a a zero shows up in the background color; so only two colors are obtainable withi each 8 × 8 dot character area. Multicolor mode allows four colors to be selected character, at the cost of halving the horizontal resolution. Instead of 8 × 8 dots, i offers 4 × 8 "wider" dots, each of which can take one of four colors.

Multicolor mode is enabled by setting bit 4 of VIC-II register $16 to 1. This is done by using POKE 53270,PEEK(53270)OR16 (normally, POKE 53270,216). The following command switches back to normal mode: POKE 53270,PEEK(53270) AND239 (normally, POKE 53270,200).

The above POKEs enable and disable multicolor mode globally, over the who text area; but it must also be enabled on a character-by-character basis to have an effect. This is done by the value in the corresponding color RAM location: If it is from 0 to 7, then the character appears in ordinary mode, and if it is from 8 to 15, then the character will be in multicolor mode. In other words, bit 3 in a color RAM location determines whether the corresponding character is in ordinary or multicolor mode. Thus, the screen may simultaneously display multicolored and ordinary characters.

To get the feel of this, type some lettering in several colors, including the less saturated Commodore-key colors. Enable multicolor mode with the POKE given above. You'll see that characters in black through yellow are unchanged, while those in orange through light gray alter dramatically, because bit 3 of their color RAM has this dual function.

The source of the color in each two-dot unit is shown by the following table:

Bit Pattern:	Color Specified By:	Address of Register:
0 0	Background 0 color register (screen background color)	53281 $D021
0 1	Background 1 color register	53282 $D022
1 0	Background 2 color register	53283 $D023
1 1	Lower three bits of color RAM (character color)	

The three registers can take values from 0 to 15; the three bits specified by color RAM select values 0–7. Notice that units containing 00 appear as the background color whether the display is in standard or multicolor mode. Note that the border color in 53280 is independent of the background colors, unlike VIC-20's multicolor mode.

It follows from this table that an orange (Commodore key–BLK) character will be displayed as black when multicolor mode is enabled—try it with reverse-space block in orange. Similarly, a light green character switches to green.

Consider how the character *A* is defined in ROM:

Normal:	Multicolor:	Displays As:
0 0 0 1 1 0 0 0	00 01 10 00	BG0 BG1 BG2 BG0
0 0 1 1 1 1 0 0	00 11 11 00	BG0 CR CR BG0
0 1 1 0 0 1 1 0	01 10 01 10	BG1 BG2 BG1 BG2
0 1 1 1 1 1 1 0	01 11 11 10	BG1 CR CR BG2
0 1 1 0 0 1 1 0	01 10 01 10	BG1 BG2 BG1 BG2
0 1 1 0 0 1 1 0	01 10 01 10	BG1 BG2 BG1 BG2
0 1 1 0 0 1 1 0	01 10 01 10	BG1 BG2 BG1 BG2
0 0 0 0 0 0 0 0	00 00 00 00	BG0 BG0 BG0 BG0

The first illustration shows how the definition is interpreted in normal mode: zeros display in the background color and ones display in the foreground color, specified by the character's color RAM.

The second and third illustrations show how the bits are interpreted as grouped in pairs by the 64 in multicolor mode. The abbreviations BG0, BG1, and BG2 represent the three background color registers, which are set to 6 (dark blue), 1 (white), and 2 (red), respectively, on power-up. (The SX-64 sets BG0 to white, however.) CR is color RAM, which is 14 (pale blue) on powering-up the 64. Note that the three background colors apply over the whole screen area; only the character color can vary from character to character. Therefore, when designing multicolor graphics, select the three colors you wish to spread most widely on the screen, and let the character color vary locally.

Assuming the 64 has its power-up values, enter POKE 53270,216 to enable multicolor mode. All characters will be displayed in multicolor mode, since their color RAM value is greater than 7. Assuming the relevant registers have their power-up values, BG0 will show up as dark blue, BG1 as white, BG2 as red, CR as a dark blue (produced by the pale blue value with bit 3 stripped off). This is what the colors should be, but they may not show up particularly clearly on your TV or monitor.

The cursor disappears because the reverse space character is a block of bit pairs in the pattern 11; the color is given by color RAM and thus shows up in multicolor mode as dark blue. Type Commodore key–GRN to make it reappear: printing will continue in multicolor mode; CTRL-GRN will also make it reappear, but causes printing to continue in standard mode because of the different color RAM settings. Enter POKE 53283,1 to make BG2, as well as BG1, white; the multicolored characters now contain large areas of white. Type Commodore key–WHT followed by a few more characters: even larger areas now show as white, as BG1 and BG2 and CR are all now white. Usually, of course, contrasting colors will be used. CTRL-WHT will select a foreground color value less than 8; type this and then further characters: these display in standard mode, because of the color RAM value, and are unaffected by BG1 and BG2 settings.

These multicolor characters have a chunky appearance, since they have half the horizontal resolution of standard characters. They can be used for decorative borders and designs, and for graphics. You may need to experiment to find the best combinations of colors for this effect. They are easier to use than user-defined characters and take up no extra space in RAM. Finding characters which look right may be difficult, though.

With some work characters in multicolor mode can produce impressive results. For example, BG0 may be set to 12, and BG1 and BG2 to 8 and 14, giving orange and light blue and the local colors on a medium gray background, allowing, say, a gray sky, orange ground, and light blue middle-distance, with small objects in any of the eight main colors.

The following BASIC program lets you experiment with all combinations of BG0, BG1, BG2, and CR. It displays almost the entire character set twice, once at the top of the screen in standard mode and again below it in multicolor mode. The function keys f1, f3, f5, f7 advance the values in the three register values and the color RAM of the multicolor mode characters.

You may prefer to experiment with character sets in two colors only; if so, modify the program to POKE the background registers with 0, and make the function keys toggle, with POKE 53281,1−(PEEK(53281) AND 15) or a similar statement. The AND 15 is necessary to remove the high nybble, which varies. Also try replacing line 250 with 250 NC=1−NC.

Program 12-19. Multicolor Mode

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 VIC=53248:COL=55296                           :rem 230
20 PRINT "{CLR}"                                 :rem 198
30 FOR J=0 TO 239                                 :rem 66
40 POKE 1024+J*2,J:POKE COL+J*2,0                 :rem 21
50 POKE 1024+520+J*2,J:POKE COL+J*2+520,8        :rem 162
60 NEXT                                          :rem 165
100 POKE VIC+22,PEEK(VIC+22) OR 16                :rem 76
200 GET X$:IF X$="" THEN 200                     :rem 117
210 IF X$="{F1}" THEN POKE 53281,(PEEK(53281)+1) A
    ND 15                                        :rem 144
220 IF X$="{F3}" THEN POKE 53282,(PEEK(53282)+1) A
    ND 15                                        :rem 148
```

```
230 IF X$="{F5}" THEN POKE 53283,(PEEK(53283)+1) A
    ND 15                                   :rem 152
240 IF X$<>"{F7}" THEN 200                  :rem 168
250 NC=((PEEK(COL+520)+1) AND 15) OR 8      :rem 139
260 FOR J=0 TO 239:POKE COL+J*2+520,NC:NEXT
                                            :rem 135
270 GOTO 200                                :rem 100
```

Multicolor mode is probably the 64's most popular graphics mode. Although in theory resolution is halved, in practice TV limitations mean that 320 individual colored dots (that is, 40 sets of 8) aren't really distinguishable across a TV screen. The 64's Commodore key–+ character, for example, is *not* made of alternate 0's and 1's. It's composed of alternate 00 and 11 pairs. This is why multicolor characters often look similar to their normal equivalents, and why normal characters—Commodore key–Z, for instance—often appear thicker than you'd expect.

Even with multicolor mode enabled, characters don't *have* to be displayed in multicolor mode, which adds to the mode's versatility. Programs can be developed using PRINT and/or POKE to move characters around; such programs will work just as well if the graphics are redefined in multicolor form. This requires more work, since character definitions must be loaded into RAM and the VIC chip made to access them. However, this is still easier than full bitmapping.

Extended Background Color Mode

This is a relatively new display mode, and the VIC-20 has no analogous mode; it cannot coexist with other modes. The screen blacks out as long as multicolor mode or bitmapping is also switched on. Like multicolor mode, the full graphics set is divided by four to allow more color. Usually the 64's background color (BG0) extends over the whole background, and though each of 1000 character colors can vary, the background has to be in common—though this is disguisable by including solid blocks of local color. Extended background color mode allows the background and color of each character to be chosen from one of four colors. Dots are interpreted singly, not in pairs like multicolor mode.

The trade-off is only 64 characters can be displayed at one time. The two highest bits of each character determine the background color:

High Bits of Character	BG Color Specified By:	Address of Register:
0 0	Background 0 color register (usual screen background)	53281 $D021
0 1	Background 1 color register	53282 $D022
1 0	Background 2 color register	53283 $D023
1 1	Background 3 color register	53284 $D024

The displayable characters are the first 64 in the character definition area. The foreground color is set by the color RAM nybble. In summary, each of the 1000 characters' foregrounds can be set to colors 0–15; each background can be set to one of four colors, each of which may be 0–15; and only 64 differently shaped characters can be displayed, each in two colors at most.

Extended background color mode is selected by setting bit 6 of VIC-II register $11 to 1; this can be done with POKE 53265,PEEK(53265) OR 64. The command POKE 53265,PEEK(53265) AND 191 switches back to normal mode. (POKE 53265,91 for on and POKE 53265,27 for off normally work fine.)

You can reset the VIC-II registers using RUN/STOP–RESTORE. Background colors 0–3 are set to 6, 1, 2, and 3, corresponding to blue, white, red, and cyan. Now, enable extended background color mode with the above POKE. The cursor flashes red rather than pale blue, because reverse-space is POKEd as 160 to the screen: the bit pattern is %10100000, which is 32 with %10 as the leading bits. So it shows as a space character with background color governed by BG2, which is red.

Type a few unSHIFTed letters, and they will appear the usual light blue on dark blue. Now try SHIFTed letters; they are unSHIFTed on the screen, but their background is now white, governed by BG1. The POKE codes for A and SHIFT-A differ by 64, so the *same character* is displayed in extended background color mode.

Type CTRL-{RVS} followed by unSHIFTed letters; now the background is red, like the cursor, because bits %10 select register BG2. Finally, without pressing RETURN, type in a few SHIFTed letters: this {RVS}-SHIFT combination adds bits %11, selecting BG3's cyan background. The result is a bit hard to read on some sets; try POKE 53281,0: POKE 53283,7, setting BG0–BG3 to black, white, yellow, and cyan, with red lettering (POKE 646,2).

For a further demonstration, add these four lines to the "Multicolor Mode" demo program and run the result:

Program 12-20. Extended Background Color Mode

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
235 IF X$="{F8}" THEN POKE 53284,(PEEK(53284)+1) A
    ND 15                                   :rem 164
236 IF X$="M" THEN POKE(53265),PEEK(53265) AND 191
    : POKE 53270,PEEK(53270)OR16            :rem 82
237 IF X$="E" THEN POKE(53270),PEEK(53270) AND 239
    : POKE 53265,PEEK(53265)OR64            :rem 81
250 NC=PEEK(COL+520)+1                      :rem 215
```

Extended background color mode can now be selected by typing E, and multicolor mode by typing M. Pressing f8 advances the value in register BG3, while the other keys function as before.

You'll see the reduced character set and extra background colors clearly. The small available range of character shapes makes this mode unsuitable for most purposes. But if you're content with numerals, uppercase letters, and punctuation symbols, extended background color mode allows colored highlighting which is otherwise much harder to program. The unSHIFTed, SHIFTed, reverse, and reverse-SHIFTed characters (as ordinarily entered) will be displayed on background colors as stored in 53281–53284. Note that unSHIFTed space, conveniently, appears as the default background color.

Smooth Screen Scrolling

Sometimes it's nice to be able to display text scrolling smoothly up the screen or landscape-type graphics shifting left or right. The scrolling we've looked at so far shifts whole characters and is rather jerky. We can improve on this with the VIC-II chip's facility to move the screen. It positions the screen with single-dot resolution, allowing a maximum of eight dots of movement, so the whole picture can be shifted slightly. This allows screen scrolling, but since sprites have to be handled separately, the technique can be difficult. Bits 0–2 of $D011 (53265) set the vertical position, and bits 0–2 of $D016 (53270) set the horizontal position.

Here's the method for upward scrolling: the screen is moved up by one row of pixels (dots) seven times; on every eighth movement the screen moves back to the lowest position, and the screen characters are scrolled up by one whole character. Normally, this gives a jiggling motion at the top and bottom borders, which can be reduced by matching the border and background colors, or, better, by using the VIC chip to cut out the picture edges.

The screen scroll must be fast; otherwise, the TV scan will display part of the old picture, giving an uneven effect. The first demonstration, Program 12-21, is in BASIC.

Program 12-21. Change Vertical Position

```
10 V=(PEEK(53265)-1) AND 7
20 WAIT 53266,128,128:WAIT 53266,128
30 POKE 53265,PEEK(53265) AND 248 OR V
40 GOTO 10
```

Program 12-21 repeatedly moves the screen up, then flips the register back down when the register changes from 0 to 7. Line 20 helps insure that the VIC chip is altered only when the TV is scanning outside the picture; the timing can be fine-tuned by putting in extra colons.

Black bands may appear at the top or bottom of the picture. When the Y value is 3, as it is on power-up, the screen fits perfectly, but not otherwise. The VIC chip allows the edges of the picture to be suppressed. Bit 3 in $D016 (53270) selects 38 columns when set to 0, and bit 3 in $D011 (53265) selects 24 rows when set to 0. This affects only the appearance onscreen. The 64 still internally assumes a 40 × 25 format.

Add the line 5 POKE 53265, PEEK(53265) AND (255-8) to Program 12-21 and run it. The black bars are no longer displayed and the screen has one row less. In order to demonstrate scrolling, we need to scroll the screen up at regular intervals. A simple PRINT statement, or SYS 59626, will scroll up a whole line. BASIC has no sideways scroll, so a simple demonstration has to scroll up.

Add 25 IF V=7 THEN READ X$: PRINT "{HOME}{25 DOWN}" X$; to the program (don't forget the semicolon), plus some DATA statements with strings. You'll see the strings scroll up, after being written invisibly to the bottom line of the picture. However, it isn't possible to get a completely smooth scroll by this method, unless the text or graphics doesn't change. This is because screen scrolling typically takes about 1/25 second, so a mixture of the new and the old screen is displayed. This doesn't matter if the text is identical at each line of the screen, as you can see by altering line 25 to print some fixed X$ string.

Improving this requires the use of ML. The problem is the time spent scrolling the screen. (The situation is worse for bitmapped screens, which have more data to move.) One scan of the TV picture takes 1/60 (U.S.) or 1/50 (U.K.) second. A full-screen scroll must take less than about 1/50 second to be invisible. A full screen, and color RAM, cannot be scrolled in this time using BASIC; the example program above moves characters, not color RAM, because of this.

Program 12-22. Smooth Scroll

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
5 POKE 53270, PEEK(53270) OR 16                  :rem 16
10 FOR J=1 TO 10: C$=C$+"KLMN":NEXT              :rem 99
20 C$=LEFT$(C$,39)                              :rem 165
30 POKE 53265,PEEK(53265) AND 247               :rem 174
40 PRINT "{CLR}":FOR J=1 TO 25:PRINT "{DOWN}";:NEX
   T                                            :rem 123
50 FOR J=55296 TO 56295:POKE J,PEEK(646):NEXT
                                                :rem 250
60 FOR J=49152 TO 49213:READ X:POKE J,X:NEXT
                                                :rem 221
70 POKE 53265,PEEK(53265) OR 7                   :rem 26
100 SYS 49152                                   :rem 149
110 PRINT C$ "{UP}"                              :rem 91
115 C$=C$+CHR$(65+RND(1)*26):C$=RIGHT$(C$,39)
                                                :rem 125
120 FOR Y=1 TO 7                                 :rem 27
130 SYS 49152                                   :rem 152
140 FOR D=1 TO 15:NEXT                          :rem 176
150 NEXT Y:GOTO 100                              :rem 50
200 DATA 173,17,208,48,0,173,18,208,201,255,208
                                                  :rem 4
210 DATA 244,169,7,44,17,208,208,8,13,17,208
                                                :rem 120
220 DATA 141,17,208,208,4,206,17,208,96         :rem 125
230 DATA 169,3,141,45,192,141,48,192,162,4,160,64
                                                :rem 109
240 DATA 185,232,7,153,192,7,200,208,247,238,45
                                                 :rem 15
250 DATA 192,238,48,192,202,208,238,96           :rem 96
```

Load and run Program 12-22, then wait for the scrolling to begin. Each SYS call in the program scrolls the screen up one dot; it does this by checking the value in the Y register and decrementing it if it exceeds 0. When it becomes 0, Y is replaced by 7 and a scroll routine is called. The ML routine is located at 49152, so SYS 49152 can be used as a versatile scroll (note that it's not relocatable). Line 70 insures that the first PRINT statement in line 110 is in the invisible, bottom screen line, so the scroll is completely smooth. It's easy enough to modify the register checking and character shifting to scroll left.

User-Defined Characters

This section explains how new character sets can be created for the 64, by exploiting the VIC-II chip and controlling BASIC's memory map. Some of the earlier explanatory part also applies to the creation of bitmapped graphics, dealt with in the next section, and will not be repeated there.

Using the VIC-II Chip

The Video Interface Chip (VIC-II) controls the 64's screen display. Two locations, $D018 and $DD00 (53272 and 56576), are crucial in this. In BASIC, location $0288 (648) is also important.

Understanding the system is not particularly easy. Everything is designed around the hardware without any attempt to ease programming tasks, so don't feel too discouraged if it appears difficult. The key is to grasp a few details of the 64's software.

VIC-II is a chip with 14 address lines. Since $2^{14}=16K$, the chip can address a range of 16K bytes. This is actually the minimum amount necessary to the chip, since, as we'll see, it's required to look at at least 8K bytes of character definitions in bitmap mode. It could have been designed to address all 64K, but wasn't—perhaps because all the pins were already used. Unlike the VIC-20, whose VIC chip cannot look outside a narrow range of memory, the 64's VIC-II can be made to address any one of the four 16K subdivisions of 64 memory. Two bits in a CIA control this. The CIA extends the 14-bit range to 16 bits. Although this enhances the versatility of the 64, it means that only a 16K range is accessible to VIC-II at any given moment.

The 64 has all its ROM at the top of memory, but screen RAM is positioned low in memory, starting at $0400. The designers of the 64 included a hardware patch so that the character generator ROM and the screen could be simultaneously used by VIC-II. Character ROM appears to the 6510 to start at $D000, but the VIC-II chip is wired to see it as RAM from $1000, in the same bank as the screen. When VIC-II's registers are pointed to a different area of memory from usual, character definitions are taken from RAM. This trick is carried out by the logic array chip and is transparent to the programmer.

User-Defined and Bitmapped Displays

The 64 has two graphics modes: user-defined characters, discussed here, and bitmapped graphics. Both modes divide up the screen into 1000 square regions. User-defined characters, which are similar to the normal 64 characters, allow VIC-II to display up to 256 different characters. It follows that many screen characters must be identical, and in fact the screen is often largely filled with space. For example, simple line-drawing illustrations, where much of the screen is blank, in user-defined graphics use much less space than the bitmapped equivalent, so many more can be fitted into the 64's memory. A complete set of 256 characters takes 8*256=2K of memory to store. (Character ROM at $D000–DFFF is 4K, enough for *both* the 64's complete character sets.) Bitmapping allows every part of the screen to be individually controlled; with practice, this mode is easily recognized, since there's no duplication of portions of the screen. Images like pinball or flight simulations, and graphics-drawing programs which allow any part of the screen to be drawn on, must use bitmapping, where 8000 bytes are needed—nearly four times as many as user-defined character sets require. (Note that reduced-sized screens in either mode permit less memory to be used.)

A character editor is a program that helps you design your own user-defined characters, allowing up to 256 individual 8 × 8 dot characters to be designed. A full-screen graphics editor is appropriate to bitmapping. Examples of each are included in this book. Sprites, if they're used, must also be defined within the VIC-II's 16K bank.

How the VIC-II Controls the Display

The registers at $D018 and $DD00 control two major parameters. These are the screen starting position (or *video matrix*) and the character definition start (or *character base*). Try PRINT PEEK(53272) on your 64 just after power-up; the result is 21, and the 64's screen starts at $0400, while the character images are at $1000. SHIFT–Commodore key changes the PEEKed value to 23, implying that the character now starts at $1800 in lowercase mode. These registers are two special cases of the VIC-II chip's control over graphics; their relevant bits are usually labeled as shown here:

$D018 (53272)	VM13	VM12	VM11	VM10	CB13	CB12	CB11	1
$DD00 (56576)	x	x	x	x	x	x	B1	B0

Bit 0 of $D018 is unused; it is always set to 1. Bits 2–7 of $DD00 are irrelevant to graphics.

The Screen start position is controlled by bits VM13–VM10 (the four high bits of 53272), and by B1 and B0 (the two low bits of 56576). We can find the actual full 16-bit address by listing bits 0–15 and inserting the register values, like this:

Bit Number:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Screen Start:	B1	B0	VM13	VM12	VM11	VM10	0	0	0	0	0	0	0	0	0	0

Note that bits B1 and B0 are inverted (that is, a 1 value is treated as 0), and that all bits not controlled by the registers are set to 0.

Usually, $D018 and $DD00 contain 12 and 191 (=%0001 0101 and %xxxx xx11). Thus, bits VM13 to VM10 are 0001, and bits B1 and B0 are both 1. The screen therefore starts at %0000 0100 0000 00000 = $0400. Because bit VM10 is the least significant programmable bit controlling the screen position, and 2^{10}=1024, screen positions can be selected only to the nearest 1K. Where multiple screens are used, they can be stored next to each other, but not overlapping, in RAM.

The start of the character definitions is controlled by bits CB13–CB11, and by B1 and B0, in exactly the same way:

Bit Number:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Screen Start:	B1	B0	CB13	CB12	CB11	VS10	0	0	0	0	0	0	0	0	0	0

Again, B1 and B0 are inverted, and all lower bits are 0. The reset 64 has $15 in $D018, so, taking the right nybble 0101 and dropping bit 0, we see that CB13–CB11 are 010. The characters, therefore, are taken from %0001 0000 0000 0000 = $1000, the RAM image of the character-generator ROM. SHIFT–Commodore key toggles $D018 between $15 and $17, and we can see now that the second value sets the right nybble of $D018 to %0111, so CB13–CB11 are now 0111. The characters now come from %0001 1000 0000 0000 = $1800.

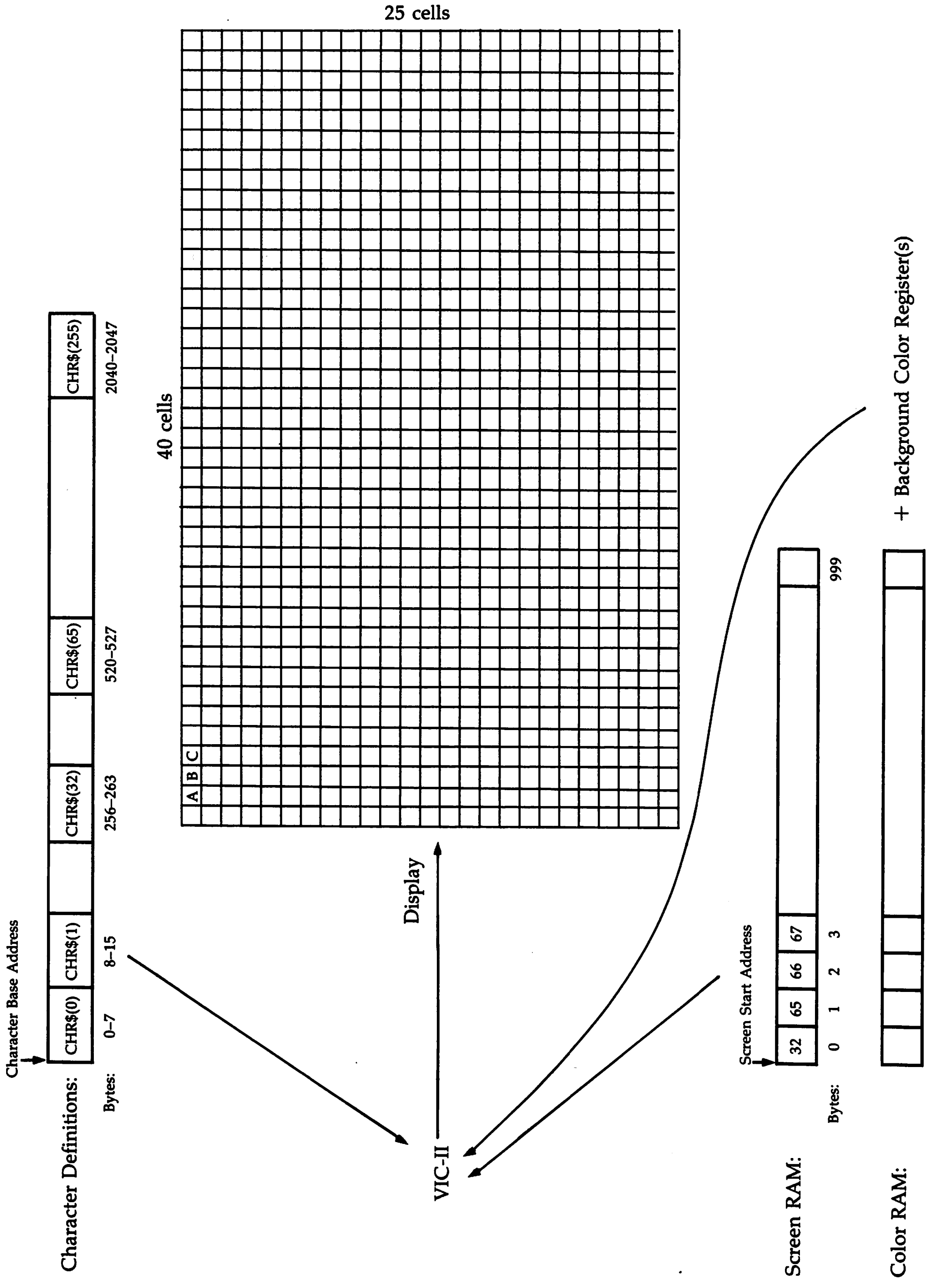
Bit CB11 has the least effect on assigning the memory region from which VIC draws its character definitions, controlling it to the nearest $800, or 2K, bytes. In other words, character memory is treated as though it's divided into 2K chunks by the 64's VIC chip. This allows multiple character sets to be placed in memory adjacent to each other. One example is the built-in graphics and lowercase sets, which are adjacent in memory.

How Characters Are Displayed

Figure 12-4 shows how screen RAM and character definitions are combined by the VIC-II chip as it generates the TV display. In the example the top-left screen contains a space, then ABC; these appear as their ASCII values 32, 65, 66, and 67. When the VIC-II chip generates the TV scan for the top lines, it looks at screen RAM for the current character, multiplies this by 8, and offsets the result from the character-definition start. This takes it to the eight bytes which define the character. The byte selected depends on the line being scanned; the first line gets the first byte, the second line the second byte, and so on. Whether a pixel is set depends on the bit value within the byte, and also on the color information, which VIC-II takes from color RAM. All this intricate work is performed by the VIC-II.

To illustrate, consider a CHR$(32) at the top left of the screen. The VIC-II calculates 8*32=256, and offsets this from the character base, which typically is $1000 as the chip sees it. The bytes read are 256–263 (decimal) locations away from $1000. They hold the 64 bits of information needed to define CHR$(32). Normally, these bits are all 0, since CHR$(32) appears as the space character. Similarly, any number from 0 to 255 in screen RAM is automatically converted into its bit equivalent. It's clear that a full 256 characters aren't *necessary;* a game, for example, might have a number of frogs, aliens, spaceships, or whatever, some made up from several graphics to increase the size, without using the full set.

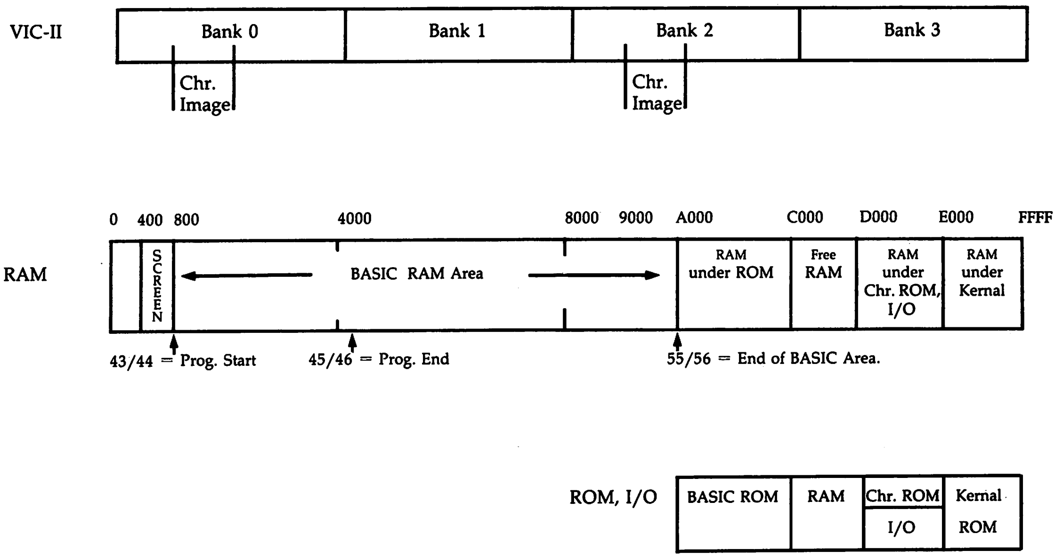
Figure 12-4. VIC-II Screen and Character Management



Memory Maps of the 64 with User-Defined Graphics

The memory map in Figure 12-5 shows all the 64's memory features needed to design your own characters and incorporate them in your programs. Advanced methods are possible—the use of several screens, allowing animation or instantaneous switching between displays, the use of several character sets, and the use of interrupts for split-screen effects—and they rely on the same principles.

Figure 12-5. User-Defined Graphics Memory Maps



VIC-II's current bank at any one time must contain screen, character, and sprite data. But BASIC normally starts at $0800 and ends where there's hardware, either at $9FFF or, if an autorun cartridge is present which returns to BASIC, at $7FFF. Together, these facts mean that a long BASIC program with user-defined characters *must* either store characters at the low end of BASIC, followed by BASIC itself, starting higher in RAM than usual; store them after the BASIC program, but before BASIC's area for variables; or store them higher in memory, in banks 1, 2, or 3. In any of these cases, it's necessary to alter some of the pointers in (43–44), (45–46), and (55–56) to alter the boundaries allocated to BASIC, unless the characters are stored under ROM, with the screen RAM at or near $C000. (This is quite simple, though it's best to avoid the I/O area. Note that there's a possible problem if you use the area to edit characters, because, being under ROM, they're difficult to PEEK in the usual way, and are therefore difficult to save; VIC-II, of course, is designed to PEEK them without any problem.)

User-Defined Graphics Examples

Bank switching. Bits 0 and 1 in $DD00 (56576) control bank switching. Strictly, bits 0 and 1 in $DD01 (56577) could also be set to 1, but as they are 1 already unless specifically changed, these bits can usually be left to take care of themselves. Enter and run the following one-line program:

```
0 INPUT X: X=3-X: POKE 56576,(PEEK(56576) AND 252) OR X: GOTO 0
```

Entering 0 selects the normal default, bank 0, so there's no change. Note the use of 3−X which converts 0 to 3; this allows for the fact that the two controlling bits are inverted.

Entering 2 gives garbage, because bank 2 now treats RAM at $8400 as the start of screen. The characters are displayed as normal 64 characters, however, because the ROM character set is active in this bank. Entering 1 or 3 displays a screenful of characters which aren't properly defined—they're from RAM starting at $5000 or $D000, respectively. Try exiting the program and executing FOR J=33792 TO 34047: POKE J,X: X=X+1: NEXT which puts 0–255 into $8400 on. You'll find that bank 2 now displays all 256 normal characters at the top of the screen, color RAM permitting.

Moving the screen. The following line will move the screen in sixteen 1K steps within its current bank (insert a value in the range 0–15 for X):

```
POKE 53272,(PEEK(53272) AND 15) OR 16*X
```

The two lines below will allow you to instruct VIC-II to treat $0, $0400, $0800, and so on, as the screen start. Enter 2, for example, to select $0800. You'll see the BASIC program at the top of the screen. (SHIFT–Commodore key may make it clearer.)

```
0 INPUT S: POKE 53281,247: REM MAKE CHRS VISIBLE
1 POKE 53272,(PEEK(53272) AND 15) OR 16*S: GOTO 0
```

Enter 0 to make the screen from the zero page on, so the region including the stack and the keyboard buffer is displayed. Entering 1 returns the screen to the normal position.

Moving the character definitions. Enter and run the following line:

```
0 INPUT C: POKE 53272,(PEEK(53272) AND 240) OR 2*C: GOTO 0
```

As C varies from 0 through 7, the character definitions are taken by VIC-II from $0, $0800, $1000, and so on, in 2K steps. Two interesting examples are C=0 and C=3: the former draws character definitions from the zero page, which means that some of them continually fluctuate with background BASIC activity. When C is 3, characters start from $1800, so the screen goes into lowercase mode. Toggling with SHIFT–Commodore key alternates with uppercase.

Bank 0 characters can't occupy $1000–$1FFF, since VIC-II fetches data in this area from ROM. They could be positioned anywhere else, subject to some restrictions. RAM from $0800 on can be used, but the start of BASIC would have to be moved up; $00 is usable, too, but an entire set of 256 characters couldn't be defined. Bank 1 characters can be placed in any of the eight available areas. Bank 2 is acceptable except for $9000–$9FFF, where VIC-II uses the ROM sets. Bank 3 is all available, though $D000–$DFFF RAM can be POKEd only after switching out I/O.

Moving the character ROM into RAM. This requires care to avoid I/O and ROM selection clashes by VIC-II. Bit 2 of location 1 controls whether ROM or I/O appears from $D000 on. As an example, delete GOTO 0 from line 0 directly above, and add the following:

```
1 POKE 56333,127: POKE 1,51
2 FOR J=0 TO 2047: POKE 12288+X,PEEK(53248+X): X=X+1: NEXT
3 POKE 1,55: POKE 56333,129
4 POKE 56,48: CLR
```

Line 2 moves 2K of character ROM from $D000–$D7FF down to $3000–$37FF. Lines 1 and 3 control the hardware line which turns on the ROM. Note that one of the POKEs turns off IRQ interrupts, which is necessary since they make use of the I/O chips. NMI interrupts aren't turned off; when using this, *don't* press RUN/STOP–RESTORE, or you'll crash the program. Line 4 lowers the top of BASIC to $3000, protecting the character set in RAM, and incidentally dramatically reducing the free bytes available to BASIC.

Run the program, entering 6 to point the VIC-II to fetch its characters from $3000. When *READY* comes back, toggling with SHIFT–Commodore key produces garbage characters—since the lowercase set is not copied.

Toggle back to the readable characters and enter the following direct mode line:

```
FOR J=0 TO 255: FOR K=0 TO 7: POKE 14336+8*J+K,PEEK(12288+8*J+7-K): NEXT:
NEXT
```

This copies our RAM characters into $3800 on, but inverts them. Now SHIFT–Commodore key toggles between ordinary and upside-down characters. Backward and other modified characters are also feasible. It's possible to have 80-column alphanumerics on the 64. Unfortunately, they're truly readable only with a monitor, not a TV.

Saving and reloading character definitions. Character definitions (and screens) can be saved as DATA statements or as sequential files (written by PRINT#, read back by INPUT#) or, most efficiently, as program files, that is, as a block of data to be loaded back, typically with a forced (nonrelocatable) LOAD like this at the start of a program:

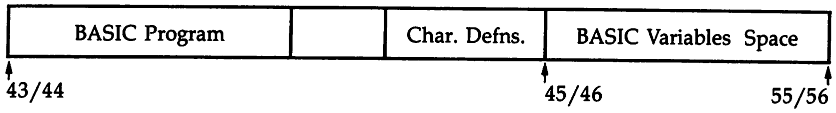
```
0 IF X=0 THEN X=1: LOAD "CHRS",8,1.
```

Chapter 6 explains how to save blocks of data; one method is used in the character editor program that follows.

If you're familiar with the internal workings of BASIC, you'll appreciate that characters can be saved along with BASIC, as a single program, which makes a convenient package. All that's necessary is to manipulate the pointers at 43–44 and/or 45–46, which mark the limits of the program which will be saved to tape or disk, and/or the pointer at 55–56 to the end of RAM allocated to BASIC.

One way of including characters in BASIC is to follow the program almost immediately with character definitions—there must be a 2K boundary, however. This is mapped in Figure 12-6 below:

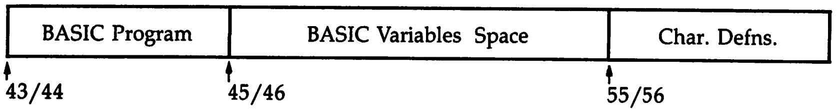
Figure 12-6. Character Definitions Below BASIC Variables



The principle is exactly the same as saving BASIC followed by ML. All that's needed is to alter the address in (45–46) to point after the characters. This will save satisfactorily, and will also run properly when reloaded, assuming it loads back into the same RAM area.

Figure 12-7 shows characters at the end of BASIC. This is a typical situation when bank 0 is used, and character definitions are stored from $3000 (if there are two sets) or $3800 (if there's just one).

Figure 12-7. Character Definitions Above BASIC Variables



It's easy to save such a program: just POKE 45,0: POKE 46,64 to force the 64 to save from BASIC start, right up to $4000. On LOAD, this program will store its variables *after* BASIC and will work perfectly well, becoming converted into the first type of program/character definitions. In fact, it may well have more RAM than it would have if its variables were forced to exist below the characters. But it makes sense, particularly with tape, to keep the characters after BASIC without too great a gap, as in the first example; saving right up to $4000 means saving about 14K bytes.

The only problem where redefined characters are saved above BASIC and variables is that editing BASIC will probably disrupt the display. Changing a line or two moves the whole character set in RAM, so the characters alter with the program length. Obviously, this doesn't matter with a finished program, but where it's important to be able to edit, include this line:

```
0 POKE 45,000: POKE 46,000: POKE 55,0: POKE 56,48: CLR
```

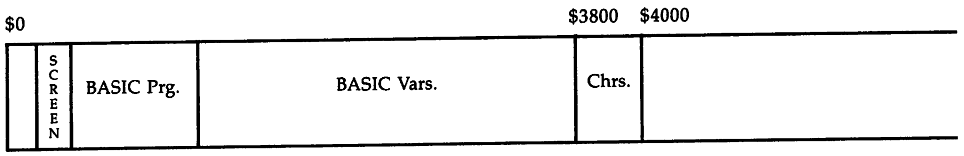
This will allow you to put different values into 45 and 46. (They must be put in just before saving the program. The three zeros allow any figure to be input without changing the program's length, 013 for 13, for example.)

Memory Configurations with User-Defined Characters

The examples show how to go about arranging memory as you choose. The simplest method stores characters below $4000, so bank switching isn't needed.

VIC-II in bank 0. Single character set starting at $3800. This example shows BASIC's variables stored below the character set; they could just as easily be stored above, by putting POKE 45,0: POKE 46, 64: CLR in the program, to move the end-of-program up to $4000.

Figure 12-8. VIC-II Bank 0, Characters at $3800



Program 12-23. Mosaic

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 FOR J=0 TO 39:READ X:POKE 14336+J,X:NEXT
                                          :rem 152
20 POKE 646,0:POKE 53280,1:POKE 53281,1:PRINT "
   {CLR}";                                :rem 247
30 POKE 53272,31                          :rem 37
40 POKE 55,0:POKE 56,56:CLR               :rem 223
50 GET Q$:IF Q$="" GOTO 50                :rem 23
60 L=ASC(Q$)-64                           :rem 35
70 A$="": FOR J=1 TO L: A$=A$+CHR$(RND(0)*5+64):NE
   XT                                     :rem 165
80 A$=A$+A$+A$+A$+A$+A$+A$+A$:M=LEN(A$)   :rem 184
90 PRINT "{HOME}";:FOR J=1 TO 960/M:PRINT A$;:NEXT
                                          :rem 255
100 PRINT LEFT$(A$,960-M*INT(960/M));     :rem 36
110 GOTO 50                               :rem 48
500 DATA 36,36,255,0,0,255,36,36          :rem 31
501 DATA 36,36,231,36,36,231,36,36        :rem 134
502 DATA 36,18,9,132,66,33,144,72         :rem 90
503 DATA 36,72,144,33,66,132,9,18         :rem 91
504 DATA 36,66,153,36,66,36,153,66        :rem 152
```

Program 12-23, above, is memory-mapped as Figure 12-8 shows. It uses only five characters, which print as @, A, B, C, and D, and PEEK as 0–4. (Try RUN/STOP, then type keys @ through D.) Line 10 POKEs in the characters' bit patterns; line 30 points VIC-II's character base to $3800 and line 40 puts BASIC's topmost byte just below $3800.

VIC-II in bank 3. Character sets under Kernal ROM. The screen RAM is moved to $CC00, and the character sets to $E000–$EFFF. This requires POKE 648,204: POKE 53272,57: POKE 56576, PEEK(56576) AND 252 to change the VIC-II chip, and also the following:

0 POKE 56333,127: POKE 1,51: FOR J=0 TO 4095
1 POKE 57344+J,PEEK(53248+J): NEXT: POKE 1,55: POKE 56333,129

After this, characters are taken from $E000 on. They can be altered by POKEing, but cannot easily be read back. The usual screen area from $400 to $7FF is now unused, and RAM from $C000 to $CBFF can be used for ML. No BASIC pointers need be set since the characters aren't in a part of RAM which BASIC can overwrite.

Switching between several screens in bank 1 for animation. Sixteen full screens can be stored between $4000 and $7FFF. Switching between them requires one POKE, and the effect is instantaneous, allowing animation. The screens share color RAM, so the easiest arrangement is to keep the color identical in all the screens. The difficult part is the design of all the screens.

Character Editor

The following character editor, Program 12-24, processes 2K bytes, the definitions of the first 256 characters in ROM, which are stored at $3000–$37FF (12888–14335). The finished definitions can be stored and reloaded from tape or disk, giving a permanent record of your new characters.

Program 12-24. Character Editor

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
6 POKE 56,48: CLR: SCREEN=1064: VIC=53248: CHARS=1
  2288                                        :rem 209
8 DEF FN RV(Q)=Q+128*(1+(Q>127)*2)            :rem 230
11 DIM CO$(15):FOR J=0 TO 15:READ CO$(J):NEXT
                                              :rem 170
12 FOR J=49152 TO 49213:READ X:POKE J,X:NEXT
                                              :rem 218
18 INPUT "FETCH CHARACTERS FROM ROM";X$        :rem 96
19 IF X$="Y" THEN GOSUB 500                   :rem 147
22 POKE 646,0:POKE 650,128:PRINT "{CLR}"      :rem 248
27 FOR J=0 TO 127:POKE SC+2*J+401,J:POKE SC+J*2+72
   1,J+128:NEXT                               :rem 148
40 POKE VIC+24,(PEEK(VIC+24)AND240) OR 12     :rem 217
100 NC=0:OF=NC*8:GOSUB 3000                    :rem 11
140 POKE 53280,5:POKE 53281,5:POKE 53282,2:rem 191
146 POKE 53283,8:POKE 48197,0:SYS 49188:GOSUB 2000
    0                                         :rem 248
180 XC=0:YC=0:CC=SC:POKECC,PEEK(CC)OR128      :rem 202
200 GET X$:IFX$="" THEN 200                   :rem 117
202 IF X$="{UP}" AND YC> 0 THEN YC=YC-1: GOTO 250
                                              :rem 154
204 IF X$="{RIGHT}" AND XC<39 THEN XC=XC+1: GOTO 2
    50                                         :rem 93
206 IF X$="{DOWN}" AND YC<23 THEN YC=YC+1: GOTO 25
    0                                          :rem 79
208 IF X$="{LEFT}" AND XC> 0 THEN XC=XC-1: GOTO 25
    0                                         :rem 169
218 IF X$="{HOME}" THEN GOSUB 4000: GOTO 180
                                              :rem 185
220 IF X$=" " THEN GOSUB 2000: GOTO 200       :rem 150
226 FOR Z=1 TO 8:IF X$<>MID$("{F1}{F3}{F5}{F7}{F2}
    {F4}{F6}{F8}",Z,1) THEN NEXT               :rem 11
228 CB=Z-1:ON Z GOSUB 21000,21000,21000,21000,1100
    0,12000,13000,14000                       :rem 149
229 IF Z>0 AND Z<9 GOTO 200                    :rem 97
```

```
236 FOR Z=1 TO 7:IF X$<>MID$("=DCIMT",Z,1) THEN NE
    XT                                         :rem 117
237 ON Z GOSUB 6000,7000,9000,9500,10000,10500
                                               :rem 250
239 IF Z>0 AND Z<8 GOTO 200                     :rem 97
246 IF X$="S" THEN GOSUB 16000:GOTO 22         :rem 248
247 IF X$="L" THEN GOSUB 18000:GOTO 22         :rem 244
248 GOTO 200                                   :rem 105
250 GOSUB 4000:CC=XC+YC*40+SC:GOSUB 4000:GOTO 200
                                                :rem 12
387 DATA "BLACK ","WHITE ","RED{3 SPACES}","CYAN
    {2 SPACES}","PURPLE"                       :rem 124
388 DATA "GREEN","BLUE","YELLOW","ORANGE","BROWN "
    ,"LT RED"                                  :rem 101
389 DATA "GRAY 1","GRAY 2","LT GRN","L BLUE","GRAY
     3"                                        :rem 236
390 DATA 120,162,8,160,0,169,208,133,252   :rem 171
391 DATA 169,48,133,254,169,0,133,251,133 :rem 235
392 DATA 253,177,251,145,253,200,208,249   :rem 188
393 DATA 230,252,230,254,202,208,242,88,96 :rem 28
395 DATA 0,169,216,133,252,169,0,133,251   :rem 179
396 DATA 162,4,160,0,173,35,192,145,251    :rem 129
397 DATA 200,208,248,230,252,202,208,243,96:rem 75
500 POKE 56333,127:POKE 1,51:SYS 49152     :rem 248
540 POKE 1,55:POKE 56333,129:RETURN        :rem 222
2000 IF YC>7 OR XC>7 THEN RETURN            :rem 34
2010 PT=CH+YC+OF:BY=PEEK(PT):MS=2↑(7-XC)  :rem 217
2030 POKE CC,382-PEEK(CC)                  :rem 108
2040 IF (BY AND MS)=0 THEN POKE PT,BY+MS:GOTO 2060
                                               :rem 240
2050 POKE PT,BY-MS                          :rem 46
2060 POKE 214,YC:PRINT:PRINTTAB(9)"{3 SPACES}
     {4 LEFT}";PEEK(PT)                      :rem 82
2070 RETURN                                 :rem 169
3000 PRINT "{HOME}":FOR Y9=0 TO 7:PT=CH+Y9+OF:BY=P
     EEK(PT)                                :rem 240
3020 FOR X9=0 TO 7                          :rem 132
3030 IF (BYAND2↑(7-X9))>0 THEN PRINT "Q";:GOTO 304
     0                                      :rem 158
3035 PRINT "-";                               :rem 4
3040 NEXT:PRINT "{4 SPACES}{4 LEFT}"BY:NEXT
                                            :rem 153
3060 PRINT "{HOME}{12 SPACES}SCREEN CODE=
     {3 SPACES}{4 LEFT}" NC                 :rem 202
3070 RETURN                                 :rem 170
4000 POKE CC,FNRV(PEEK(CC)): RETURN          :rem 72
6000 IF YC<9 THEN RETURN                    :rem 117
6010 NC=FNRV(PEEK(CC)):OF=NC*8:GOSUB 3000:RETURN
                                            :rem 180
7000 IF YC>7 OR XC<13 OR XC>27 THEN RETURN :rem 53
7030 POKE CC,FNRV(NC):RETURN                :rem 227
```

```
9000 FOR J=CH+OF TO CH+OF+7:POKE J,0:NEXT:GOSUB 30
     00                                        :rem 215
9025 IF YC<8 AND XC<8 THEN GOSUB 4000          :rem 196
9030 RETURN                                    :rem 172
9500 FOR J=CH+OF TO CH+OF+7:POKE J,255-PEEK(J):NEX
     T                                         :rem 184
9525 GOSUB 3000:IF YC<8 AND XC<8 THEN GOSUB 4000
                                                :rem 70
9530 RETURN                                    :rem 177
10000 V=PEEK(VIC+22)                           :rem 107
10010 IF (VAND16)>0 THEN POKE VIC+22,V AND 255-16:
      RETURN                                    :rem 62
10020 POKE VIC+22,V OR 16:RETURN                :rem 55
10500 IF YC<9 THEN RETURN                      :rem 165
10520 NN=FNRV(PEEK(CC)):OG=NN*8                :rem 101
10530 FOR J=0 TO 7: POKE CH+OG+J,PEEK(CH+OF+J):NEX
      T:RETURN                                 :rem 144
11000 T=(PEEK(53280)+1) AND 15                 :rem 225
11003 POKE 53280,T:POKE 53281,T:GOTO 20000:rem 247
12000 T=(PEEK(53282)+1) AND 15                 :rem 228
12003 POKE 53282,T:GOTO 20000                   :rem 14
13000 T=(PEEK(53283)+1) AND 15                 :rem 230
13003 POKE 53283,T:GOTO 20000                   :rem 16
14000 T=(PEEK(49187)+1) AND 15                 :rem 239
14003 POKE 49187,T:SYS 49188:GOTO 20000         :rem 96
16000 GOSUB 19000:PRINT:SYS 57812NM$,DN        :rem 245
16020 POKE 193,0:POKE 194,48:POKE 174,0:POKE 175,5
      6                                        :rem 190
16040 SYS 62957:RETURN                          :rem 33
18000 GOSUB 19000:POKE 147,0:SYS 57812NM$,DN,0
                                               :rem 237
18033 L9=12288:POKE 781,L9 AND 255:POKE 782,L9/256
                                               :rem 118
18034 SYS 62622:RETURN                          :rem 27
19000 INPUT "{CLR}{9 SPACES}FILENAME";NM$ :rem 156
19005 INPUT "DISK OR TAPE (D/T)";X$           :rem 152
19010 IF X$="D" THEN DN=8:RETURN                :rem 27
19020 DN=1:RETURN                               :rem 22
20000 PRINT "{HOME}{DOWN}"TAB(29)"BG0 "CO$(PEEK(53
      280) AND 15)                              :rem 47
20040 PRINT TAB(29)"BG1 "CO$(PEEK(53282) AND 15)
                                               :rem 206
20060 PRINT TAB(29)"BG2 "CO$(PEEK(53283) AND 15)
                                               :rem 210
20080 PRINT TAB(29)"TXT "CO$(PEEK(49187) AND 15):R
      ETURN                                     :rem 59
21000 IF YC>7 OR XC>7 THEN RETURN               :rem 83
21010 CP=CC AND 2046:IF CB>1 THEN POKE CP,81:GOTO
      {SPACE}21030                             :rem 149
21020 POKE CP,45                                :rem 76
```

```
21030 CP=CP+1: IF(CB AND 1)=1 THEN POKE CP,81:GOTO
      21050                                 :rem 183
21040 POKE CP,45                             :rem 78
21050 GOSUB4000:PT=CH+YC+OF:BY=PEEK(PT)     :rem 149
21060 MP=7-XC AND 6:MK=2↑MP*3:BY=(BY AND NOT MK) +
      CB*2↑MP                               :rem 213
21070 POKE PT,BY:POKE 214,YC:PRINT           :rem 35
21080 PRINT TAB(9)"{3 SPACES}{4 LEFT}" PEEK(PT):RE
      TURN                                  :rem 213
```

When you run the character editor, you're asked FETCH CHARACTERS FROM ROM? Enter Y to call an ML routine to copy character ROM into RAM. Otherwise, characters already present at $3000 are retained. The screen displays all 256 of the RAM characters in the bottom half of the screen. The top displays an enlarged version of the current character plus its screen code—for example, @ is shown as 0—and a list of the four current color settings. (All character colors are the same.) There's also a central scratch-pad area where several graphics can be placed so that their joint effect can be checked. This is useful where several graphics characters together build a larger composite character.

Editing is done by moving the inverted cursor with the usual cursor control keys on the enlarged 8 × 8 diagram, and typing the space bar to invert the dot beneath the cursor. The result of the modifications is instantly visible in the display at the screen bottom and on the scratch area.

New characters are selected for editing by moving the cursor to the lower part of the screen and typing an equal sign (=) over the chosen character. Then home the cursor and edit.

Other commands are:

C	Clears a character, setting all bits 0.
D	Draws the current character in the scratch area.
I	Inverts the current character's bits.
T	With the cursor over a character in the bottom half of the screen, transfers the current character's definition there—now there are two of them.
f2	Advances the screen background color.
f8	Advances the text color.
M	Toggles between ordinary and multicolor modes. The text color must be from orange to gray 3 for multicolor mode to show. When it does, keys f1, f3, f5, and f7 set pairs of points to 00, 01, 10, and 11, corresponding to the four colors. Keys f4 and f6 also advance background colors 1 and 2.
L and S	Load and save (respectively) the character set, allowing your choice of filename and device.

Applications could include the creation of chess and other game pieces, special alphabet sets, musical and other notations, monsters, bombs and so on, for arcade games. If you're going to PRINT your graphics, remember that the second batch of 128 characters on the screen is generated with {RVS}, even though characters need not appear in reverse.

The early part of the program handles initialization. Lines 200–250 handle the keypresses and call appropriate subroutines.

Bitmapped Graphics

Bitmap mode gives the best graphics available on the 64. However, 64 BASIC has no specific commands to handle this mode: in fact, it would be possible to program a 64 for years and never discover that bitmap mode existed.

Bitmapping allows each dot on the screen to be controlled. Since VIC-II maps the screen into individual characters of 8 × 8 pixels, and displays 40 × 25 of these characters, there are 320 × 200 = 64000 addressable dots. (In practice, resolution isn't this good, because, for example, ordinary TVs cannot display alternate on/off dots without color interference, and some color combinations don't have sufficient contrast to be properly distinguishable, even on color monitors.) These figures still apply if the screen is narrowed or shortened—see the earlier section on smooth scrolling—or mixed with ordinary text by the use of split-screen techniques explained below, but offscreen graphics are obviously less important.

Bitmapping is a more accurate expression than high resolution, with which it's often confused. Bitmapping resolution is in fact identical to that of normal characters. The distinction should be between high resolution and multicolor mode.

A bitmap is 8000 consecutive bytes (not quite 8K, which is 8192 bytes), enough to map the whole screen as 64,000 bits. The display is treated by VIC-II just like 1000 consecutive user-definable characters. Bitmap mode is selected by bit 5 in 53265 ($D011).

In bitmap mode, since each bit in the bitmap can only be on or off, just two alternatives exist for each point on the screen, which means a choice of two colors. VIC-II allows each of the 1000 characters 2 independent colors, selectable from the full range of 16 colors. In each bitmapped character, the 2 colors are *not* set in color RAM, which has only one usable nybble. Instead, they're controlled by screen RAM, the area, usually starting at $400, treated by BASIC as the screen, which of course has two nybbles available for each character. This new usage can be confusing. Unlike all other modes, the screen's normal background color setup is no longer operative.

You don't control the color of *every* dot on the screen, though. One reason is the memory requirement: a choice of 16 colors per bit would require 32K of RAM to store the full bitmap.

Multicolor mode is selected by bit 4 in 53270 ($D016). Where multicolor mode is used with bitmap mode, there's the usual trade-off—pairs of bits together allow a choice of 4 colors. Each of the 1000 characters has a choice of 3 colors, plus a common background color. All the colors are selected from the full palette of 16 colors.

Any 64 graphics design program, and generally flight-simulators and games where the entire screen is filled without repetition, must be bitmapped. (Sprites can sometimes give a similar impression, though.) The high-resolution bitmap mode has finer resolution than the multicolor version, but is less colorful—except in the sense of being more prone to unwanted color fringing. For example, you may find an adventure game including black line drawings on white, which are colored by a fill-in color (unless they're on a boundary), since three colors can't coexist in one 8 × 8 area. Multicolor mode builds the picture from 160 short horizontal bit pairs by 200 down. This is more versatile than regular 64 multicolor graphics; the background color is in common, but all the other three colors are independently variable within every 8 × 8 block. This allows the character boundaries, where colors can change, to be made imperceptible, at the cost of some loss in resolution.

The VIC-II Chip and Bitmapping

Everything about the VIC-II chip in the previous section applies to bitmap mode, except that the character base can only be made to start at 8K intervals (at $0, $2000, $4000, $6000, . . . $E000). Since 8000 bytes are necessary for a bitmap, this is reasonable enough. There are eight possible positions for the bitmap area, and therefore two possible positions within each of the four VIC-II banks. Note that bits 1 and 2 in $D018 have no effect; it's not possible to adjust the bitmap character base to the nearest $800 bytes.

Screen RAM is controlled by VIC-II as in character mode, but interpreted differently, as color information. With BASIC, this gives odd effects, as we'll see. Meanwhile, BASIC bitmap programs under development should be run from time to time with the bitmap POKEs REMed out, since syntax errors won't be readable in bitmap mode. Otherwise, you may not even know that a syntax error has occurred. Remember also that if you've used RUN/STOP–RESTORE to get back to normal, POKE 648,4 will be needed if the screen was moved.

How to Use Bitmapping

Bitmap mode is interesting, but needs to be approached with caution. Here is a brief discussion of when its use is appropriate.

Normal BASIC screen commands don't work in bitmap mode, since the characters and screen are organized differently. Therefore, POKEs and often PEEKs are essential. The 8 × 8 organization means it's essential to go through a conversion process. If you want to locate a pixel 100 dots across and 50 dots down the screen, it must be translated into the bit definition corresponding to the twelfth character across and seventh down the screen; then the byte and its bit have to be found. This illustrates that ML is necessary if you cannot tolerate delays. BASIC is too slow even to clear the bitmapping screen satisfactorily.

Bitmapped pictures can be loaded from disk and tape. All that's needed is to first save the relevant 8K bytes and then save their screen RAM for color information. Later a forced LOAD of both, with VIC-II set, reconstructs the picture. This works well; a disk can hold about 18 such pictures. Where this isn't suitable, the bitmap can be loaded with the program. If movement is wanted, the 8K bitmap itself must be processed, probably with ML.

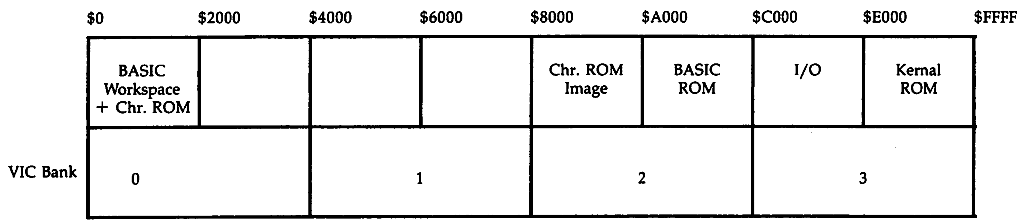
Fairly simple pictures can be effective. The most efficient method is to draw the figures while the program runs. Fairly simple line-segment drawings can be composed like this more quickly than loading 8K of data. Subroutines to draw a line segment between two points and to color an enclosed region are often used.

Several bitmaps can coexist in RAM; switching between them requires a simple POKE, which can also change the color RAM. So it's possible to store two or more completely independent pictures and instantly switch from one to the other. However, this uses a lot of memory.

Memory Maps in Bitmap Mode

Figure 12-9 shows all eight available bitmap positions. Two of these can't be used for full-screen bitmaps, since the ROM characters appear, immovably, in the lower half of the screen. Because $C000–$DFFF occupies what could be useful free RAM and also risks conflict with I/O, it is easier to use another location.

Figure 12-9. Available Bitmap Addresses



Of the five other regions, $2000–$3FFF involves no bank switching and is easiest, but the others aren't much more difficult. Remember that each bitmap screen must have 1000 bytes in the same bank for its color. So if $A000 onward holds the bitmap, the colors must start somewhere like $8C00, avoiding character-ROM images.

Bitmaps can be tucked away below ROM. VIC-II uses these regions happily. The drawback is that they can't be PEEKed without switching ROM out; this makes graphics which need to be altered (for example, to give the effect of motion) slightly trickier.

Note that $4000–$7FFF can't be used to store two full bitmaps, because there's no place to put the color screens in the same bank. If you want to switch between full screens, it's therefore necessary to have bitmaps and color RAM in two or more different banks. If the positions are similar in each, a single POKE to bank select will switch between them. This applies to full screens. However, it's possible to overlap the color with the bitmap—for example, by starting the bitmap at $2000 and the screen at $3C00. At the screen bottom, 104 characters (about 2-1/2 lines) will echo the colors in bitmap form. This is acceptable with split-screen techniques. It's the only way two full bitmaps can occupy the same VIC bank.

Calculations

One way to visualize bitmapping is to imagine that all 8000 bytes are divided into 25 sets of 320 bytes. Each 320-byte block corresponds to a horizontal line, eight dots high, on the screen. Another way to visualize bitmapping is to consider the information as 1000 eight-byte chunks of memory-defined characters 0–999 in the familiar 40 × 25 layout.

To control a screen dot with given X and Y coordinates, we have to determine which bit of which byte to process. Let's consider X and Y relative to the top left of the screen, with X=0–319 and Y=0–199. The object is to calculate where a particular point, say, X=100, Y=50, will be. Points with Y from 0 to 7 lie in the top row of characters; Y from 8 to 15 must be in the next row, and generally Y has INT(Y/8) complete pixel rows above it, each of 320 bytes. Now, points with X from 0 to 7 fall within the first character in any row; X from 8 to 15 corresponds to the next character, and so on. Generally, the number of characters along a row is INT(X/8). The point can be in one of eight bytes in the character, determined by the remainder after dividing Y by 8; in BASIC, this is Y AND 7 for our range of Y values. If MAP is the variable storing the start of the bitmap, this is the address of the byte containing X,Y:

```
MAP + 320*INT(Y/8) + 8*INT(X/8) + (Y AND 7)
```

This expression can be improved to the following line, which BASIC evaluates faster:

```
MAP + 40*(Y AND 248) + (X AND 504) + (Y AND 7)
```

Finally, the actual bit within the target byte is 7 − (X AND 7), because X AND 7 gives 0–7, increasing with X, but the bits are arranged in the sequence 7–0. Bit 0–7 has to be set or cleared to set or clear the screen pixel, with:

```
POKE AD, PEEK (AD) OR 2↑(7−(X AND 7)) : REM SETS PIXEL
```

The same expression with AND clears the bit.

Examples Using Bitmap Mode

A bitmapped window. Turn on the 64, then enter and run this line:

```
10 POKE 53265, PEEK(53265) OR 32: REM SET BITMAP MODE
```

The 64 displays the first 8000 bytes, from $0, in bitmap mode. Character definitions, seen by the VIC-II at 4096 and following, are displayed in the bottom half of the screen. The zero page and stack are displayed at the top of the screen, so some of these locations continually change. The screen RAM, at $400–$7E7, is displayed in the top eighth to quarter of the screen; you'll see changes in the display if you cursor around the screen and type keys. Note that the colors are mainly red and black because spaces PEEK as 32 (=%0010 0000), so the high nybble is red, the low nybble black. Nonspace characters appear in colors depending on the characters' PEEK values.

Multicolor mode is more complex. If you select it, you'll see the common blue background, the mode extending over the whole screen, and the light blue of the ordinary color RAM.

Bitmapping at $6000. Add these lines to the above example to alter the bitmap and color locations:

```
20 POKE 56576,150: POKE 648,92: POKE 53272,121: REM BITMAP PARAMETERS
30 FOR J=6*4096 TO 6*4096+7999: POKE J,1: NEXT: REM POKE BITMAP
40 FOR J=23553 TO 24551: POKE J,1: NEXT:REM POKE COLOR
```

Line 20 starts the bitmap at $6000, in bank 1, and starts its color just below at $5C00. Line 30 fills the bitmap with 1, giving 40 fine vertical lines on the screen. Line 40 sets the colors to black and white. This part is faster, by eight times, than filling the screen. (Note that line 30 could clear the screen by POKEing in 0 or 255. Random numbers would fill the screen with random dots.)

After adding the following line, BASIC strings will move down to overwrite the bitmap, then the color, giving textilelike patterns.

50 X$="": FOR J=1 TO 7: X$=X$+CHR$(256*RND(1)): X$=X$+X$: NEXT: GOTO 50

Inserting a line like 45 POKE 55, 0: POKE 56, 92: CLR prevents this. This sets the top of BASIC memory at $5C00, protecting the color and bitmap information.

Bitmapping at $2000. Program 12-25 POKEs 5 into the color area, setting colors to green (since the low nybble is 5, which determines the color of bits cleared to 0) and black (since the high nybble is 0, for bits set to 1). Lines 50 to 70 scan across the screen, plotting dots. Note how the Y value is forced into the range 0–200. This can be made automatic by picking out the maximum and minimum within a loop.

Program 12-25. Bitmap Draw Routine

```
1Ø POKE 53265,PEEK(53265) OR 32
2Ø POKE 53272,25:MAP=8192
3Ø FOR J=MAP TO MAP+7999:POKE J,Ø:NEXT
4Ø FOR J=1Ø24 TO 2Ø23:POKE J,5:NEXT
5Ø FOR X=Ø TO 319:Y=SIN(X*↑/8Ø)*5Ø+1ØØ
6Ø AD=MAP+4Ø*(Y AND 248) + (X AND 5Ø4) + (Y AND 7)
7Ø POKE AD,PEEK(AD) OR 2↑(7-(X AND 7)):NEXT
8Ø GET R$:IF R$="" THEN8Ø
```

Program 12-25 puts the bitmap at $2000, but keeps the BASIC screen RAM so *READY* prints as colored blocks. Try LIST when the program has finished running (drawing a figure on the screen); the bitmapped dots remain, like a sprite, as the screen scrolls. This happens whenever BASIC's screen shares the color area. The key E which has a PEEK value of 5 gives the same black on green color effect.

Memory map examples. Bank 0 isn't very suitable for bitmapping. The range $2000–$4000 has to be used for the bitmap, and $1000–$1FFF is filled with character ROM. Therefore, if the color isn't to coincide with BASIC's screen, it must start at $0C00, leaving only 1K if BASIC starts in its usual place.

To set this configuration, POKE 53272,57: POKE 55,0: POKE 56,12: CLR, assuming bank 0 is on.

However, BASIC's start can be moved up to $4000, with POKE 43,1: POKE 44,64: POKE 16384,0:NEW. To move the end of BASIC to $A000, POKE 55,0: POKE 56,160: CLR. This provides 24K of RAM available for BASIC, the maximum possible with the bitmap and color both in free RAM. If you're writing BASIC and don't want a loader to first reconfigure BASIC, the setup with 21K for BASIC is better.

Drawing lines. Program 12-26 draws black lines on a white bitmapped screen; an ML routine (by B. Grainger) plots individual points, and this is called by a BASIC routine which calculates optimum points to generate straight lines.

Program 12-26. Drawing Lines

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
1Ø FOR J=49152 TO 4931Ø:READ X:POKE J,X:NEXT
                                           :rem 214
2Ø POKE 56576,(PEEK(56576) AND 252) OR 2  :rem 221
3Ø POKE 53272,9                            :rem 25Ø
4Ø POKE 53265,PEEK(53265) OR 32             :rem 69
```

```
50 POKE 56,64:CLR                                    :rem 176
60 CLEAR=49152:PLOT=49197                            :rem 211
100 SYS CLEAR                                        :rem 247
110 X2%=X1%:X1%=RND(0)*320:Y2%=Y1%:Y1%=RND(0)*200
                                                     :rem 146
120 GOSUB 1000:GOTO 110                              :rem 217
1000 XD%=X1%-X2%                                      :rem 73
1010 IF XD%>0 THEN XSH%=-1:GOTO 1100                 :rem 151
1020 XD%=-XD%:XSH%=1                                 :rem 111
1100 YD%=Y1%-Y2%                                      :rem 77
1110 IF YD%>0 THEN YSH%=-1:GOTO 1200                 :rem 155
1120 YD%=-YD%:YSH%=1                                 :rem 115
1200 XC%=X1%:YC%=Y1%                                  :rem 85
1210 YC%=Y1%                                         :rem 113
1220 IF XD%<YD% THEN 1500                              :rem 8
1222 ACC%=-XD%/2                                      :rem 63
1225 IF XC%=X2% THEN 2000                            :rem 246
1230 SYS PLOT,XC%,YC%                                :rem 221
1240 XC%=XC%+XSH%                                    :rem 199
1250 ACC%=ACC%+YD%                                   :rem 202
1260 IF ACC%<=0 THEN 1225                            :rem 230
1270 ACC%=ACC%-XD%                                   :rem 205
1280 YC%=YC%+YSH%                                    :rem 206
1290 GOTO 1225                                       :rem 207
1500 ACC%=-YD%/2                                      :rem 63
1510 IF YC%=Y2% THEN 2000                            :rem 245
1520 SYS PLOT,XC%,YC%                                :rem 223
1530 YC%=YC%+YSH%                                    :rem 204
1540 ACC%=ACC%+XD%                                   :rem 203
1550 IF ACC%<=0 THEN 1510                            :rem 229
1560 ACC%=ACC%-YD%                                   :rem 208
1570 XC%=XC%+XSH%                                    :rem 205
1580 GOTO 1510                                       :rem 206
2000 SYS PLOT,X2%,Y2%:RETURN                         :rem 209
30000 DATA 169,0,133,187,169,96,133,188,160,0,169
                                                     :rem 122
30001 DATA 0,145,187,200,208,251,230,188,165,188
                                                      :rem 59
30002 DATA 201,128,208,241,169,64,133,188,169,1,14
      5                                              :rem 211
30003 DATA 187,200,208,251,230,188,165,188,201,68
                                                     :rem 116
30004 DATA 208,241,96,32,121,0,32,253,174,32,138
                                                      :rem 46
30005 DATA 173,32,247,183,132,176,133,177,32,121
                                                      :rem 58
30006 DATA 0,32,253,174,32,158,183,134,187,169,0
                                                      :rem 61
30007 DATA 133,188,234,169,96,133,140,165,187,41
                                                      :rem 77
30008 DATA 7,133,139,69,187,162,3,10,38,188,202,20
      8                                              :rem 218
```

```
30009 DATA 250,133,187,101,139,133,139,165,188,101
                                              :rem 162
30010 DATA 140,133,140,165,187,162,2,10,38,188,202
                                              :rem 143
30011 DATA 208,250,101,139,133,139,165,188,101,140
                                              :rem 147
30012 DATA 133,140,165,176,41,7,170,69,176,101,139
                                              :rem 158
30013 DATA 133,139,165,177,101,140,133,140,232,56
                                               :rem 97
30014 DATA 169,0,106,202,208,252,1,139,129,139,96
                                              :rem 110
```

Summary of Bitmap Mode

Start of bitmap and color. Bits 0 and 1 of $DD00 select the VIC-II bank. Bit 3 of $D018 defines the start of the bitmap from either the beginning or the midpoint of the selected bank, while bits 4–7 select the main color area.

Mode. Bit 5 of $D011 controls the 64's graphics mode. Therefore, POKE 53265, PEEK(53265) OR 32 selects bitmapping, while POKE 53265, PEEK(53265) AND 223 selects characters.

Bit 4 of $D016 controls the color mode. Thus, POKE 53270, PEEK(53270) OR 16 selects multicolor bitmap mode, and POKE 53270,PEEK(53270) AND 239 selects high-resolution bitmap mode.

Colors. In high-resolution mode, colors are determined by 1000 bytes starting from the area set by VIC-II. The high nybble determines the color of bits set to 1, and the low nybble of bits set to 0, in each 8 × 8 dot area.

In multicolor bitmap mode, one of four colors is chosen according to bit pairs:

11	Low nybble of color RAM from $D800 (55296)
10	Low nybble of screen RAM, often from $0400 (1024)
01	High nybble of screen RAM
00	Low nybble of $D021 (53281), the background color

Drawing Onto the Bitmapped Screen

The pair of programs which follow allow drawings to be made directly to the bitmapped screen, which is set up from 8192 to 16191, with its color from 1024 to 2023. (Finished pictures can be saved, using techniques discussed in Chapter 6.)

The high-resolution version, Program 12-27, is joystick controlled. Dots are plotted in any of eight directions or erased if the delete mode is on. The fire button toggles between plot and delete. If the stick and button are pressed at the same time, a flashing cursor moves without altering the screen. Keys f3 and f5 advance background and foregound colors, respectively.

Program 12-27. Bitmap Drawing with a Joystick

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 POKE 56,8192/256:CLR                       :rem 226
14 POKE 53265,PEEK(53265) OR 32                :rem 70
```

```
15 POKE 53272,PEEK(53272) OR 8                   :rem 22
24 CE=16*Ø + 3                                   :rem 74
25 FOR J=1Ø24 TO 2Ø23:POKE J,CE:NEXT            :rem 23Ø
29 FOR J=49152 TO 49171:READ X:POKE J,X:NEXT
                                                :rem 229
3Ø SYS 49152                                    :rem 1Ø3
8Ø FOR I=Ø TO 7:P2(7-I)=2↑I:NEXT                 :rem 4Ø
9Ø X=16Ø:Y=1ØØ                                  :rem 246
1ØØ PE=PEEK(5632Ø)                              :rem 217
1Ø1 R=PE AND 8                                  :rem 193
1Ø2 D=PE AND 2                                  :rem 174
1Ø3 L=PE AND 4                                  :rem 185
1Ø4 U=PE AND 1                                  :rem 192
1Ø5 B=PE AND 16                                 :rem 228
2ØØ IF B=Ø THEN DEL=1-DEL                        :rem 68
21Ø IF B=Ø AND U*D*L*R=Ø THEN MOV=1              :rem 85
22Ø IF B<>Ø THEN MOV=Ø                          :rem 157
34Ø IF L=Ø THEN X=X-1:IF X<Ø THEN X=Ø           :rem 218
35Ø IF R=Ø THEN X=X+1:IF X>319 THEN X=319       :rem 187
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>199 THEN Y=199       :rem 191
5ØØ OF=4Ø*(Y AND 248)+(X AND 5Ø4) +(Y AND 7)
                                                 :rem 47
51Ø SCREENCHR=INT(OF/8)                         :rem 168
52Ø BIT=X AND 7                                  :rem 21
53Ø CHAR=8192+OF                                :rem 135
54Ø PE=PEEK(CH)                                 :rem 1Ø8
55Ø POKE CH,PEEK(CH) OR P2(BIT)                 :rem 212
56Ø IF DEL THEN POKE CH,PEEK(CH) AND NOT P2(BIT)
                                                :rem 139
57Ø IF MOVE THEN POKE CH,PE: REM RESTORE VALUE IF
    {SPACE}MOVE                                 :rem 145
58Ø IF MOVE=Ø THEN POKE 1Ø24+SC,CE              :rem 1Ø6
6ØØ GET X$                                      :rem 242
61Ø IF X$="{F3}" THEN CE=((CE+1) AND 15) OR (CE AN
    D 24Ø)                                       :rem 63
62Ø IF X$="{F5}" THEN CE=(CE+16) AND 255        :rem 121
7ØØ GOTO 1ØØ                                     :rem 97
2ØØØØ DATA 162,32,138,133,252,169,Ø,133,251,145
                                                :rem 251
2ØØ1Ø DATA 251,2ØØ,2Ø8,251,23Ø,252,2Ø2,2Ø8,246,96
                                                 :rem 92
```

The multicolor mode editor, Program 12-28, is keyboard controlled, using cursor keys rather than a joystick. Four colors are set when the program is run, and the fourth, with bit pattern 11, chosen when plotting starts. Keys 1–4 select current background, high nybble, low nybble, and color RAM colors. Keys f1, f3, f5, and f7 advance the background, high nybble, low nybble, and color RAM, so plotting in different colors on the screen is simple. Note that the background color affects the entire screen—press f1 to see this. Typing the space bar toggles between plot and move modes.

Program 12-28. Multicolor Bitmap Draw Routine

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 POKE 56,8192/256:POKE 55,0:CLR            :rem 17
11 PRINT "BACKGROUND?";:GOSUB 50000:BA=G  :rem 125
12 PRINT "{6 SPACES}AUX1?";:GOSUB 50000:A1=G
                                             :rem 172
13 PRINT "{6 SPACES}AUX2?";:GOSUB 50000:A2=G
                                             :rem 175
14 PRINT " CHARACTER?";:GOSUB 50000:CC=G    :rem 48
15 POKE 53265,PEEK(53265) OR 32              :rem 71
16 POKE 53272,PEEK(53272) OR 8               :rem 23
17 POKE 53270,PEEK(53270) OR 16              :rem 67
20 POKE 53281,BA                             :rem 67
26 FOR J=49152 TO 49205:READ X:POKE J,X:NEXT
                                             :rem 224
28 CE=16*A1 + A2:POKE 49188,CC:POKE 49192,CE
                                              :rem 31
30 SYS 49152                                 :rem 103
80 FOR J=0 TO 7:P2(J)=2↑J:NEXT               :rem 199
90 B=3:X=160:Y=100                           :rem 226
100 GET G$:G=ASC(G$+CHR$(0))                 :rem 218
110 IF G>48 AND G<53 THEN B=G-49:GOTO 500 :rem 103
200 IF G$=" " THEN MOVE=1-MOVE                :rem 69
210 IF G$="{RIGHT}" THEN X=X+2:IF X>318 THEN X=318
                                             :rem 255
220 IF G$="{LEFT}" THEN X=X-2:IF X<0 THEN X=0
                                             :rem 168
230 IF G$="{UP}" THEN Y=Y-1:IF Y<0 THEN Y=0
                                             :rem 160
240 IF G$="{DOWN}" THEN Y=Y+1:IF Y>199 THEN Y=199
                                               :rem 7
250 IF G$="{F1}" THEN BA=(BA+1) AND 15:POKE53281,B
    A                                         :rem 10
260 IF G$="{F3}" THEN A1=(A1+1) AND 15:CE=(CE+16)
    {SPACE}AND 255                           :rem 168
270 IF G$="{F5}" THEN A2=(A2+1) AND 15:CE=(CE+1) A
    ND 255                                   :rem 118
280 IF G$="{F7}" THEN CC=(CC+1) AND 15      :rem 251
500 SCREENCHR=40*INT(Y/8)+INT(X/8)           :rem 31
510 ROW=Y AND 7:BIT=6-(X AND 7)             :rem 154
520 CHAR=8192+8*SCR+ROW:PE=PEEK(CH)         :rem 107
530 POKE CH,(PE AND (255-3*P2(BIT))) AND NOT B*2↑B
    IT                                       :rem 255
540 POKE CH,(PE AND (255-3*P2(BIT))) OR B*2↑BIT
                                             :rem 221
550 IF MOVE=1 THEN POKE CH,PE               :rem 120
590 IF MOVE=0 THEN POKE 1024+SC,CE:POKE 55296+SC,C
    C                                         :rem 82
600 GOTO 100                                  :rem 96
20000 DATA 162,32,138,133,252,169,0,133,251:rem 53
```

```
20001 DATA 145,251,200,208,251,230,252,202,208
                                         :rem 191
20005 DATA 246,134,252,134,254,169,216,133,253
                                         :rem 219
20006 DATA 169,4,133,255,162,4,160,0,169,3 :rem 12
20010 DATA 145,252,169,53,145,254,200,208 :rem 221
20011 DATA 245,230,253,230,255,202,208,238,96
                                         :rem 163
50000 GET G$:IF G$="" GOTO 50000          :rem 35
50010 PRINT G$ "{RVS}{8 SPACES}"          :rem 68
50020 G=PEEK(646):PRINT "{WHT}":RETURN    :rem 187
```

Sprites

Sprites (or Movable Object Blocks) are large user-definable graphics which can be put anywhere on the screen. The VIC-II chip handles them automatically—a considerable technical achievement to Commodore's credit. Since many people feel intimidated by sprites, this section begins with simple demonstrations and leaves the technical details for later.

First, turn on the 64, type POKE 53269,1 and press RETURN. You now have a sprite. However, you can't actually see it since it is not in the screen display area. Type POKE 53248,100 and POKE 53249,100 and a sprite will appear. Vary the values in these locations and watch the sprite move. The sprite is, or is supposed to be, white. This color was set on power-up.

At this point, the sprite's shape is not very satisfactory; it is defined by the first 63 bytes of RAM. We can alter it by POKEing different values into location 2040, the location which tells the VIC-II where to find this sprite's shape data. Some values yield a sprite which continuously changes; this means that the RAM which defines the sprite's shape is being used for BASIC workspace and isn't a flaw in the 64. POKE 2040,16 causes the top one-and-a-half screen lines (strictly, the first 63 characters) to define the sprite; try homing the cursor and redefining the sprite with {RVS}–Commodore key–B, {RVS}-*, SHIFT-U, and @, setting bit patterns 11111111, 10101010, 01010101, and 00000000.

Now, POKE 53287 with different values. These change the color of the sprite. Color changes in the sprite correspond to bits set to 1 in the sprite's definition. (You may get color effects because of the spacing of the defining bits on the screen.) Bits set to 0 don't represent a color, but are treated as transparent by the VIC chip, so the background shows through.

POKE 53276,1 sets multicolor mode for the sprite. (Poke 53276,0 to return to high-resolution.) Multicolor mode increases the available colors from 1 to 3; the extra two are stored in 53285 and 53286, so POKEs into these locations will alter multicolor sprites (if bit patterns 10 or 01 are present), but will leave high-resolution sprites unchanged. All multicolor sprites share these extra two colors.

POKE 53277,1 makes the sprite expand horizontally, and POKE 53271,1 expands it vertically to twice the unexpanded dimensions. Without special techniques, only eight sprites are available at one time, so this can be useful where you'd like reasonable coverage of the screen.

POKE 53275,1 changes the priority of the sprite with regard to text; check to see that characters are now displayed in front of it.

PRINT PEEK(53279) is set to 1 if the sprite overlaps text. This is called a collision. As you'll find, this location is reset only when it's read from, so two PEEKs are actually necessary to give the current status.

Finally, POKE 53265,187 sets bitmap mode. You'll see that a sprite is still displayed—the graphics mode doesn't affect it.

Detailed Description of Sprites

Enabling and disabling sprites. The seven bits of the VIC register at $D015 (53269) control sprite enabling. The VIC chip can handle up to eight sprites at one time, numbered 0–7, turned on when corresponding bits 0–7 are 1, and turned off when bits 0–7 are 0. Examples: POKE 53269,1 turns on sprite 0—the introduction used this sprite; POKE 53269,255 turns on all sprites. Sprites coexist with all graphics modes—ordinary text, user-defined characters, and bitmapping.

When sprites are on, the VIC-II needs extra processing time, which it takes at the expense of the 6510 central processor. For example, if all sprites are enabled, the 6510 operates at 0.95627 times the speed it runs at with all sprites disabled, and this effect is present even if the VIC chip is disabled in border-color mode. This slowdown affects disk, tape, and RS-232 operations, so it makes sense to press RUN/STOP–RESTORE before loading or saving.

Sometimes it's helpful to quickly disable sprites, then reenable them (for example, when moving a sprite). Two POKEs are needed to alter the position, and if one acts noticeably before the other, motion will be in two parts, horizontal and vertical (another way to handle sprites is to move them when the raster scan is off the screen).

Defining sprites. Unexpanded sprites are about 3 × 3 characters wide. Nine characters use 9*8*8 dots, and require 72 bytes for definition, but to fit sprite data in RAM compactly, VIC allows 64 bytes per definition, allowing 24 × 21 dot sprites. Byte 64 is ignored, while the other 63 are arranged in 21 three-byte groups as in Figure 12-10.

Sprite definitions must coexist in the VIC-II bank with the screen and any user-defined characters or bitmap and color screen that may be in use. Since only 64 bytes are needed per definition, there's usually plenty of room.

When sprites are enabled, VIC-II immediately displays them in accordance with the values held in its various sprite control registers. The only parameters not in the VIC-II chip are eight pointers to the sprite shape definitions, which are stored after the screen. If the screen is moved, these pointers move with it, but normally locations 2040–2047 apply. They can hold anything from 0 to 255, since 256*64 is exactly the size of a VIC-II bank. So, POKE 1020,13 points sprite 0's start to 13*64=832, near the start of the tape buffer.

Positioning sprites. Since there are 40 × 25 characters, and sprites can be controlled to the nearest pixel, at least 320 horizontal (X) and 200 vertical (Y) positions have to be programmable. Each sprite's Y value has its own one-byte VIC register, but X values require nine bits. The X values are stored in one-byte registers, with all the extra high bits collected elsewhere in another register. The first 16 VIC registers,

$D000–$D00F (53248–53263), hold X, then Y positions for sprites 0–7, followed by the high-bit register at $D010 (53264). The X,Y pairs define the top-left corner of the complete sprite because VIC scans the screen from the left and down.

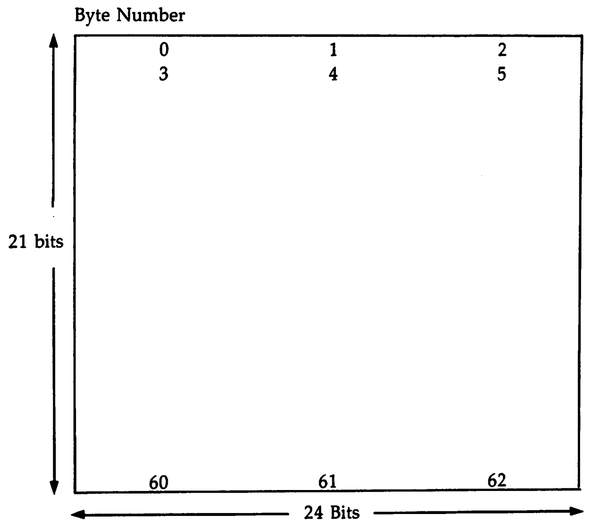
You might at first expect X values to range from 0 to 319, but this wouldn't take into account the way the VIC-II chip scans the screen. In fact, X values from 24 to 342 define the screen width, and Y values from 50 to 249 define the height. These figures refer to the sprite's top-left corner, and you must allow for the sprite's size.

An unexpanded sprite is 24 dots across by 21 down, so to just fit a sprite in the screen, X should range from 24 to 296, and Y from 50 to 208. By using X and Y values outside of these ranges, you can move a sprite gradually off the screen in any direction. For instance, if Y=29, the sprite is just above the top of the screen; if Y=250, the sprite is below the screen, and so on.

Note that X=0 makes an unexpanded sprite just vanish off the left border. But an expanded sprite with X=0 is still partly visible. It can be made to move further left only by making X high, up to a maximum of 511. (PAL TVs, in the U.K., have a different maximum of 503. A larger X won't display at all.) This means it's tricky to move X-expanded sprites smoothly off the left side of the screen.

Sprite colors. High-resolution sprite colors are stored in eight registers from $D027 (53287) to $D02E (53294). On power-up, these are set to 1, 2, 3, 4, 5, 6, 7, and 12.

Figure 12-10. Sprite Data Arrangement



Sprite high-resolution and multicolor modes. Bits 0–7 of $D01C (53276) control the modes of each sprite independently. High-resolution sprites interpret each bit set to 1 in their definitions as the sprite color, stored in one of the color registers at $D027–$D02E. Bits cleared to 0 are transparent, using the background color.

Multicolor sprites interpret pairs of bits in their shape definitions like this:

00	Transparent
01	Sprite MCM register 0 ($D025 = 53285)
10	Sprite color in one of $D027–$D02E
11	Sprite MCM register 1 ($D026 = 53286)

The two new colors are shared by all multicolor sprites. Note that character or background colors aren't used, which makes sense, since you do not generally want the sprite's colors to vary as it moves. Note that if the second bit is set to 1, an MCM register is selected.

Expanded sprites in effect use each dot twice, so the resolution isn't improved. Note that expanded multicolor sprites are still displayed in multicolor—the sprite handling does not treat 01 as 0011, for example.

Expansion of sprites. Bits 0–7 of $D01D (53277) and $D017 (53271) control the X and Y expansion of each of the eight sprites. Thus, POKE 53277,1: POKE 53271,9 causes sprite 0 to be doubled in both directions, and sprite 3 to be doubled in the Y direction. Expansion in the Y direction elongates a sprite downwards from its present position, and X expansion stretches it to the right.

Priority of sprites. When sprites are superimposed, the VIC-II chip can't display both at once, but has to select which gets priority. The same happens on a background of graphics, so we have to distinguish two types of priority.

Sprite-sprite priority determines which of two or more sprites is displayed where they overlap. Lower numbered sprites always appear in front of higher numbered ones; this is built into VIC-II and must be taken into account when designing programs with sprites which may overlap. The nearest sprite could be 0, say, and the furthest sprite, 7. This priority applies to the nontransparent parts of both high-resolution and multicolor sprites. Transparent parts of a sprite allow lower priority sprites or background graphics to show through.

Sprite-data priority is more complicated. Seven bits of $D01B (53275) control sprite-data priority for each sprite. This determines whether a sprite appears behind or in front of character data when the two overlap. When a bit is 0, its sprite gets priority, and when 1, data gets priority. On power-up all bits are set to 0, so sprites initially have priority. This concept is sometimes termed sprite-background priority, since sprites are often displayed on a background of character data. Do not confuse this with the background color of the screen, over which sprites always have priority.

With high-resolution sprites the transparent parts always allow what's underneath to show, but the parts mapped with 1 allow character data to show only if the bit in $D01B is 1. Thus, an airplanelike sprite can be made to fly in front of or behind user-defined character "mountains."

Multicolor-sprite priority with data is controlled in the same way. When the sprite's bit in $D01B is 0, the sprite, except the transparent areas defined by bit pattern 0, has priority over data. When the bit is 1, data has priority over the entire sprite.

Priority with several sprites and data is more complex. The setting of the lower numbered sprite gets priority. Now, suppose sprite 0 overlaps sprite 1, and there's some data under both. If sprite 0 is set to appear below data, then data will show through, *even where sprite 1 is set with priority over data.*

Collision detection. This is an essential aspect of sprite programming. The idea is to signal whenever a nontransparent part of a sprite contacts screen data or another sprite. VIC-II has two registers for this purpose, plus two interrupt registers. Without all these, detecting collisions would be enormously difficult. They can be PEEKed or used with ML to generate interrupts. Here's a simple example:

Program 12-29. Sprite Collision

```
10 FOR J=0 TO 62:POKE 832+J,7:NEXT
20 POKE 2040,13:POKE 2041,13
30 V=53248
40 POKE V+21,3
50 POKE V+39,1:POKE V+40,0
60 POKE V,135:POKE V+1,70
100 POKE V+3,60
110 PRINT "{CLR}{WHT} SPRITE-SPRITE{3 SPACES}SPRIT
    E-DATA{3 SPACES}INT REG"
120 FOR J=0 TO 255: POKE V+2,J
130 PRINT SPC(8) PEEK(V+30) SPC(11) PEEK(V+31);
140 PRINT SPC(8) PEEK(V+25) AND 6:PRINT"{UP}";
150 POKE V+25,6
160 NEXT:GOTO 20
```

The seven bits of $D01E (53278) register sprite-sprite contact, so at least two bits are set on any sprite overlap. The register must be read to determine which sprites were involved. Reading resets all the bits to 0. If the register is not read, the collision bits simply stay 1 indefinitely, which is why reading twice is essential to find the current status. Note that offscreen collisions set these flags, too.

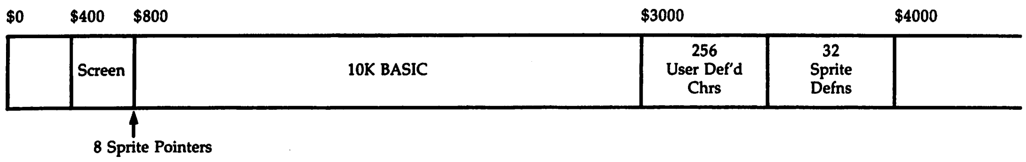
Bits 0–7 of $D01F (53279) register sprite-data collisions in the same way. Typically, just one bit will be set. In high-resolution this is straightforward. With multicolor sprites there's more flexibility. The sprite color and the transparent "color" are each treated as transparent for collision flagging; only bit-pairs 10 or 11, that is, the colors in multicolor registers $D025 and $D026, cause collisions to be registered.

Sprite Memory Maps

Each sprite definition takes up 64 bytes which must coexist in a VIC-II bank with screen RAM, characters, and/or a bitmap with its color. Areas giving images of ROM characters can't be used.

Figure 12-11 shows how bank 0 can hold a full set of 256 user-defined characters, 32 sprite definitions, and a 10K BASIC program starting at the usual $0800:

Figure 12-11. Mapping Sprites with User-Defined Characters



Program 12-30 uses the memory map in Figure 12-11. The program is long, but is as short as a reasonable demonstration can be:

Program 12-30. Programming Sprites with User-Defined Characters

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
1Ø POKE 55,Ø:POKE 56,48:CLR                      :rem 221
2Ø POKE 53272,29                                  :rem 43
3Ø FOR J=12288 TO 12288+31:READ X:POKE J,X:NEXT
                                                 :rem 1Ø7
4Ø DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø                           :rem 5Ø
5Ø DATA 1,3,7,31,31,63,63,127                    :rem 13Ø
6Ø DATA 255,255,255,255,255,255,255,255          :rem 148
7Ø DATA 128,128,224,24Ø,24Ø,254,255,255          :rem 13Ø
8Ø FOR J=1 TO 5:M1$=M1$+"@@@AC@@@":NEXT          :rem 134
85 FOR J=1 TO 5:M2$=M2$+"@@ABBC@@":NEXT          :rem 145
9Ø FOR J=1 TO 5:M3$=M3$+"@ABBBBC@":NEXT          :rem 147
95 FOR J=1 TO 5:M3$=M3$+"ABBBBBBC":NEXT          :rem 156
1ØØ POKE 5328Ø,14:POKE 53281,14                   :rem 8Ø
11Ø FOR J=1 TO 4Ø:SP$=SP$+"@":BL$=BL$+"B":NEXT
                                                 :rem 241
12Ø PRINT "{CLR}{7}":FOR J=1 TO 6:PRINT SP$;:NEXT
                                                 :rem 133
13Ø PRINT "{BLU}" M1$ M2$ M3$ M4$;                :rem 77
14Ø FOR J=1 TO 5:PRINT BL$;:NEXT                  :rem 57
15Ø FOR J=1 TO 8:PRINT "{RED}" BL$;:NEXT         :rem 157
2ØØ DATA 1,255,224, Ø,7,24Ø, Ø,3,96, 192,54,195, 2
    27,252,251                                   :rem 184
21Ø DATA 255,255,255, 111,255,251, Ø,127,3, Ø,14,Ø
                                                 :rem 238
22Ø FOR J=14336 TO 14362:READ X:POKE J,X:NEXT
                                                   :rem 4
23Ø FOR J=14363 TO 144ØØ:POKE J,Ø:NEXT            :rem 4Ø
24Ø POKE 53269,3                                  :rem 45
25Ø POKE 2Ø4Ø,224:POKE 2Ø41,224                   :rem 68
26Ø POKE 53277,1:POKE 53271,1                    :rem 244
27Ø POKE 53287,Ø:POKE 53288,11                    :rem 46
28Ø POKE 53275,2                                  :rem 45
3ØØ XØ=Ø                                         :rem 136
31Ø POKE 53249,5Ø+15Ø*RND(1)                     :rem 172
32Ø POKE 53251,1ØØ+3Ø*RND(1)                     :rem 159
```

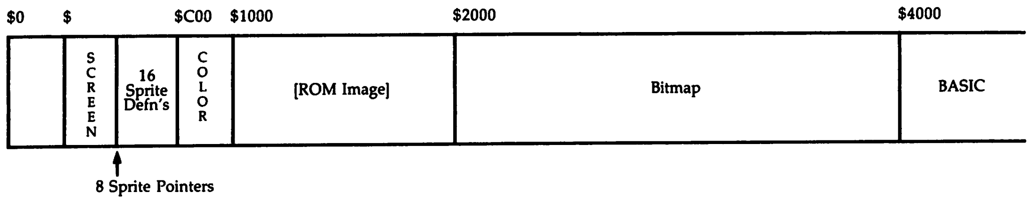
```
330 XØ=XØ+2:IF XØ>255 THEN POKE 53264,PEEK(53264)
    {SPACE}OR 1                              :rem 69
340 IF XØ<256 THEN POKE 53264,PEEK(53264) AND 254
                                             :rem 253
350 IF XØ>36Ø GOTO 3ØØ                        :rem 82
360 POKE 5325Ø,XØ/3                          :rem 221
370 POKE 53248,XØ AND 255:GOTO 33Ø           :rem 251
```

Four user-defined characters build a mountain range (@, A, B, and C become blank, left slope, solid block, and right slope, respectively). One sprite, which looks like an airplane, is defined. It's displayed twice: once as sprite 0, in enlarged format in black, with priority over data, and again as sprite 1, in gray at normal size, with priority lower than data. The two planes move left to right, the nearer plane moving faster, the further disappearing behind the mountains and also, because of automatic sprite-sprite priority, behind sprite 0 if they cross. Lines 300 onward move the planes, and lines 320 and 330 allow for the possible high bit in the X direction as the plane moves. (If you experiment with sprite shape definitions, you'll see single dot widths aren't handled well by the VIC chip. Sprites generally need to be fairly chunky to work well.)

The tape buffer has room for three sprite shape definitions (at $0340, $0380, and $03C) and is useful for small experiments with BASIC. Values of 13, 14, and 15, POKEd into the sprite pointers after the screen, access the area.

The memory map shown in Figure 12-12 illustrates how a bitmapped screen, its color, and 16 sprite shape definitions can be used with the normal screen RAM in bank 0. BASIC has to be shifted up to start at $4000, with POKE 43,1: POKE 44,64: POKE 16384,0: NEW.

Figure 12-12. Mapping Sprites and a Bitmap



Notes on Programming with Sprites

Extending the number of sprites available. Without special techniques, only eight sprites can exist on the screen at one time. Unexpanded sprites occupy about a 7.9 character area, so even eight fully expanded sprites cover only 25 percent of the screen. If this isn't enough, you'll need user-defined characters or bitmapping to add graphic interest. The VIC-II chip can be caused, with interrupt techniques, to display more than eight sprites simultaneously, so it is possible to fill the screen with sprites, but ML is essential.

How to use sprites. The number of sprites is usually limited, so it is necessary to mix them with built-in graphics. If that will not suffice, the next easiest method is to mix them with user-defined graphics. In either case they must be used carefully. For example, to liven up a word processor or calculation program, you could use sprites to define arrows, pointing fingers, rectangular frames, or other prompts. Games might use several moving sprites to simulate cars or motorcycles on a conventional graphics background.

Another example is a frog trying to cross a road and a stream without being hit. A screenful of moving trucks, cars, and logs isn't possible with simple sprite techniques: user-defined graphics are better, allowing duplication of the moving objects. If the frog is defined as a sprite, with priority over character data, it will always be visible upon its road, log, or wherever. Several sprite definitions cover the cases where it moves left or right and extends its legs. You can choose either a high-resolution frog in one color or a chunkier multicolor frog.

Sprites can be superimposed if extra color detail is needed, but this isn't often done, partly because it reduces the number of available sprites, partly because motion now requires two sprites to be moved, and partly because TVs may not display the result satisfactorily anyway.

Movement with sprites. To animate sprites, you need to replace a sprite with a similar sprite, possibly repeating the process many times. We could change the sprite definitions themselves, change the sprite pointers to point to different definitions, or cycle through the sprites, say, from 0 through 3.

Generally, changing the definition pointers is best, since one POKE is all that's needed to update a sprite, and it's easy to store plenty of sprites in RAM. Sometimes, though, changing the actual definition is better (for example, where an object is fired at, and you want to make parts of it disappear). In this case, it may be easier to set bits in the sprite definition to 0. The third option isn't usually good; it uses up valuable sprites.

Receding and approaching motion can be simulated to a certain extent by using the expansion feature along with changing definitions; a 2 × 2 and a 3 × 3 unexpanded sprite will give a sequence of four sprites in about the right ratio to suggest constant speed. Note, however, that expansion stretches the sprite downward and to the right (rather than equally in all dimensions), which makes this feature less than ideal for three-dimensional effects. Lighter and bluer colors suggest distance, as opposed to deeper and redder colors.

Sprite Editor

Rather than drawing sprites on 24 × 21 areas of squared paper and converting the result into bytes, try the following sprite editor, Program 12-31, which automates the process. The program processes one sprite at a time, which is displayed in four ways, standard and enlarged, and both high-resolution and multicolor modes. There's also an enlarged diagram of the sprite, on which individual points can be set and cleared with the space bar, while the cursor keys allow movement. Function keys control colors and allow plotting in pairs for multicolor mode; instructions on the screen explain which function keys to use. Press C to erase a sprite. (You may prefer to omit the C option to remove the risk of accidental deletion.)

The program asks which block is to be used; if a sprite already exists in memory, it's not cleared and can be examined and altered.

Program 12-31. Sprite Editor

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
5 INPUT "{CLR}{3 DOWN} WHICH BLOCK OF 64 BYTES (EG
   13)";B:SP=64*B                              :rem 225
10 FOR J=0 TO 3:POKE 2040+J,B:NEXT              :rem 96
15 V=53248:SS=1024                             :rem 227
20 POKE V+21,15                                  :rem 7
25 POKE V+0,240:POKE V+2,240:POKE V+4,250:POKE V+6
   ,250                                        :rem 171
30 POKE V+1,90:POKE V+3,140:POKE V+5,190:POKE V+7,
   220                                         :rem 125
35 FOR J=39 TO 42:POKE V+J,6:NEXT               :rem 90
40 POKE V+37,0:POKE V+38,13                    :rem 191
45 POKE V+32,1:POKE V+33,1                     :rem 136
50 POKE V+29,3:POKE V+23,3                     :rem 141
55 POKE V+28,5                                 :rem 229
60 FOR J=0 TO 7:P(J)=2↑(7-J):NEXT               :rem 72
65 GOSUB 500                                   :rem 128
100 PRINT "{CLR}{2 DOWN}{BLK}":GOSUB 3000       :rem 36
130 XC=0:YC=0:C=160:POKE SS+C,PEEK(SS+C) OR 128
                                               :rem 159
140 GOSUB 20000                                  :rem 7
200 GET X$:IF X$="" THEN 200                   :rem 117
210 IF X$=" " THEN GOSUB 2000:GOTO 400         :rem 151
220 IF X$="C" THEN GOSUB 1000:GOTO 100         :rem 215
230 IF X$="{F1}" THEN CB=0:GOSUB 21000:GOTO 400
                                               :rem 123
240 IF X$="{F3}" THEN CB=1:GOSUB 21000:GOTO 400
                                               :rem 126
250 IF X$="{F5}" THEN CB=2:GOSUB 21000:GOTO 400
                                               :rem 129
260 IF X$="{F7}" THEN CB=3:GOSUB 21000:GOTO 400
                                               :rem 132
270 IF X$="{UP}" AND YC>0 THEN YC=YC-1:GOTO 400
                                               :rem 156
280 IF X$="{DOWN}" AND YC<20 THEN YC=YC+1:GOTO 400
                                                :rem 75
290 IF X$="{RIGHT}" AND XC<23 THEN XC=XC+1:GOTO 40
   0                                            :rem 88
300 IF X$="{LEFT}" AND XC>0 THEN XC=XC-1:GOTO 400
                                               :rem 159
310 IF X$="{F2}" THEN GOSUB 4000:GOTO 200       :rem 33
320 IF X$="{F4}" THEN GOSUB 5000:GOTO 200       :rem 36
330 IF X$="{F6}" THEN GOSUB 6000:GOTO 200       :rem 39
340 IF X$="{F8}" THEN GOSUB 7000:GOTO 200       :rem 42
350 GOTO 200                                    :rem 99
400 POKE SS+C,PEEK(SS+C) AND 127:C=XC+40*YC+160
                                               :rem 102
410 POKE SS+C,PEEK(SS+C) OR 128:GOTO 200       :rem 207
```

```
500 DATA "BLACK ","WHITE ","RED{3 SPACES}","CYAN
    {2 SPACES}","PURPLE"                      :rem 111
505 DATA "GREEN ","BLUE{2 SPACES}","YELLOW","ORANG
    E","BROWN "                               :rem 113
510 DATA "LT RED","D GRAY","M GRAY","LT GRN","L BL
    UE","L GRAY"                               :rem 16
520 DIM CO$(15):FOR J=0 TO 15:READ CO$(J):NEXT:RET
    URN                                       :rem 249
1000 FOR J=SP TO SP+62:POKE J,0:NEXT:RETURN:rem 77
2000 PT=SP+XC/8+YC*3:BY=PEEK(PT)           :rem 195
2010 BP=XC-INT(XC/8)*8:MS=P(BP)             :rem 68
2020 IF (BY AND P(BP))=0 THEN POKE PT,BY+MS:rem 70
2030 IF (BY AND P(BP))>0 THEN POKE PT,BY-MS:rem 74
2040 POKE SS+C,254-PEEK(SS+C):RETURN       :rem 161
3000 FOR YC=0 TO 20:FOR XB=0 TO 2          :rem 181
3010 PT=SP+XB+YC*3:BY=PEEK(PT)              :rem 93
3020 FOR J=0 TO 7                           :rem 61
3030 IF (BY AND P(J))>0 THEN PRINT "Q";: GOTO 3050
                                           :rem 180
3040 PRINT "-";                              :rem 0
3050 NEXT:NEXT:IF YC<>20 THEN PRINT:NEXT   :rem 246
3060 RETURN                                :rem 169
4000 T=(PEEK(V+32)+1) AND 15               :rem 151
4010 POKE V+32,T:POKE V+33,T:GOTO 20000    :rem 143
5000 POKE V+37,(PEEK(V+37)+1) AND 15:GOTO 20000
                                           :rem 183
6000 T=(PEEK(V+39)+1) AND 15               :rem 160
6010 FOR J=V+39 TO V+42:POKE J,T:NEXT:GOTO 20000
                                           :rem 189
7000 POKE V+38,(PEEK(V+38)+1) AND 15:GOTO 20000
                                           :rem 187
20000 PRINT "{HOME}{BLK}";                 :rem 161
20010 PRINT "BACKGROUND = "CO$(PEEK(V+32) AND 15)"
      {2 SPACES}F2{3 SPACES}F1=00"          :rem 77
20020 PRINT "SPRITE MC0 = "CO$(PEEK(V+37) AND 15)"
      {2 SPACES}F4{3 SPACES}F3=01"          :rem 15
20030 PRINT "SPR COLOR{2 SPACES}= "CO$(PEEK(V+39)
      {SPACE}AND 15)"{2 SPACES}F6{3 SPACES}F5=10"
                                           :rem 243
20040 PRINT "SPRITE MC1 = "CO$(PEEK(V+38) AND 15)"
      {2 SPACES}F8{3 SPACES}F7=11"          :rem 28
20050 RETURN                               :rem 215
21000 CP=(C+SS) AND 2046:IF CB>1 THEN POKE CP,81:G
      OTO 21030                            :rem 115
21010 POKE CP,45                            :rem 75
21030 CP=CP+1:IF (CB AND 1)=1 THEN POKE CP,81:GOTO
       21050                               :rem 183
21040 POKE CP,45                            :rem 78
21050 PT=SP+XC/8+YC*3:BY=PEEK(PT):MP=7-XC AND 6:MK
      =2↑MP*3                              :rem 174
21060 BY=(BY AND NOT MK)+CB*2↑MP:POKE PT,BY:RETURN
                                            :rem 14
```

When sprites have been defined, press RUN/STOP to exit the program. The sprite definition values can then be saved as DATA or written to a file or block saved, using the techniques discussed earlier in this book.

Using Interrupts with Graphics

Chapter 8 explains NMI and IRQ interrupts on the 64. Graphics applications include such things as clocks, countdown indicators, and radar displays to show approaching aliens and screen responses to keypresses. Any display which needs to be periodically updated is a candidate for processing during the interrupt. Interrupts require ML, but offer a solution to many problems. Often, a similar effect can be achieved in BASIC, but this is far clumsier and slower.

Program 12-32 processes graphics during interrupts, PEEKing the first 256 screen positions for values specified as DATA, replacing them by the next in the sequence. Note how the interrupt routine is transparent to BASIC. The ML is set (by the 5 in line 200) to search and replace at about 1/10 second intervals.

Program 12-32. IRQ Polling

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 FOR J=49152 TO 49207:READ X:POKE J,X:NEXT
                                          :rem 219
20 POKE 56333,127:POKE 788,0:POKE 789,192:POKE 563
   33,129                                 :rem 213
30 PRINT "{CLR}":FOR J=1 TO 100:PRINT CHR$(228) "
   {SPACE}";:NEXT                         :rem 129
200 DATA 206,47,192,208,23,169,5,141,47,192,160,0
                                          :rem 103
201 DATA 162,7,185,0,4,221,48,192,240,9,202,16
                                          :rem 202
202 DATA 248,200,208,240,76,49,234,232,224,8,208
                                           :rem 58
203 DATA 2,162,0,189,48,192,153,0,4,76,25,192
                                          :rem 165
204 DATA 5                                :rem 229
210 DATA 100,111,121,98,248,247,227,224   :rem 120
```

This process can be extended, with user-defined characters, to simulate flying birds, crawling insects, and so on.

This style of interrupt is a *poll*. At regular intervals, the 6510 goes off to perform what may be a long series of operations, deciding to do some and not do others. CIA chip 1 allows the frequency of interrupts to be changed, but the process is always similar. However, the VIC-II chip provides a more active control over interrupts.

VIC-II's Interrupt Registers

Locations $D019 (53273) and $D01A (53274) are VIC's interrupt flag register and interrupt mask register, respectively. They allow the source of any interrupt to be identified, and also allow interrupts to be enabled or disabled (masked). Six other registers are related to the VIC-II chip's interrupts, as discussed below.

Bit 7 of the interrupt flag register is set to 1 when the VIC chip generates an interrupt; otherwise it's 0. This allows you to determine the source of an interrupt, where there are several possibilities, since whenever bit 7 is on, one of the register's other four used bits will reveal this. Note, though, that bits 4–6 are unused and that bits 0–3 are set even when interrupts aren't enabled.

PRINT PEEK(53273) AND 143 prints the value of this register; ANDing the value with 143 masks the bits which are fixed at 1. Bits 0–3 act as the flags. When a bit is low (contains 0) the flag is clear; if a certain event occurs, the bit becomes high (contains 1) and the flag is set. Bit 0 is set by a raster compare; sprite-data collisions show in bit 1; sprite-sprite collisions show in bit 2; and connecting and using a light pen sets bit 3.

Once a flag bit is set, it can be cleared only by POKEing it *high* (unlike many other flags, which are cleared by POKEing a bit low). This is called *latching*, and the idea of this dual function in the register is to keep a record of interrupts within the flags themselves, saving programming effort. As an example, a sprite-data collision sets bit 1; if no other triggering events have occurred, the PEEKed value will be 2. POKE 53273,2 is required to turn off the collision flag (assuming the sprite no longer overlaps data) and return the bit to 0. The interrupt flag register, therefore, is designed to store the past results of four possible events until they're cleared, and also shows whether the VIC-II chip caused an interrupt currently in force.

The use of this register requires considerable care. Confusion will result if you forget to clear a flag, try to clear it by POKEing its bit low, or try to clear it while the condition that triggered it still exists. Other anomalies might arise because the keyboard and light pen share common wiring.

Using the interrupt enable register to enable interrupts is simple, but the IRQ routine which they're wired to needs modifications to handle them properly. Try PRINT PEEK(53274) AND 15. The result is 0, showing that the VIC chip has no interrupts enabled. Enter POKE 53266,0: POKE 53274,1. This enables raster-scan interrupts: each time the TV picture is scanned, an interrupt occurs. But, in addition, the flag register isn't cleared, so immediately when an interrupt finishes, a new one begins. This stops processing, although (try SHIFT–Commodore key) the keyboard is processed normally. Similarly, POKE 53274,2 has no effect until there's a sprite-data collision, whereupon interrupts continuously come into effect. Because of this, user-written interrupt routines based on the VIC-II chip always clear the interrupt flag register before exiting.

Sprite Collision and Light Pen Interrupts and Registers

The demonstration routine below, Program 12-33, adds a sprite-data interrupt to the normal IRQ.

Program 12-33. Sprite-Data Collision

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 FOR J=49152 TO 49172:READ X:POKE J,X:NEXT
                                          :rem 220
20 POKE 2040,13:POKE 53269,1              :rem 182
30 FOR J=832 TO 894:POKE J,255:NEXT       :rem 170
40 POKE 53248,100:POKE 53249,53            :rem 90
```

```
100 POKE 56333,127:POKE 788,0:POKE 789,192:rem 207
110 POKE 56333,129                          :rem 141
120 POKE 53274,2                             :rem 37
130 X=100:D=1                               :rem 166
140 PRINT "{CLR}{DOWN}{2 SPACES}£J£{16 SPACES}£J£"
    :                                       :rem 174
200 IF PEEK(2)=0 GOTO 230                     :rem 4
210 IF X>100 THEN D=-1                       :rem 87
220 IF X<101 THEN D=+1                       :rem 85
230 X=X+D:POKE 53248,X:GOTO 200             :rem 233
1000 DATA 173,31,208,240,8,173,31,208,169,2:rem 57
1001 DATA 141,25,208,133,2,173,31,208,76,49,234
                                              :rem 0
```

After Program 12-33 is run, there are two sources of interrupts, the result of redirecting the IRQ vector in locations 788–789. The sprite's direction reverses when it collides with data, so it moves left and right across the screen. This is the ML loaded by Program 12-33:

```
      LDA  $D01F
      BEQ  EXIT
      LDA  $D01F
      LDA  #$02
      STA  $D019
EXIT  STA  $02
      JMP  $EA31
```

This reads the sprite-data collision register and resets the interrupt flag register on a collision, as well as clearing $D01F a second time. The result is passed to BASIC in location 2. JMP $EA31 continues in the ordinary key-scan interrupt. In fact, the extra interrupt isn't necessary here; reading the relevant register works. But in more complex situations, when several interrupts may be enabled, processing them properly requires a routine like LDA $D019:AND #1:BNE RASTER:AND #2:BNE SPRDATA, and so forth, with RASTER STA $D019, and so forth, where the first act is to turn off the flag.

Light-pen registers can be read during interrupts, but can also generate interrupts of their own, by setting bit 3 in the IRQ Mask Register ($D01A), and setting bit 3 of the Interrupt Flag Register ($D019) high to clear the flag once the light pen causes an interrupt. But since the light pen cannot trigger more than 60 interrupts per second (50 in the U.K.), it's easier not to bother, but simply to add a test for bit 3 of $D019 to the normal IRQ, reading from $D013 and $D014 when the bit becomes high.

Raster Interrupt and Registers

TV pictures are generated by an electron beam which sweeps horizontal lines from top to bottom of the screen, lighting up individual phosphor dots, called *pixels*, in the process. Each row is a *raster line* and is scanned left to right. Persistence of vision, plus the time taken for the phosphors to dim, allows realistic movement, and the picture is improved by interlacing, where alternate scans display odd numbered and even numbered lines; in other words, a single screen scan displays half the picture. Sixty top-to-bottom scans (50 in the U.K.), are displayed every second, so the entire picture is refreshed every 1/30 second (1/25 in the U.K.). The VIC-II chip generates each raster line, keeping track of the current line as it does so.

NTSC (U.S.) TVs have 262 lines per frame, PAL (U.K.) TVs have 312. The 64 displays lines 51–251, making 200 lines—enough for 25 sets of eight dots. Raster lines 0–50 and 252 up aren't visible. TVs in the U.K., with more lines, give a more compressed picture.

Bit 7 of $D011 (53265) with register $D012 (53266) together make a nine-bit register for use with raster scanning. Nine bits are necessary to include all the raster lines, though the highest bit is often not used. The register has two different functions. It allows you to PEEK the current raster line. Writing to one or both registers latches the new value, setting the VIC-II chip so an interrupt flag, bit 0 in $D019, goes high whenever the current raster line matches all nine bits. If the interrupt is enabled, an interrupt will be caused. POKEs which set bit 7 of $D011 and also put a high value in $D012, prevent raster interrupts from occurring. This fact allows the 64 to deduce which type of VIC chip is fitted and set its PAL/NTSC flag in $02A6. On power-up, VIC's raster register is POKEd with 311, a value too high for U.S. (NTSC) signals, but within the range of U.K. (PAL) TVs.

So, when we use raster lines in programs, we have a choice of methods: reading the raster line from its registers or, more ambitiously, generating precisely timed interrupts synchronizing with the screen.

Using the Raster Interrupt

WAIT 53265,128,128: WAIT 53265,128 shows how the raster line can be used, in this case to detect the scan somewhere above midscreen. A carefully timed delay, followed perhaps by a change of background color, allows a smooth change when the scan is offscreen. But BASIC isn't generally fast enough; the raster may scan several lines in the time it takes to perform one PEEK.

A SYS call to the ML routine below changes screen colors twice every screen scan, until a keypress returns to BASIC.

```
START LDA  $D012  ;READ EIGHT BITS OF SCREEN LINE
      BNE  START  ;WAIT TILL WE GET POSITION 0
      LDA  #2
      STA  $D021  ;RED BACKGROUND
      LDA  #$80
WAIT  CMP  $D012  ;WAIT TILL RASTER LINE IS 128
      BNE  WAIT
      LDA  #3
      STA  $D021  ;CYAN BACKGROUND
      LDA  $C5    ;EXIT ON KEYPRESS
      CMP  #$40
      BEQ  START
      RTS
```

Routines like this use the processor full time to handle the screen. Alter #$80 to watch the dividing-line move; the loops WAIT and START occupy almost all the 6510's time. Interrupts are trickier but allow other processing. Effects include the use of alternate fields: semitransparent sprites, fades, new colors obtained by superimposing one or more standard colors. Eighty-column lettering is sometimes displayed with half of each character in every other field. All that's needed is an interrupt routine synchronized with the screen with a counter to select alternative processing paths. Experiments like this, however, tend to be flickery and unclear.

More important are split-screen methods, where the screen is separated across the middle into zones, allowing mixed bitmapping with text, multiple sprites, or different character sets in coexistence.

Timing is important. For example, the keyscan routine takes roughly a millisecond. Since the TV draws 15 raster lines in this time, interrupt-driven routines are vulnerable to slight timing errors which make the split-screen boundary unstable.

Split screens. It is possible to divide the screen in half using a raster interrupt. The bottom half can be text, and the top half bitmapped graphics, stored from $2000 in VIC bank 1, for example. (This means characters printed in the top half would show as colored squares.) However, you can allow scrolling by filling the top text line with spaces and filling the line above with color codes. It's possible to avoid this problem by moving the bitmap color elsewhere. Most of the interrupt routine would be taken up by alternately changing $D011 and $D018 between text and graphics, then resetting $D012 to cause an interrupt half a screen later.

If the program allows the normal CIA interrupt as well, you'll see the dividing line become irregular, since the exact raster synchronization is lost. The cursor flashes at twice the normal rate, if the ML exits to $EA31, the key-scan routine. For more on using split screens, see "Split Screens" and "Son of Split Screens" in *Compute's First Book of Commodore 64.*

Thirty-two sprites demonstration. The next example, Program 12-34, avoids the fast cursor problem by exiting at $EA81 for all interrupts except one per frame. It puts 32 sprites on the screen, by generating multiple interrupts in each frame. Using sprites like this requires more work than usual. For example, to move a sprite down the screen means controlling X and Y for several sprites, some of which won't be displayed because the scan is in the wrong place for them.

Program 12-34. Thirty-Two Sprites

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
1 REM 32 SPRITES DEMONSTRATION PROGRAM      :rem 163
2 REM 828 COUNTS Ø TO 3                     :rem 154
3 REM 829-832 STORES 4 SETS OF SPRITE ENABLES
                                             :rem 61
4 REM 833,834 &C=4 SETS LO/HI RASTERS. (HI=128)
                                              :rem 1
5 REM 841-844 STORES 4 SETS OF X-POSN HIGH BITS
                                            :rem 13Ø
6 REM 845-86Ø,861-876,877-892,AND 893-9Ø8  :rem 54
7 REM STORE ALL 4 SETS OF SPRITES' X,Y PAIRS
                                             :rem 54
8 REM OTHER LINE 1ØØ1 REMOVES COLORED BANDS:rem 25
9 REM                                        :rem 29
1Ø FOR J=49152 TO 49242:READ X:POKE J,X:NEXT
                                            :rem 218
```

```
19 REM ENABLE ALL 32 SPRITES, & SET POSNS  :rem 46
20 POKE 828,3                                :rem 146
30 POKE 829,255:POKE 830,255:POKE 831,255:POKE 832
   ,255                                      :rem 100
40 POKE 833,90:POKE 835,130:POKE 837,170:POKE 839,
   210                                       :rem 42
50 POKE 834,0:POKE 836,0:POKE 838,0:POKE 840,0
                                             :rem 190
60 POKE 841,0:POKE 842,0:POKE 843,0:POKE 844,0
                                             :rem 186
70 FOR K=845 TO 860 STEP 2:READ X:POKE K,X:POKE K+
   16,X                                      :rem 214
75 POKE K+32,X:POKE K+48,X:NEXT              :rem 66
80 FOR J=0 TO 3:READ Y:FOR K=846 TO 861 STEP 2
                                             :rem 134
85 POKE K+16*J,Y:NEXT:NEXT                   :rem 100
99 REM ENABLE RASTER INTERRUPTS              :rem 238
100 POKE 56333,127                           :rem 138
110 POKE 788,0:POKE 789,192                  :rem 157
115 POKE53265,PEEK(53265)AND127              :rem 223
120 POKE 53274,129                           :rem 143
199 REM PUT SPECIMEN MOB IN TAPE BUFFER      :rem 45
200 FOR J=2040 TO 2047:POKE J,15:NEXT        :rem 244
210 FOR J=960 TO 1023:POKE J,255:NEXT        :rem 253
999 REM ML FOR RASTER INTERRUPT HANDLING     :rem 242
1000 DATA 174,60,3,232,224,4,208,2,162,0,142,60,3
                                             :rem 72
1001 DATA 142,33,208,189,61:REM 234,234,234,189,61
                                             :rem 206
1002 DATA 3,141,21,208,189,73,3,141,16,208,138,10
                                             :rem 90
1003 DATA 170,189,66,3,13,17,208,141,17,208,189,65
                                             :rem 166
1004 DATA 3,141,18,208,138,10,10,10,170,160,0,189
                                             :rem 79
1005 DATA 77,3,153,0,208,189,78,3,153,1,208,232,23
     2                                       :rem 202
1006 DATA 200,200,192,16,208,236,169,1,141,25,208,
     173                                     :rem 36
1007 DATA 13,220,41,1,240,3,76,49,234,76,129,234
                                             :rem 50
1999 REM SAMPLE X & Y POSNS OF 32 SPRITES    :rem 16
2000 DATA 40,70,100,130,160,190,220,250{17 SPACES}
                                             :rem 92
3000 DATA 60,100,140,180                     :rem 134
```

At each interrupt, all the X and Y positions are reset, so the sprite positions are all independent. The sprite-enable registers are also reset, so any number of sprites from 0 to 32 can be chosen. The colors, priority, expansion, and so on, are shared between them here to save space. Note the color bands, to show where interrupts occur: Their positions can be changed in lines 40 and 50. The bands are removable.

The ML disassembly is too long for inclusion here. It consists of loops which load new X,Y values, the sprite-enable register, and the timing for the next interrupt.

Motion Without Sprites

Although sprites are powerful, and the screen can be filled with them, you may prefer to PRINT strings of characters (or use JSR $FFD2 in ML) if, for example, you've defined your own graphics. It's fairly easy to move characters one full step in any direction, but the result is inevitably somewhat jerky, though this is less noticeable when bunches of characters move together. Where it's a problem, an improvement is to make up intermediate characters from halves of the original. Following is a simple example routine to give you an idea of how to use the standard graphics that are available on the 64.

Program 12-35. Animation

```
1Ø M$="⁅C⁆⁅F⁆"
2Ø PRINT "{2 SPACES}";
3Ø FOR J=1 TO 4Ø
4Ø FOR K=1 TO 3Ø:NEXT
5Ø PRINT "{2 LEFT} ⁅B⁆";
6Ø FOR K=1 TO 3Ø:NEXT
7Ø PRINT "{LEFT}" M$;:NEXT
```

Program 12-36 makes use of the fact that the checked block can be imitated by another pair of graphics. Resolution is to four dots. This method is unwieldy for a lot of movement in many directions, because any single character needs eight more characters to allow half-character motion in the main directions.

If the movement is one-dimensional, as in games, 80 or so mobile characters can be used. Smooth motion like this requires 15 characters just to move one single character, so only 17 different objects would use the entire set. This is usually less practical than using sprites.

Dynamic redefinition of characters. An advanced method to simulate motion and other effects is to alter the character definitions or bitmaps themselves, so partial characters are generated when needed, rather than being stored. ML is necessary since 16 POKEs to alter the bytes defining two characters would be slow.

Planning is important. First, the original definitions should usually be kept in memory, away from the area that will be redefined, because characters might be irrecoverably changed by dynamic redefinition. Second, characters should be numbered conveniently—generally consecutively down or across the screen—to simplify ML.

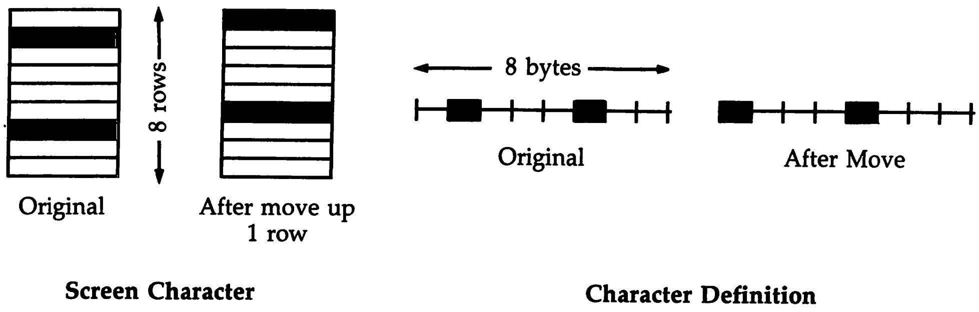
Vertical smooth motion. The following BASIC routine, Program 12-36, demonstrates up-and-down motion in this way. ROM graphics are copied into $3000 up, and screen codes 128–137 (normally reverse-@ through reverse-I) are redefined as numerals 0–9. These numerals, starting at $3400, are processed by ML routines, giving a realistic odometer effect, while retaining the normal numeral definitions. Up-and-down motion is simple to program; as Figure 12-13 shows, all that's necessary is to move over the bytes making up the characters.

Program 12-36. Vertical Motion

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 FOR J=49152 TO 49177:READ X:POKE J,X:NEXT
20 POKE 56333,127:POKE 1,51
30 FOR J=0 TO 511:POKE 12288+J,PEEK(53248+J):NEXT
40 POKE 1,55:POKE 56333,129
50 POKE 56,48:CLR:POKE 53272,29
100 FOR J=0 TO 10:PRINT "{RVS}" RIGHT$("IHGFEDCBA@
    ",J);"{UP}":NEXT
110 FOR J=0 TO 79:POKE 13312+J,PEEK(12672+J):NEXT
120 FOR J=0 TO 79:SYS 49152:NEXT
130 FOR J=0 TO 79:POKE 13312+J,PEEK(12672+J):NEXT
140 FOR J=0 TO 79:SYS 49164:NEXT
150 GOTO 100
200 DATA 162,79,189,254,51,157,255,51,202
210 DATA 208,247,96,162,1,189,0,52,157
220 DATA 255,51,232,224,80,208,245,96
```

Figure 12-13. Vertical Character Motion

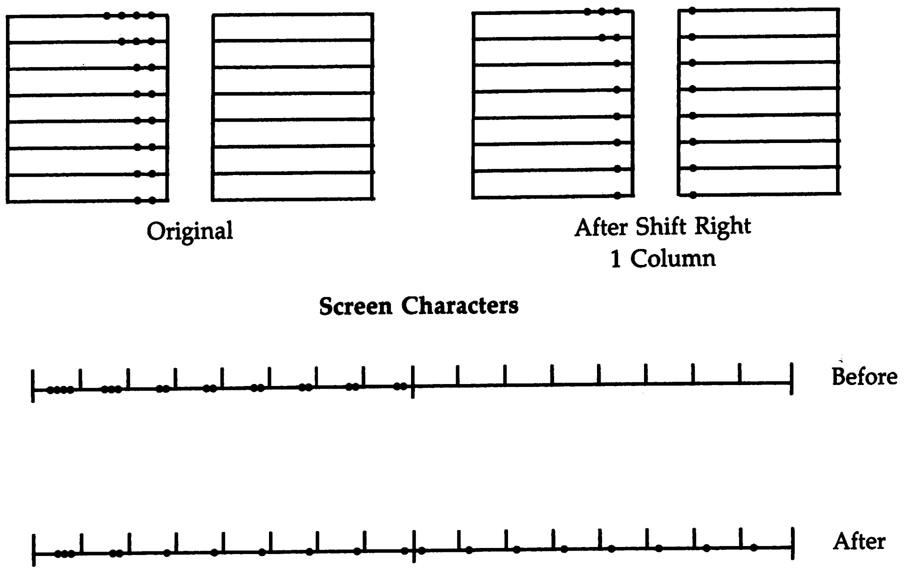


SYS 49152 uses the loop below, which moves only the required portion of the character definitions. (You may need to add a zero byte at the end points.)

```
     LDX  #$4F
LOOP LDA  $33FE,X
     STA  $33FF,X
     DEX
     BNE  LOOP
     RTS
```

Horizontal smooth motion. Because of the way screen and character memory are allocated, this is trickier. It is necessary to number the relevant screen locations consecutively. As Figure 12-14 shows, you must store a bit from each byte and rotate it into another byte eight bytes away.

Figure 12-14. Horizontal Character Motion



Replace the following lines in Program 12-36 and run it. Program 12-37 will produce a similar effect, as the special numerals 0–9 scroll smoothly sideways.

Program 12-37. Horizontal Motion

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 FOR J=49178 TO 49205:READ X:POKE J,X:NEXT
20 POKE 56333,127:POKE 1,51
30 FOR J=0 TO 511:POKE 12288+J,PEEK(53248+J):NEXT
40 POKE 1,55:POKE 56333,129
50 POKE 56,48:CLR:POKE 53272,29
100 FOR J=0 TO 10:PRINT "{RVS}" RIGHT$("@ABCDEFGHI
    ",J);"{UP}":NEXT
110 FOR J=0 TO 79:POKE 13312+J,PEEK(12672+J):NEXT
120 FOR J=0 TO 79:SYS 49178:FORK=1TO99:NEXT:NEXT
150 GOTO 100
200 DATA 160,7,152,170,24,126,0,52,8,138,24
210 DATA 105,8,170,224,79,176,5,40,208,240
220 DATA 240,238,40,136,16,231,96
```

Chapter 13

Sound

- Sound Waves: Analysis and Synthesis
- The SID Chip
- Music Theory
- Sound Demonstrations

Chapter 13

Sound

This chapter explains sound and music on the 64 and shows how the SID chip handles both in practice. Although the theoretical side is important for full understanding, it is also quite difficult, so you may prefer to try the programs first to get the feel of SID.

Sound Waves: Analysis and Synthesis

Sound is a psychological phenomenon produced by vibrations. The pressure waves that produce sound are three-dimensional and are mainly concentric waves in air centered on sound sources. Additional complications arise due to conductance, reflection, and so on, which cause reverberation, echo, and other acoustic effects. Any elastic medium (such as metal, bone, or water) propagates sound at a speed depending on the medium's elasticity—faster in warm air than in cold air, and much faster in metal. The medium's molecules oscillate with very little net movement, and this motion gives rise to compressions and rarefactions which transmit the wave. The wave moves a lot, but the medium moves very little.

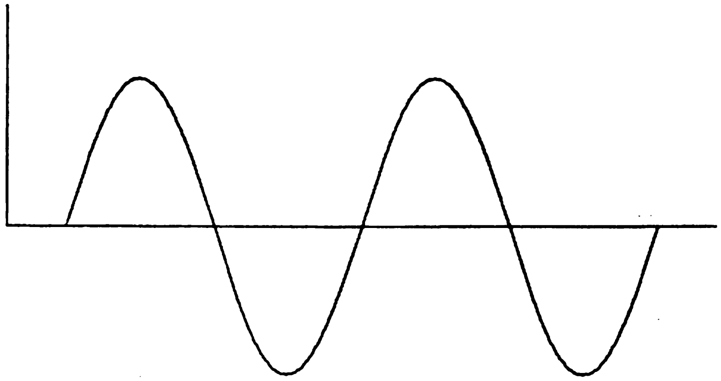
Waves are usually depicted by graphs which show how the back-and-forth movements vary with time: A *cycle* of a regular wave is the interval from one point on the wave to the next similar point in the repeating wave pattern. The *frequency* of a wave is its number of cycles per second. One *hertz* (abbreviated Hz) means the same as cycles per second. The distance in magnitude between the peak and the trough of a wave is known as its *amplitude*. The larger the amplitude of an audible wave, the louder the sound will be. The process of perceiving sound from pressure waves is performed by the ear and brain. Frequency, a physical property of the vibration, is related to the psychological property of pitch: high frequency is perceived as a high note and low frequency as a low note. The maximum range of frequencies audible to human beings is about 20–20,000 Hz, though the actual range varies between individuals and generally decreases with increased age and exposure to noise. These frequencies are determined by such factors as the size of the eardrum, which resonates when vibrated.

Sounds with a distinct, steady pitch, produced, for example, by tuning forks and some musical instruments, result from the repetition of similar vibrations. A note may sound very different on different instruments; this quality, the *timbre*, depends on the relation between its *fundamental* (lowest) frequency and its *harmonics*.

Sine Waves

Sine waves are important for two reasons. First, any note in any timbre can be analyzed by breaking it into its component sine waves, and this analysis gives a common basis of comparison between all notes and timbres. This makes the results of electronic sound processing predictable. The second reason is psychological, in that a similar process of analysis happens in the brain. As a result, we can think of the timbre of the sine wave as the simplest and purest sounding of any waveform. Figure 13-1 outlines a sine wave.

Figure 13-1. Sine Wave



Circular motion generates sine curves; the SIN function describes this motion. As it happens, the prongs of a sounding tuning fork give a good approximation of sinusoidal motion. In fact, simple harmonic motion derives its name from the sound analogy.

A tuning fork is a simple musical instrument. It works the way it does because a struck prong is pushed toward its still position by a force that varies with (among other things) the distance from its still position, and it can also be shown that this produces a sine curve. This is a typical application of physics to sound.

Consider a tuning fork with a smaller tuning fork attached to one prong, so the movement of both forks contributes to the movement of the small fork. The composite motion will be two sine waves added together.

If the smaller fork's frequency is exactly 2, 3, 4, or any integer times the frequency of the larger, then the ear-brain mechanism fuses them into a harmonious tone, of richer timbre than a pure sine wave. As stated above, sine waves which are multiples of a sine wave are known as harmonics of that sine wave. The fundamental frequency is known as the first harmonic, the wave at twice this frequency is known as the second harmonic, that at three times its frequency is the third harmonic, and so on. Most musical instruments are designed to generate harmonics, and the relative importance of harmonics determines the instrument's timbre. For example, a closed air column can vibrate stably along its full length, or half its length, or one-third, and so on, and therefore has a full range of harmonics; a string, plucked in the center, cannot easily vibrate for half its length and has only odd harmonics.

Harmonic analysis is the process of determining the sine wave components that make up a waveform. Fourier analysis, a popular technique, is based on the principle that any repeating function f(x) is of form b1 sin (x) + b2 sin (2x) + . . ., where the integral from minus pi to pi of f(x) sin (kx) is bk. The idea isn't new, and ancient astronomers built up ellipses from many circular movements. The harmonics and their relative amplitudes make up the harmonic spectrum of the waveform, which can be drawn on a graph as amplitude against frequency.

Conversely, any repeating waveform, like that from the two tuning forks, can be built up by adding harmonics with the right amplitudes: this is *additive synthesis*. The process of starting with timbres which are rich in harmonics and deriving a sound with a desired spectrum by filtering out the unwanted ones is known as *subtractive synthesis*.

The SID chip isn't designed to generate sine waves, but as we'll see, it does generate quite complex waves with rich spectra and also allows limited subtractive synthesis, since it has a filter.

Waveforms and Their Harmonic Contents

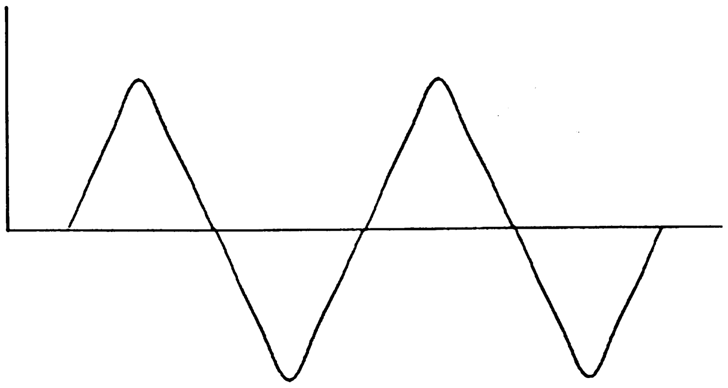
The SID has four main waveforms: noise, pulse (square), sawtooth, and triangle. You can view the SID's outputs by modifying the "ADSR Plotter," later in this chapter. Use PEEK (SID+27) to read voice 3's wave output, decrease SID+15 to reduce the frequency, and POKE SID+18 with 129, 65, 33, or 17 for noise, pulse, sawtooth, and triangle waves, respectively (setting SID+16 and SID+17 when trying the pulse wave).

Following is a discussion of the three repeating (and one nonrepeating) waveforms that the SID chip generates.

Triangle

The triangle wave (see Figure 13-2) is SIN (X) − SIN (3*X)/9 + SIN (5*X)/25 − SIN (7*X)/49 + . . . where X is the harmonic number. A triangular wave with 100 Hz as its fundamental contains frequencies of 300, 500, 700, 900 Hz, and so on, which rapidly decrease in importance.

Figure 13-2. Triangle Wave

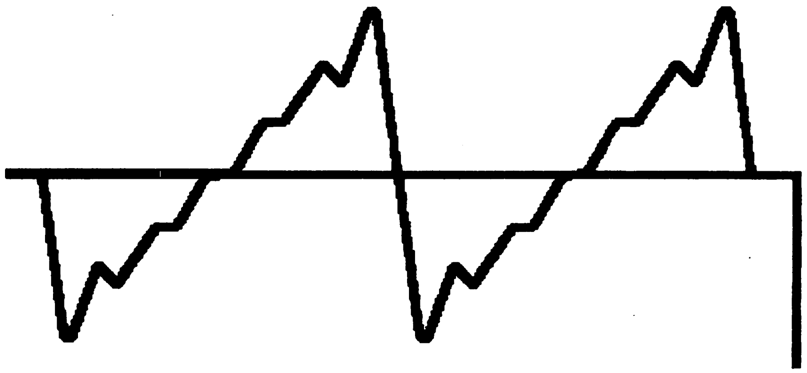


This is SID's closest approximation to a sine wave, and it sounds like a flute or xylophone. Four sinusoidal harmonics—the first, third, fifth, and seventh—add to form an approximately triangular waveshape.

Sawtooth

The sawtooth wave is SIN (X) + SIN (2*X)/2 + SIN (3*X)/3 + SIN (4*X)/4 + ... It contains the complete harmonic series, with amplitude in inverse proportion to the harmonic number, so the second harmonic has half the amplitude of the fundamental, and so on. The harmonics have more importance than the triangular wave's: Figure 13-3 adds six harmonics to provide an approximation.

Figure 13-3. Sawtooth Wave



The large numbers of significant harmonics enable this wave to simulate such instruments as the trumpet, oboe, clarinet, and accordion, especially when filtered to change the harmonic balance.

This wave's asymmetry causes a few oddities. For example, every other term of its analysis makes up a square wave: if this is subtracted, what's left is SIN (2*X)/2 + SIN (4*X)/4 + . . ., another sawtooth wave of twice the frequency and half the amplitude. Add the square wave to the sawtooth on the diagram to see this. The sawtooth also tends to sound higher than its fundamental would suggest.

Pulse (Square) Wave

This is the most useful waveshape provided by SID. The signal is alternately held high for a measured time period, then low for another, generally different period. In other words, there is a regular pulse. The ratio of the time the signal is high to the complete cycle is the *duty cycle* of the pulse wave.

A square wave is the special case in which the duty cycle ratio is 1:2. It analyzes into SIN (X) + SIN (3*X)/3 + SIN (5*X)/5+ . . . . Therefore, like the triangular wave, it contains only odd harmonics, but with larger amplitudes, inversely related to the harmonic number. The third harmonic has one-third the amplitude of the fundamental, for example. Figure 13-4 shows a wave constructed from the first, third, fifth, and seventh harmonics.

The pulse wave's interesting feature is the way its harmonic content changes when the duty cycle is altered. Full analysis is tricky. The Kth harmonic varies with (COS(NK)−COS (K*PI))/K, but generally if the duty cycle is 1:N, then every Nth harmonic will be absent, and harmonics between these suppressed harmonics will be boosted. Very asymmetrical pulse waves have very irregular harmonic spectra, with some large amplitudes in the harmonics. The square wave, with N=2, lacks all even harmonics, as we've seen.

A wave of duty cycle 1:5 lacks fifth and tenth harmonics, and so on, but does have second, third, fourth, sixth, seventh, eighth, and ninth harmonics, with the seventh and eighth harmonics of higher amplitude than the sixth and ninth. A duty cycle of 1:5.1 will cause the fifth, tenth, etc., harmonics to be much reduced in amplitude, but not totally negated.

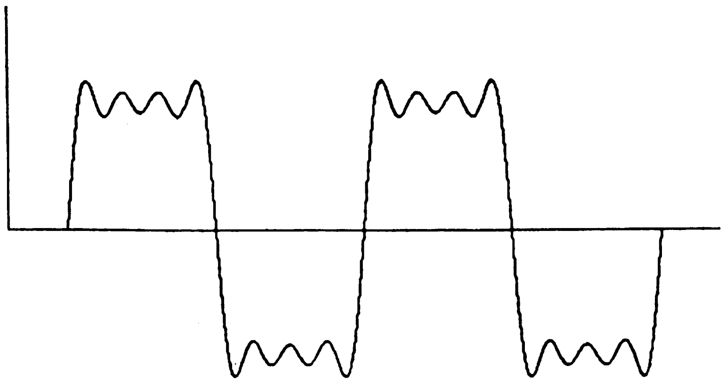
Altering the duty cycle while a note is playing causes most harmonics to change their levels, and some to vanish altogether or reappear. Thus, we have a way of producing a dynamically changing spectrum with richer, more interesting sounds than the triangle or sawtooth waves.

If the pulse width is altered from square toward a narrow pulse, loudness diminishes, and the timbre becomes more nasal or buzzy compared with the well-rounded sound of the square wave, as some high harmonics are boosted. As the width of the pulse becomes very narrow, the decreasing amount of energy present is spread out among very many harmonics, and the sound fades to inaudibility.

The square wave is louder than any other pulse wave of equal amplitude, mainly because the total energy in a square wave is greater: A very short pulse is simply quieter. A secondary factor is the energy distribution amongst the harmonics: a square wave's harmonics taper off smoothly, but a pulse wave with duty cycle of 1:2 has a very irregular distribution, and some boosted harmonics may be inaudible.

Typical sound simulations are a piano (square wave), organ (1:4 wave), and banjo (1:10 wave). Very narrow pulses at low frequency give a car engine sound.

Figure 13-4. Pulse Wave



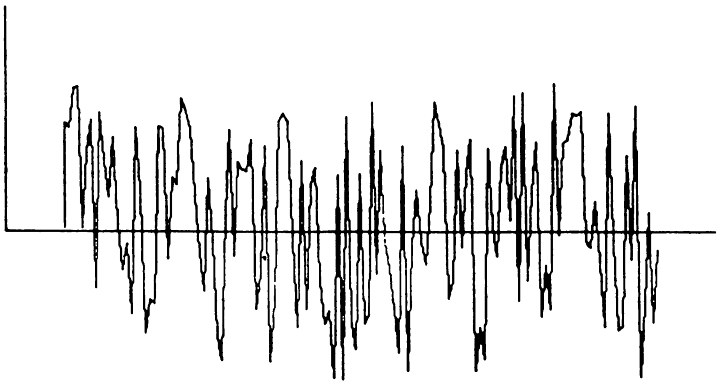
Noise

Noise is sound with no fundamental frequencies and is typically generated by an unmusical source which has no dominant modes of vibration. Rumbles and hisses, crashes and explosions, scrapes and rattles, jet engines and gas burners, wind and water flow, illustrate this sort of sound.

SID generates noise by loading new random values into the oscillator output, at regular intervals controlled by the frequency setting. While the result cannot be analyzed into repeating sine waves, any single interval can be analyzed, so the ear gets a definite overall feeling of low, medium, or high pitch. *White noise* contains all frequencies in equal proportion. The SID's noise generation method sets a minimum frequency and plays all available notes above this minimum with equal probability, so most SID noise is biased to high frequencies; this is called *blue noise.*

Noise (diagrammed in Figure 13-5) is useful in simulating the sorts of sounds mentioned above, and also certain percussion sounds like snare drums, brushed drums, and cymbals. Dynamic changes in noise frequency can simulate ripping, tearing sounds, fireworks in motion, and moving vehicles.

Figure 13-5. Noise



Ring Modulation and Synchronization

Bells, gongs, chimes, clamped metal bars, and so on, vibrate differently from strings and air columns, generating nonsinusoidal waves, which analyze into complex waveforms containing several series of fundamentals and harmonics.

Ring modulation simulates this process. The principle isn't too difficult, and understanding it makes it easier to achieve the desired effect.

Ring modulation obtains a complex signal by combining two input signals. The process is simply a point-by-point multiplication of one input signal by the other. The trigonometric identity SIN(F1) * SIN(F2) = −SIN (F1+F2+90)/2 + SIN (F1−F2+90)/2 gives a clue as to what happens; two sine waves are combined, giving two new sine waves with frequencies F1+F2, of reduced amplitude and different phase.

As mentioned, the ring modulated output from two sine waves with frequencies F1 and F2 consists of sine waves with frequencies (F1+F2) and (F1−F2). The input signals aren't present in the output. So if one of the input signals to the ring modulator has a frequency of 1 Hz, the output is two signals of F+1 Hz and F−1 Hz, which should be heard as a tremolo effect, because notes which are close together pulsate in loudness. Piano tuners use this beat effect. Its frequency is half the difference between the component frequencies. In fact, the SID cannot generate very slow frequencies satisfactorily; you'll get clicks and pops rather than pulsations, so tremolo is better achieved by controlling volume.

When one or both inputs contain harmonics, each harmonic gives rise to a sum and difference signal with each harmonic of the other input. SID's ring modulator can be used only when the triangular wave is selected for the ring modulated voice (even though this waveform is less rich in harmonics than others). Ring modulation is controlled by setting the frequency of the modulating voice; other parameters of the modulating voice are irrelevant to the effect.

Consider two frequencies, a and b. The triangular waves have harmonics a, 3a, 5a, . . . (and b, 3b, 5b, . . .). Combining only the first three harmonics of each (to save space) gives:

Adding	a+b	a+3b	a+5b
	3a+b	3a+3b	3a+5b
	5a+b	5a+3b	5a+5b
Subtracting	a−b	a−3b	a−5b
	3a−b	3a−3b	3a−5b
	5a−b	5a−3b	5a−5b

This shows there are two triangular waveforms based on (a+b) and (a−b); look at the main diagonals to see this. There are also many extra waves, with such frequencies as 3a+b and 5a+3b.

Consider the case where b is very small. The result will be many sine waves, each of frequency approximately a, many of frequency about 3a, and so on. The result is a modified triangular wave, where all the harmonics beat.

In the special case where a is a multiple of b, the output is a harmonic series. For example, if 100 Hz is input with 300 Hz, all the sum and difference signals must be multiples of 100 Hz. The result is a distinctly pitched note, typically something like a square wave. If the frequencies aren't quite exact multiples, the output is a pair of enhanced triangular waves which will beat, since their fundamentals aren't quite a ratio. This gives realistic banjo twangs or rubber-band "boings."

Where one input changes continuously relative to the other, some frequencies (those produced by addition of harmonics) slide up, and the others (produced by subtraction) slide down, and the output passes through points where it has a single pitch.

Bell-like timbres are produced with such inputs as 110 and 152 Hz, which give two audible notes (42 Hz and 262 Hz) of unrelated frequency.

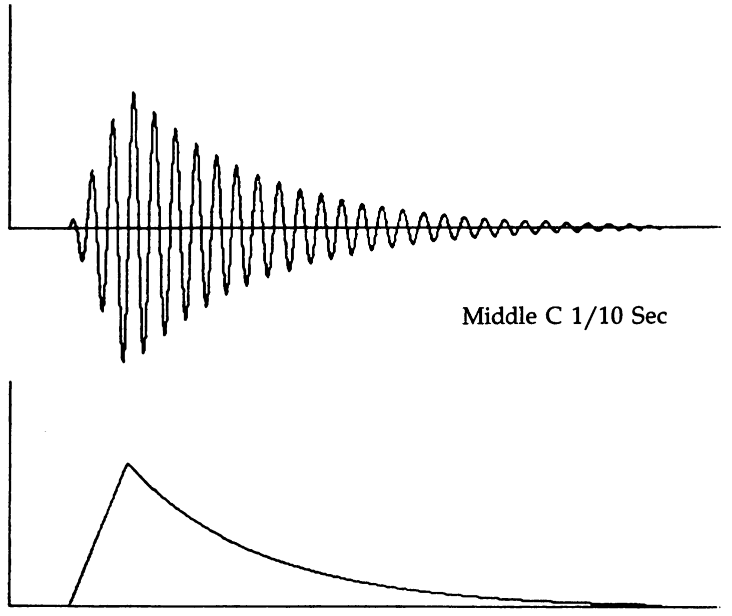
Synchronization also forms a joint output from two inputs, but in a different way. The frequency and waveform remain virtually unchanged. However, at intervals decided by the inputs, the waveform restarts. The effect is to add high harmonics to the wave, while keeping frequency fairly constant.

Envelopes

So far we've discussed steady tones, produced by stable waveforms. A note's *envelope* describes the rise and decay of the note's loudness against time. The time scale of SID envelopes is much longer than for waves.

Figure 13-6 shows the envelope of a triangular wave over a tenth of a second. The envelope has a short *attack* phase, and a much longer *decay* phase. The waveform is independent of the envelope, but it's easy to forget the distinction.

Figure 13-6. Envelope Diagram



The SID chip's programmable envelope has four parts:

- **Attack.** The time in which the note reaches full volume from zero. Typically very short.
- **Decay.** Allows the volume to be reduced from its initial maximum to a steady level.
- **Sustain.** Period where the volume is steady, like the waves discussed earlier in this chapter.
- **Release.** Period during which amplitude falls from the sustain level to zero.

The above parameters help the programmer emulate sounds by controlling the envelope. Figure 13-7 gives some illustrations of how ADSR envelopes might be used.

Energy is used to generate sound. In a system radiating sound with a relatively constant loudness, like an organ or violin, energy must be input as long as a steady note is desired. With notes produced by a piano or a guitar, though, energy is applied once, to dissipate as the note dies away.

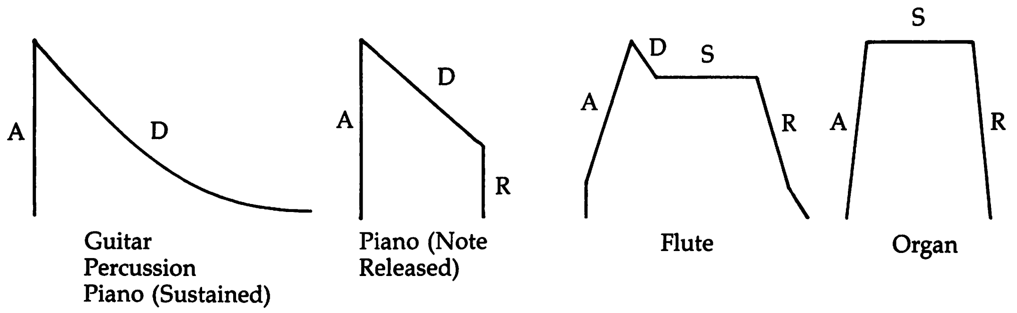
The guitar envelope has zero attack time, decay starts immediately, and there is no sustain level. A triangular wave approximates its odd-harmonic wave. This envelope is characteristic of cymbals, bells, drums, and many percussion instruments.

A piano envelope is similar, except that a piano key may be released, and if it is, decay is interrupted and the release phase terminates the note. A square wave approximates the piano's timbre.

The organ has a noticeable attack and decay, as the note is established or fades. While a key is pressed the sustain level is held, after which there's a short fade-out. A pulse wave gives about the right timbre.

A flute has a longer attack and release, and has roughly a triangular waveform.

Figure 13-7. Common Envelope Shapes

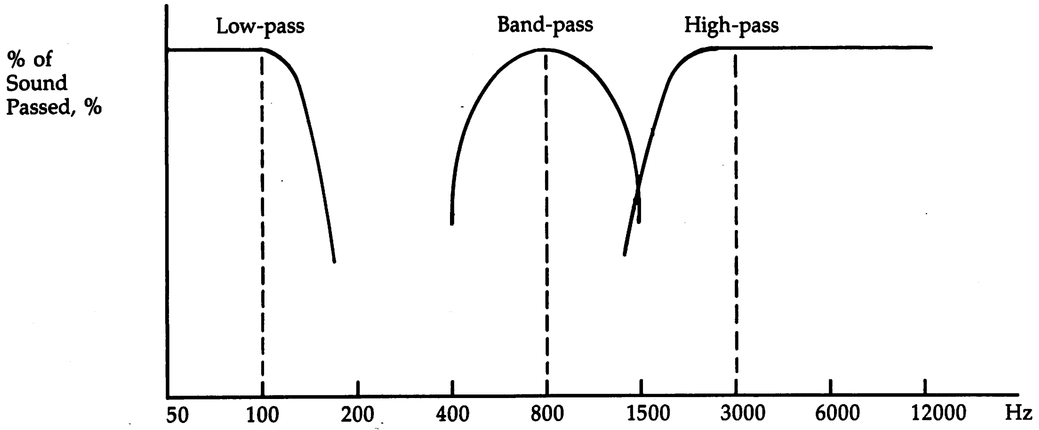


Filtering and Resonance

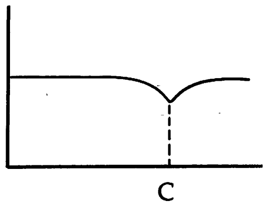
Filtering in electronic music means removing a range of unwanted frequencies. The result can be predicted by analysis of the input waveform: for example, a *low-pass* filter passes low harmonics, but cuts out high ones, cutting treble and giving a more bass tone. This is the most common type of filter. The level where it begins to act is called its *cut-off* frequency. A high-pass filter passes frequencies only above the cut-off frequency, and a band-pass filter passes a narrow band of frequencies, removing those not in the vicinity of what's called the center frequency. The cut-off isn't always sharp. The SID's low-pass filter, for example, reduces perceived volume by about half for each octave above cut-off.

A *notch* filter takes effect when low- and high-pass filters are on together. The SID's filter type is controlled by three bits, each of which can be on or off, so the SID can be set as a notch. Resonance amplifies or boosts frequencies close to the cut-off point. SID's filter has 16 linear resonance settings. See Figure 13-8.

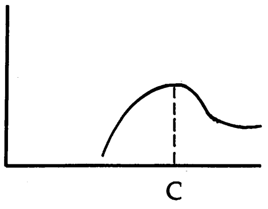
Figure 13-8. Filters



Low-pass filter with cut-off frequency 100 Hz
Band-pass filter with center frequency 800 Hz
High-pass filter with cut-off 3000 Hz



Notch



High-pass filter with resonance

Synthesis Notes

Any waveform can be synthesized by sampling (by measuring the sound's amplitude at a series of points) and simply duplicating the sequence. This is not how most synthesizers work, though they generate sawtooth, noise, and variable pulse waves, and have controls for such things as vibrato and chorus (where all voices are turned on in approximate unison). ADSR envelopes are often used, the release phase being entered when a key is no longer pressed. The Commodore 64 SID chip, designed by a specialist in music synthesis, has many of these features.

Sounds produced by acoustic instruments tend to have extremely complex changing harmonic structures, especially in the attack phase. And harmonics aren't absolutely perfect. Their frequencies aren't quite exact multiples of the fundamental frequency. For these reasons, it's difficult to make electronic instruments sound just like acoustic ones.

Speech Synthesis

The vocal cords generate a roughly triangular wave of fundamental 70 (men) or 100 (women), which resonates in the throat, mouth, and nose and is shaped by the tongue, palate, and lips. Speech is typically synthesized from three main frequencies (*formants*), plus low-frequency noise (*nasals*) and high-frequency noise (*fricatives*). For example, "oo" is roughly a sine wave, and "ee" a sine wave plus a 3000 Hz tone. The frequencies of formants typically change somewhat in the course of vowel sound. Consonants are often bursts of noise which are abruptly cut off. Since SID has only three voices, no sine wave capability, and a single filter, speech is not easy to synthesize with the 64, although it is possible.

Plug-in speech cartridges usually include their own speech chip, bypassing SID and providing a known quality of output. Such chips typically store waveforms of 64 component parts of speech sound (*phonemes*) and allow control of such parameters as duration. Their software converts some form of phonetic spelling into phonemes; for example, A$="H/E/LL/OO/P2" may define a string which sounds like "hello" followed by a pause. Standard words and sounds and the alphabet may be supplied in ROM. Commands are either wedged into BASIC or accessed by SYS calls.

The SID Chip

Overview of the SID Chip

Before a more technical discussion of the SID chip, here is a plain English description. The SID will generate a range of waveforms, as discussed above. These waves are always output in an ADSR envelope. In other words, the chip is designed to produce individual notes, of programmable timbres and frequencies. Steady notes are obtainable in the extreme case where sustain is simply left on. Accurate direct control of the output signal isn't part of the SID chip's function. The next best option is to set a pulse wave at its far range so that it never actually pulses and to alter the sustain level.

Notes are controlled like an organ: individual notes must be consciously turned on and turned off. To play a full ADSR note therefore requires two POKEs or ML stores—one to start it, the other to trigger the final release phase.

To save money and memory, Commodore didn't incorporate commands like SOUND or ENV into BASIC for the 64. Consequently, programming sound involves many POKEs. Even the simplest sounds require four POKEs.

It's fairly easy to play multipart tunes with the SID chip. Once each voice has its envelope set up and waveform selected, all that's needed is to periodically change a note's pitch, turn the note on, wait for some suitable sustain interval, turn the note off, and wait for sufficient release time. The automatic timing of attack, decay, and release phases saves some work. If you're happy to do without a sustain phase, a note can be made up of attack and decay components only; this simplifies programming. BASIC programs are slow enough for some of the delays to occur involuntarily. This is the usual way the SID chip is programmed; it assumes that the envelopes are satisfactory. Dynamic effects such as raining or ripping sounds, the Doppler effect, and envelopes controlling filters rather than volume require more work and probably ML subroutines as well.

The SID chip (officially, the 6581 Sound Interface Device) has 29 eight-bit registers, numbered 0–28 in its specification. The SID chip starts at $D400 (54272) in the 64. Address decoding is incomplete, and 31 extra images of the SID chip registers appear at $D420, $D440, and so on. The SID images are a side effect of the way the SID chip is wired into the 64; they are not designed to be used at all. While it is possible to use image registers in place of the actual SID registers, there is no real advantage in doing so, and the practice is likely to confuse anyone else who looks at such a program.

The SID chip has three electronic voices. They may be used independently, or linked, giving ring modulation from two voices, for instance. Each voice has an oscillator, able to generate sawtooth, triangular, variable-width pulse, noise, and other combination waveforms, of frequency ranging from about 1/20 Hz to 4000 Hz, in extremely small steps. Much higher frequencies than this relatively low top limit are generated by harmonics. Each voice has its own envelope shaper, with programmable attack, decay, sustain, and release phases. Any of the three voices may be routed together via the SID chip's single filter, which has programmable low-pass, high-pass, and band-pass modes, resonance, and cut-off or center frequency. The often published block diagram of the SID chip charts all these features.

Most SID registers are write-only; PEEKing returns 0. Consequently, monitor-style programs need an array to keep track of values which have been POKEd in.

However, the last four registers are read-only. Two of them return potentiometer readings, used with game paddles, drawing tablets and other devices. These are dealt with in Chapter 16. Although sound could be controlled by them, perhaps as an alternative to the keyboard, they're not directly relevant and we'll say little more about them here.

The two other read-only registers return, respectively, the oscillator and the envelope, as output by voice 3. We can use these to inspect the SID chip's waveforms and envelopes, and for such control purposes as modifying oscillator frequencies for vibrato or siren effects, or changing filter frequency to get a wah-wah effect.

The SID chip produces a reasonably high quality electrical signal, obtainable on the audio-video socket at the rear of the 64 (pin 3 is the signal, and pin 2 is the ground). Play this signal through an amplifier and speaker, preferably with some treble removed, for a significantly higher quality sound than is available via your TV. External input can also be processed by the SID chip: the signal, to pin 5, must be in the range 5.7 to 6.3 volts DC, matched to the SID chip's 100K ohm impedance. The signal can be filtered and mixed with output from the SID chip's voices, but not processed in any other way. If you're not sure how to make such external connections, get help from someone who understands electronics. The SID chip can be damaged by an external input that exceeds its design levels.

Voice Generator Registers

The three voice's registers are arranged in identical patterns of seven bytes, so you may find it mnemonically helpful to use expressions like SID=54272 and HF=SID + VOICE*7 + 1 in BASIC. The second example sets the high byte of the frequency of one of the voices, depending on the value of VOICE (which must be in the range 0–2).

Frequency control registers (offsets of 0 and 1, 7 and 8, 14 and 15). The first two registers of each group hold a two-byte value which controls the frequency of the note produced by that voice. The frequency is directly proportional to the value, which means that doubling the value doubles the frequency. The frequency resolution is extremely good. There are about 250 values between middle C and the next semitone, permitting glissandos (pitch changes between notes with no perceptible steps), though resolution is not as good at extremely low frequencies.

The registers' values increment every 256 clock cycles, setting a new output value whenever 0 is reached; this explains how the timing works out.

The 16-bit register value is related to the true frequency like this:

U.S. True frequency = 0.0609597 * Register value
U.K. True frequency = 0.058717 * Register value

Or the other way around:

U.S. Register value = 16.40426 * True frequency
U.K. Register value = 17.0309 * True frequency

To produce a tuning standard note A (frequency 440 Hz) in the U.S., the value is 440*16.404261, or about 7218. Note that U.S. software plays at a lower pitch in the U.K.

The highest obtainable pitch is about 0.0609597*65535 = 3995 Hz, about four octaves above middle C, and the lowest about 0.06 Hz, far below audibility. Two or three octaves below middle C are perceptible as a pitched sound, so the SID chip's total range of pitched sounds is about six or seven octaves.

If the variable F contains a value to be placed into these registers, then H%=F/256 then L%=F−H%*256 will assign the high and low bytes to H% and L%.

Pulse width registers (offsets of 2 and 3, 9 and 10, 16 and 17). The pulse width is set by a 12-bit value. The lower 8 bits are set by the first of these register pairs, and the upper 4 bits by the least significant nybble of the second register, leaving the upper nybble unused. So the maximum value it may take is 4095. The duty cycle is given by the ratio of pulse width: 4095. A duty cycle of 1:2 (a square wave) results from 2048 (8 in the second register, 0 in the first). The value 0 or 4095 gives a constant DC voltage. Consequently, if a pulse waveform is to be audible, a pulse width must be set in addition to a frequency. The pulse width registers have no effect on other waveforms.

Note that a 1:4 duty cycle sounds identical to a 3:4 duty cycle. It's the same waveform upside down, so the value 1024 will produce the same harmonic content as 3072.

However, if other tones are sounding simultaneously, they'll interact with the pulse wave, and there may be a noticeable difference between 1:4 and 3:4 ratio pulse waves. In particular, if a pulse wave and another waveform are enabled together in a control register (see next section), a low ratio gives an inaudible result, while a high ratio creates a much louder signal.

To load the contents of the variable PW into the SID chip, POKE the first register of the pulse-width register pair with PW/256, and the second with PW AND 255.

Control registers (offsets of 4, 11, and 18). The fourth register in each group has eight more-or-less independent bits; as usual, each is off when 0, on when 1. In ordinary use, only one waveform will be enabled in a voice at any given time. Enabling several waves (but not noise) in the same voice logically ANDs the outputs from point to point. A square wave cuts out half the other wave, giving variations on the sawtooth, but ANDed triangular and sawtooth waves mostly cancel out. A very wide pulse adds high frequencies without cutting volume too much. The following list describes the function of each bit of the control registers:

Bit 7: Noise.

Bit 6: Pulse waveform. Pulse width must be set.

Bit 5: Sawtooth waveform.

Bit 4: Triangular waveform.

Bit 3: Test bit. This suspends the voice; when cleared to 0, the output is continued. This can be useful where the exact realtime phase of a waveform is important. But the most common use is probably to restart a voice, locked up after noise has been enabled with another waveform.

If noise is selected with another waveform, the noise waveform is silenced and will remain silent until the test bit is set to 1 and then reset to 0.

Bit 2: Ring modulation. When this bit is set to 1, its voice's output (which must be a triangular wave) is replaced by the ring modulated signal obtained from its own signal and the previous voice, which is automatically treated by the ring modulation process as a triangular wave. Only the frequency setting of the previous voice has any effect. *It isn't necessary to set the waveform or even turn on the voice.* Both voices can still be used: voice 3 might play noise, while voice 1 could play a ring modulated signal based on the triangular wave of voice 3's frequency setting.

A common combination is to set the ring modulation bit of voice 1 and use voice 3 for the other input signal. The voice before voice 1 is treated as voice 3. If two or even three ring modulation bits are on, and appropriate triangular waveforms have been selected, still more elaborate ring modulation occurs.

Bell timbre notes can be tuned by multiplying both input frequencies by 1.059463, or using tables, to find semitone values which retain the same pattern of harmonics. Glockenspiel and vibraphone effects can be obtained in this way.

Bit 1: Synchronization bit. This links two voices just as the ring mod bit does. As with ring modulation, only the frequency of the previous voice need be set for sync to operate. There are two small differences: the voice with sync bit set can have any waveform, and the synchronized output frequency is determined by the earlier register, the one with sync bit not set.

Synchronization can be used in several ways. It provides an easy way to vary timbre. Suppose voice 1 plays a tune on its own. If voice 2's sync bit is set, voice 1 still plays the tune, but you can get new timbres on replaying the tune by simply POKEing a new value into voice 2's frequency.

With ring modulation, sync adds more high frequencies and may improve the sound. It's probably always worth trying. However, voice 3 alone, not voices 1 and 3 together, now controls the pitch.

Bit 0: Gate bit. This controls the playing of the voice's note. It has two functions. Transition from 0 to 1 immediately starts the ADSR envelope. Transition from 1 to 0 immediately starts the release phase of the ADSR envelope.

Both events take place irrespective of the stage reached by the current ADSR sequence in progress. For example, if the gate bit is turned off during a slow attack, release starts without decay or sustain phases.

Envelope shape registers (offsets of 5 and 6, 12 and 13, 19 and 20). These registers control the attack, decay, sustain, and release phases of the sound envelope. Each of the envelope register pairs is arranged as a four-nybble group in A, D, S, R order. Thus, the high nybble of the first register controls the attack, while the low nybble controls the decay. In like fashion, the high nybble of the second register is for sustain, and the low nybble determines release.

Program 13-1 allows you to watch the SID chip generating envelopes. It reads the envelope generated by voice 3 and draws the result on the screen in a distinctive color, so several envelopes can be overlapped and compared. The envelope is timed so a BASIC loop can read it. Try entering ADSR values such as 3, 3, 3, 3 and 7, 5, 5, 6 to get the idea.

Note that A, D, and R set rates of increase or decrease, but sustain is a level, so a high S value raises the level portion of the envelope. Release is timed by the program to occur about 2/3 the length of the axis.

Program 13-1. ADSR Plotter

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 SID=54272:DIM X(500)                        :rem 212
20 INPUT "ATTACK, DECAY, SUSTAIN,RELEASE"; A,D,S,R
                                               :rem 233
25 FOR J=SID TO SID+24:POKE J,0:NEXT           :rem 81
27 POKE SID+24,15                              :rem 155
30 POKE SID+19,16*A+D:POKE SID+20,16*S:REM A,S,D
                                               :rem 168
40 POKE SID+14,11                              :rem 145
42 POKE SID+15,48                              :rem 158
50 POKE SID+18,0:POKE SID+18,33                :rem 211
60 FOR J=0 TO 25:X(J)=PEEK(SID+28):NEXT        :rem 220
70 POKE SID+18,0                               :rem 102
80 FOR J=26 TO 40:X(J)=PEEK(SID+28):NEXT       :rem 19
90 Z=(Z+1) OR 8 AND 15:POKE 646,Z              :rem 168
100 GOSUB 200:FOR J=0 TO 40                    :rem 130
110 FOR K=1 TO X(J)/20:PRINT "{RVS} {UP}{LEFT}";:N
    EXT                                        :rem 88
120 GOSUB 200:FOR K=1 TO J:PRINT "{RIGHT}";:NEXT:N
    EXT                                        :rem 193
130 PRINT "{WHT}{11 SPACES}ADSR=" A;D;S;R:GOTO 20
                                               :rem 129
140 PRINT "{HOME}";:FOR X=1 TO 24:PRINT "{DOWN}";:
    NEXT:RETURN                                :rem 142
200 PRINT "{HOME}";:FOR X=1 TO 24:PRINT "{DOWN}";:
    NEXT:RETURN                                :rem 139
```

The attack/decay register (5,12,19) consists of two nybbles which hold the attack value (0–15) and decay value (0–15). POKE 16*A+D.

The sustain/release register (6,13,20) consists of two nybbles which hold the sustain level (0–15) and release value (0–15). POKE 16*S+D.

Table 13-1 is a list of approximate attack times and maximum rates for decay and release.

Table 13-1. Envelope Register Values

Value		Attack Rate	Decay/Release Rate
Dec	Hex		(Maximum time)
0	0	2 ms	6 ms
1	1	8 ms	24 ms
2	2	16 ms	48 ms
3	3	24 ms	72 ms
4	4	38 ms	114 ms
5	5	56 ms	168 ms
6	6	68 ms	204 ms
7	7	80 ms	240 ms
8	8	100 ms	300 ms
9	9	250 ms	750 ms
10	A	500 ms	1.5 seconds
11	B	800 ms	2.4 seconds
12	C	1 second	3 seconds
13	D	3 seconds	9 seconds
14	E	5 seconds	15 seconds
15	F	8 seconds	24 seconds

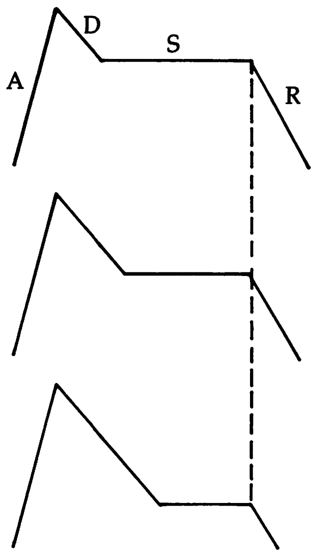
Attack times are 1/3 the other times, because they are shorter in practice. The ratios between steps are erratic: incrementing from 5 to 6 makes little difference, while raising the value from 8 to 9 is more significant. The idea is to give more choice in the useful 1/20 to 1/3 second range. Attack rises to maximum output, which declines in the decay phase to the sustain level. Sustain therefore helps determine volume of a note. To increase a note's relative importance, increase its sustain level. Release, if and when it happens, drops from sustain to zero. The phases aren't quite linear.

To understand sustain levels, consider the three similar notes shown in Figure 13-9 with different sustain levels.

The notes diagrammed above are identical except for the sustain level. Note how the decay increases in importance and release decreases in importance as the sustain level drops.

With a sustain level of 0, these envelopes decay to 0 without needing release to be gated. POKE SID+4,16: POKE SID+4,17 plays notes of this type. If release is gated, the envelope shape depends on the release value; the note is lengthened, unchanged, or shortened, respectively, if the release value is greater than, equal to, or less than the decay value.

Figure 13-9. Varying Sustain Levels



Filter and Volume Registers

Cut-off and center frequency registers (offsets of 21 and 22). The filter cut-off and center registers (FC) use 11 bits, and the range is 0–2047. The register value is related to the true filter frequency as follows:

Filter frequency in Hz = 30 + 5.8 * Value in register
Register value FC = (Desired filter frequency − 30) * 0.17

The range of frequencies is therefore about 30–12,000 Hz.

Filter and resonance control register (offset of 23). Bits 7–4: The high nybble of this register sets the amount of resonance on a linear scale. Clear all bits to 0 for none, and set all to 1 for maximum.

Bit 3: External device filter. If external sound is routed through the 64, it will be filtered if this bit is 1.

Bit 2: Voice 3 filter. 0=off; 1=on.
Bit 1: Voice 2 filter. 0=off; 1=on.
Bit 0: Voice 1 filter. 0=off; 1=on.

Volume and filter selection register (offset of 24). Bit 7: Disconnect voice 3. When this bit is set, voice 3 is turned off, even if otherwise enabled. This allows voice 3 to generate both waves and envelopes, which can be read from the read-only registers for control purposes, without itself being audible.

Bit 6: High-pass filter. 0=off; 1=on.
Bit 5: Band-pass filter. 0=off; 1=on.
Bit 4: Low-pass filter. 0=off; 1=on.

Bits 3–0: Master volume. The low nybble of this register sets overall volume of the three voices.

Notes on filters. The SID chip has only one filter. When it's on, not all voices need pass through it. Bass accompaniment could be low-pass filtered to remove high harmonics and have resonance added. For example, if you put 17 into offset register 21, 241 (or 15*16+1) into register 23, and 31 into register 24 to low-pass filter voice 1 with cut-off starting at 128 Hz (an octave below middle C). High-passing imitation cymbal noises might need 21 in register 22, 1 in 23, and 79 in 24 to high-pass from 1000 Hz without resonance.

Bandpass filtering, notches, and resonance act on quite small regions of the frequency spectrum, so it's important to get the register values correct.

Read-Only Registers

Potentiometer readings (offsets of 25 and 26). See Chapter 16.

Waveform reading of voice 3 (offset of 27). Voice 3's frequency, and a waveform, must be set, but the gate needn't be on. For example, set register 14 (frequency, low byte) to 4 and register 18 (waveform) to 32. This sets up a very low frequency sawtooth wave, which takes about eight seconds to climb from 0 through 255. A sequence of PEEKs will show these changing values. With much higher frequencies, BASIC is too slow to PEEK consecutive values, but instead returns values taken from different waves, so the pattern appears to bear no relation to the waveshape. This is a standard result of wave sampling. It makes the effects of ML programs like the one below difficult to guess at.

For random numbers, particularly when using ML, POKE register 15 with 255, and register 18 with 129. This generates a fast changing noise output. Simply PEEK (or load in ML) the location to obtain a value.

Envelope reading of voice 3 (offset of 28). Voice 3's ADSR must be set, but a waveform needn't be selected. This is similar to the previous register; the major difference is that the envelope starts only when gated, and release starts only when the gate is set to 0.

Examples and Troubleshooting with the SID Chip

It is easier to understand the functions of the SID chip once you have heard the sound it generates. Program 13-2 controls individual bits in the SID chip's registers, so you can experiment with any combination. It displays the first 25 registers (the write-only ones) as eight bits, with register numbers and text to remind you what the registers do. The cursor keys move the colored cursor, and the space bar inverts the bit under the cursor and loads the new register value into the SID, so any change will be immediately audible. Since the space bar repeats, holding it down is an easy way to repeatedly gate and ungate a tone.

Program 13-2. Simple SIDmon

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
2 DATA "FREQ CONTROL LO{2 SPACES}{RVS}VOICE 1",FRE
  Q CONTROL HI                             :rem 63
3 DATA PULSE WIDTH BITS 7-Ø,PULSE WIDTH XXXX 11-8
                                          :rem 12Ø
4 DATA "N OL{G}N{G}NM TST RING SYN GATE"  :rem 116
```

```
5 DATA ATTACK{2 SPACES}DECAY, SUSTAIN RELEASE
                                          :rem 193
6 DATA "FREQ CONTROL LO{2 SPACES}{RVS}VOICE 2",FRE
  Q CONTROL HI                            :rem 68
7 DATA PULSE WIDTH BITS 7-Ø,PULSE WIDTH XXXX 11-8
                                          :rem 124
8 DATA "N OL⁅G⁆N⁅G⁆NM TST RING SYN GATE"  :rem 12Ø
9 DATA ATTACK{2 SPACES}DECAY, SUSTAIN RELEASE
                                          :rem 197
1Ø DATA "FREQ CONTROL LO{2 SPACES}{RVS}VOICE
   {SHIFT-SPACE}3",FREQ CONTROL HI        :rem 16
11 DATA PULSE WIDTH BITS 7-Ø,PULSE WIDTH XXXX 11-8
                                          :rem 167
12 DATA "N OL⁅G⁆N⁅G⁆NM TST RING SYN GATE" :rem 163
13 DATA ATTACK{2 SPACES}DECAY, SUSTAIN RELEASE
                                          :rem 24Ø
16 DATA "XXXXX FC2-FCØ{2 SPACES}{RVS}FILTER",FILTE
   R BITS FC1Ø-FC3                        :rem 237
18 DATA RESONANCE FX F3 F2 F1, V3OFF HP BP LP VOL3
   -VOLØ                                  :rem 239
2Ø DIM RV(24)                             :rem 155
25 FOR X=Ø TO 23:POKE 54272+X,Ø:NEXT      :rem 224
3Ø RV(1)=32:RV(4)=33:RV(6)=136:RV(24)=15  :rem 181
4Ø PRINT "{CLR}{WHT}";:FOR Y=Ø TO 24:FOR X=Ø TO 7
                                          :rem 184
5Ø PRINT CHR$(49 + ((RV(Y) AND (2↑(7-X)))=Ø));
                                          :rem 57
6Ø NEXT:POKE 54272+Y,RV(Y)                :rem 2Ø
7Ø READ M$:PRINT Y; M$;:IF Y<24 THEN PRINT:rem 123
8Ø NEXT Y                                   :rem Ø
1ØØ X=Ø:Y=Ø:OL=55296:GOTO 11ØØ            :rem 168
1ØØØ GET X$:IF X$="" THEN 1ØØØ            :rem 211
1Ø1Ø IF X$=" " THEN GOSUB 2ØØØ:GOTO 1ØØØ  :rem 243
1Ø2Ø IF X$="{RIGHT}" THEN X=X+1:IF X= 8 THEN X=Ø
                                          :rem 11Ø
1Ø3Ø IF X$="{LEFT}" THEN X=X-1:IF X=-1 THEN X=7
                                          :rem 3Ø
1Ø4Ø IF X$="{DOWN}" THEN Y=Y+1:IF Y=25 THEN Y=Ø
                                          :rem 151
1Ø5Ø IF X$="{UP}" THEN Y=Y-1:IF Y=-1 THEN Y=24
                                          :rem 71
11ØØ NW=55296+X+Y*4Ø:POKE OL,1:POKE NW,4:OL=NW:GOT
     O 1ØØØ                               :rem 254
2ØØØ SP=1Ø24+X+Y*4Ø:IF PEEK(SP)=48 GOTO 2Ø2Ø
                                          :rem 134
2Ø1Ø POKE SP,ASC("Ø"):RV(Y)=RV(Y) AND (NOT 2↑(7-X)
     ):GOTO 2Ø3Ø                          :rem 98
2Ø2Ø POKE SP,ASC("1"):RV(Y)=RV(Y) OR 2↑(7-X)
                                          :rem 184
2Ø3Ø POKE 54272+Y,RV(Y):RETURN            :rem 2Ø
```

Type in the program and run it. A few registers are preset by the program.

Sawtooth: Cursor over the gate bit of register 4 and tap the space bar a few times. A sawtooth tone sounds, with frequency about 500 Hz. Its envelope has zero attack and decay, but a sustain level of 8 (midvolume setting) and decay of 8, giving about a 1/3 second decay.

Triangle: Turn the triangle bit on and the sawtooth off. Gating again turns on the note, this time with the mellower triangle waveform. Altering bits in registers 1 and 2 changes the pitch. Also experiment with different attack and decay settings.

Pulse: Select the pulse-wave bit in register 4. You'll get no pulse-wave sound until registers 2 and/or 3 are POKEd.

Noise: Set the noise bit and experiment. A long attack, for example, helps simulate a steam train.

Ring modulation: Set the ring bit in register 4; set bit 2 in register 15, which sets voice 3's frequency. This has no effect; however, selecting triangle wave in register 4 gives ring modulation. Bell-like sounds are best obtained with 00001111 in register 5 and 00000000 in register 6, so the note has only a decay phase.

The array RV stores the current register values. The program initializes volume, as well as voice 1's frequency, sustain phase, waveform, and gate. This is the minimum required for the steady tone produced on RUN.

The cursor is controlled by POKEing color RAM, so the new X,Y position changes color, and the old reverts to white.

Experiments and examples. These straightforward examples are typical SID approximations of musical sounds.

Noise	Cymbals	A4 D11 S0 R0 High-pass filter
Pulse	Piano	A0 D9 S0 R0 Square wave (2048)
	Organ	A1 D2 S5 R1 1:4 Pulse (1024)
	Banjo	A0 D9 S0 R0 1:10 Pulse (410)
Sawtooth	Trumpet	A6 D0 S10 R1 Band-pass
	Accordion	A6 D7 S5 R3 High-pass filter
Triangle	Flute	A4 D2 S10 R5

Problems with the SID Chip

Often, the registers are simply not properly set. Volume (register 24) must be on. Each voice's frequency must be set; this takes either one or two POKEs. The waveform must be kept enabled while the gate, in the same register, is POKEd on or off. And the envelope shape must be set, with one or two POKEs. Pulse width must be set for pulse waves, which takes one or two POKEs.

Filtering requires registers 21 and/or 22 to set the filter frequency, and register 23 to select the voices to be filtered (and perhaps resonance). Volume register 24 also controls the type of filter, as well as on/off for voice 3.

Ring modulation must use a triangle waveform in the voice with ring bit set. And the frequency of the modulating voice must be set.

Special difficulties. At power-on, the SID chip is POKEd with zeros, but RUN/STOP–RESTORE only turns off the volume. So it can happen that a music program doesn't work even after RUN/STOP–RESTORE, typically because a gate bit is set to 1. To avoid this problem, sound programs should always start by POKEing zeros into all the SID chip's locations.

Low-frequency (bass) notes always sound weaker than high-frequency notes of the same amplitude. A bass-drum simulation on the SID chip, low-pass filtered at 70 Hz, is almost inaudible. If this is a problem, try increasing the relative sustain level of low frequencies.

After an envelope has decayed to zero, sound may still leak through. Turning off the gate bit is the solution. Also, long decays aren't reliable. The volume doesn't fall smoothly, but jumps noticeably in places.

The filter may not give repeatable results on other 64s. Early 64 programmers were warned by Commodore that filter characteristics varied considerably.

As noted, the noise waveform can't be mixed freely with other waveforms.

ML routines with read-only registers. The routine below is about the shortest practical ML routine using the read-only registers:

```
LDA  $D41B ;READ VOICE 3 OSCILLATOR
STA  $D41F ;STORE IN FREE LOCATION
LDA  $D41C ;READ VOICE 3 ENVELOPE
STA  $D41F ;STORE IN FREE LOCATION
JMP  $EA31 ;CONTINUE INTERRUPT
```

The BASIC loader, Program 13-3, POKEs this ML into RAM, then points the IRQ interrupt to it, so about every 1/60 second, voice 3's parameters are read and POKEd somewhere else. As it stands, the ML has no effect, since $D41F isn't used by the SID chip; but replacing the fifth byte with a number in the range 0–24 directs the oscillator into the register having that offset, and replacing the eleventh byte has the same effect with the envelope reading. For example, changing STA $D41F to STA $D400 means that voice 1's frequency control (low byte) now gets altered every 1/60 second to voice 3's output. Locations 56324 and 56325 control the interrupt rate (changing this rate also affects internal operations, making the cursor blink faster or slower, and so on).

Program 13-3. ML Read-Only Routine

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 FOR J=49152 TO 49166:READ X:POKE J,X:NEXT
                                         :rem 223
20 DATA 173,27,212,141,31,212,173,28,212,141,31,21
   2,76,49,234                           :rem 168
30 POKE 56333,127:POKE 788,0:POKE 789,192:POKE 563
   33,129                                :rem 214
40 SID=54272:FOR J=SID TO SID+24:POKE J,0:NEXT
                                         :rem 169
100 POKE SID+24,9+32                      :rem 38
110 POKE SID+15,16                       :rem 197
112 POKE SID+18,16                       :rem 202
118 POKE SID+1,32                        :rem 150
120 POKE SID+6,160 :REM VOICE 1 SUSTAIN  :rem 178
130 POKE SID+4,16+1: REM GATE V1 TRIANGLE :rem 13
200 POKE 49156,0: REM V3 OUTPUT TO R0    :rem 227
```

Voice 3's frequency determines oscillator register 27's output. Register 27 is location 54299 ($D41B). A very low frequency is inaudible, but when the register is controlling an audible frequency, it causes a slow, distinct pulsation. Higher frequencies are audible in themselves and also generate much faster changing output, giving frequency-modulated output (where the waveforms are rapidly condensed, then rarefied).

Voice 3's ADSR settings and the gate bit control register 28's output. Register 28 (location 54300, $D41C) therefore needs more work than 27, often more than is justified in view of the similarity between the outputs of 27 and 28.

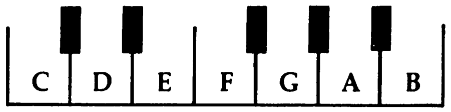
Program 13-3 shifts the triangular oscillator output of voice 3 into register 0, so the low byte of voice 1's frequency pulsates, giving vibrato. POKE 49156,1 uses register 1 and, of course, usually sounds more erratic. Register 2 or register 3, with pulse wave selected, causes fluctuations in readiness. POKE 49156,4 causes clicks and buzzes as the interrupt POKEs in assorted waveform and gate configurations. Register 22 (54294, $D416) alters the filter setting; provided voice 1 is filtered, you'll get a repetitive wah-wah effect. Register 24 (54296, $D418) alters master volume and a few other things, and gives vibrato or such effects as random changes in loudness when the noise voice is chosen. Whenever a register is partly used for different purposes, obviously LDA $D41B:AND #$0F:STA $D418 or similar constructions can help.

The envelope can modulate frequency or filtering, producing somewhat similar results: A note might start deep and faint, rise to a louder, more treble sound, then lose frequency as it decays. Voice 3 is easiest to modify in this way, since it can modulate itself. Run Program 13-3 and do these POKEs: POKE SID+4,0 (turns voice 1 off). POKE SID+19,176 (sets AD of voice 3). POKE 49162,15 (causes envelope reading to alter frequency of voice 3). Now type POKE SID+18,0: POKE SID+18,17 to gate a triangular wave, and you'll get a whistling sound. There are innumerable ways to combine voices, waveforms, ADSR shape, and duration.

Music Theory

Figure 13-10 shows a section of a piano keyboard, starting at C.

Figure 13-10. Piano Keyboard



It has 12 keys, tuned with frequencies in constant ratio to each other, so melodies can be played in any key, starting on any note. Adjacent notes are all separated by *semitones.* Octaves differ in frequency by a ratio of exactly 2, and similarly frequencies of semitones differ by the twelfth root of 2, which is 1.059463. Repeated division of a high frequency by this value generates regular subdivisions and provides an efficient way to generate tuned frequency settings for the SID chip. American standard pitch sets A at 440, while international standard pitch is 435. Whichever scale is used, any notes' frequencies can be calculated from any one note. (The subroutine starting at line 4000 in Program 13-5 evaluates 95 semitones from a single value.)

A conventional *scale* consists of 7 notes taken from these 12, plus the eighth (octave higher) note. So all scales have 12 semitones between the starting *keynote* and its octave. Of the 800 or so possible scales, in practice only a few are used. C major is the best known scale, starting with C and using only the white notes C, D, E, F, G, A, B, C.

The notes of this scale are separated by semitones like this: C *tone* D *tone* E *semitone* F *tone* G *tone* A *tone* B *semitone* C. All major scales have this pattern, and the arrangement of musical intervals in this type of scale has the same quality whatever the keynote.

Any major scale other than C uses some black keys. For example, G major is G, A, B, C, D, E, F-sharp, and G. Black keys are *sharps* moving up, and *flats* moving down, so C-sharp is the same key as D-flat. White keys are called *naturals.*

Natural minor scales have this pattern: A *tone* B *semitone* C *tone* D *tone* E *semitone* F *tone* G *tone* A. Note the total of 12 semitones.

A *chord* is a simultaneous combination of notes. Here is a look at two types of three-note chords. A *major* chord has four semitones between its first and second notes, and three between its second and third notes. The first, third, and fifth notes of a major scale identify this, so C major is C, E, and G. The fourth, sixth, and eighth notes of any major scale also produce a major chord (F, A, and C make F major), as do the fifth, seventh, and ninth notes (G, B, and D in the next octave make G major). Check the intervals between their notes to verify that they are major chords. These three chords include all the notes in the scale; consequently, any tune in a major scale may be harmonized by using one of these three chords. This is easy to implement on the 64.

A chord using the second, fourth, and sixth notes of the major scale is a *minor* chord, with three and four semitones separation between the notes. A minor chord is also produced if we build a chord on the third or the sixth notes of the scale. In the scale of C major, we get D minor, E minor, and A minor. Again, any note can be harmonized by one of these chords.

As we can see, it's possible to automate harmony to some extent, although the results are unexciting if prolonged. In general, minor chords are considered dull or sad and major chords are thought of as bright or triumphant.

It's also possible to generate tunes, by deciding on features like rhythms of measures or bars, the next note to be played, perhaps with probability depending on previous notes, and ascending or descending sequences, and then varying them randomly. Again, the results are generally unexciting.

Musical notation uses a *staff* of five lines, or two staves, with symbols to represent notes. Pitch is specified by their vertical position; the clef determines the actual pitch, and the treble clef is usually set with E on the lowest line, G on the next, and so on. Duration, and thus the rhythm of the music, is signaled by the type of symbol. Durations are relative, not absolute. A dot after a note symbol multiplies its duration by 1.5. Sharp or flat symbols precede the symbols for notes which are to be sharpened or flatted. Consult your dictionary for a list and explanation of basic musical notation.

Sound Demonstrations

Bach to the BASICs

The BASIC listing below, Program 13-4, plays a sequence of notes stored as three bytes of data each, two bytes controlling frequency and one, duration. It gates each note twice, allowing the full ADSR sequence, and consequently needs two delay loops, during sustain, then during release.

Program 13-4. Jesu Joy

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
1 DIM SH(81),SL(81)                          :rem 229
5 FOR Y=1 TO 81:READ SH(Y),SL(Y):NEXT        :rem 120
10 S=54272                                   :rem 245
20 FOR J=0 TO 24:POKE S+J,0:NEXT              :rem 15
21 POKE S+12,64:POKES+13,152:POKE S+19,64:POKE S+2
   0,32                                      :rem 217
30 POKE S+5,16*1+5                           :rem 147
40 POKE S+6,16*12+10                         :rem 243
50 X=1:POKE S+24,8                           :rem 220
60 READ HV,LV,DU                             :rem 179
70 IF HV<0 THEN FOR X=8 TO 0 STEP-1:POKE S+24,X:NE
   XT:END                                    :rem 171
80 POKE S,LV:POKE S+1,HV:POKE S+4,17:POKE S+8,SH(X
   ):POKE S+7,SL(X):POKE S+11,33              :rem 69
85 X=X+1                                     :rem 182
90 FOR D=1 TO 2*DU:NEXT                       :rem 19
93 POKE S+4,16                               :rem 224
95 FOR D=1 TO 2*DU:NEXT:GOTO 60              :rem 241
100 DATA 21,31,21,31,21,31,21,31,21,31,21,31,22,96
    ,22,96,22,96                             :rem 186
110 DATA 25,30,25,30,25,30,25,30,25,30,0,0,25,30,2
    5,30,25,30                                :rem 72
120 DATA 22,96,22,96,22,96,22,96,22,96,22,96,21,31
    ,21,31,21,31                             :rem 224
130 DATA 18,209,18,209,0,0,18,209,18,209,18,209,18
    ,209,18,209,18,209                        :rem 26
140 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
                                             :rem 251
150 DATA 21,31,21,31,21,31,21,31,21,31,21,31,22,96
    ,22,96,22,96                             :rem 191
160 DATA 25,30,25,30,25,30,25,30,25,30,25,30,21,31
    ,21,31,21,31                             :rem 174
170 DATA 18,209,21,31,22,96,18,209,18,209,18,209,0
    ,0,16,195,0,0                             :rem 14
180 DATA 16,195,16,195,16,195,16,195,16,195,16,195
    ,16,195,16,195,16,195                    :rem 213
200 DATA 21,31,32,16,235,32,18,209,32          :rem 7
210 DATA 21,31,32,25,30,32,22,96,32,22,96,32,28,49
    ,32,25,30,32                             :rem 207
220 DATA 25,30,32,33,135,32,31,165,32,33,135,32,25
    ,30,32,21,31,32                           :rem 81
```

```
230 DATA 16,195,32,18,209,32,21,31,32,22,96,32,25,
    30,32,28,49,32                              :rem 62
240 DATA 25,30,32,22,96,32,21,31,32,18,209,32,21,3
    1,32,16,195,32                              :rem 47
250 DATA 15,210,32,16,195,32,18,209,32      :rem 67
255 DATA 12,143,32,15,210,32,16,195,32,18,209,32,2
    5,30,32,23,181,32,25,30,32                 :rem 122
257 DATA 18,209,32,15,210,32,12,143,32,15,210,32,1
    8,209,32                                    :rem 14
260 DATA 22,96,32,21,31,32,18,209,32,21,31,32
                                               :rem 143
270 DATA 16,235,32,18,209,32                   :rem 94
280 DATA 21,31,32,25,30,32,22,96,32,22,96,32,28,49
    ,32,25,30,32                               :rem 214
290 DATA 25,30,32,33,135,32,31,165,32,33,135,32,25
    ,30,32,21,31,32                             :rem 88
300 DATA 16,195,32,18,209,32,21,31,32,14,36,32,25,
    30,32,22,96,32                              :rem 51
310 DATA 21,31,32,18,209,32,16,195,32,12,143,32,16
    ,195,32,15,210,32                          :rem 195
320 DATA 16,195,32,21,31,32,25,30,32,33,135,32,25,
    30,32,21,31,32                              :rem 32
330 DATA 16,195,96,-1,-1,-1                     :rem 27
```

A table of note values allows faster programming of short tunes. Weaknesses include the large amount of data needed per note, and the fact that all other processing stops. The program in the next section avoids these weaknesses.

Music Program

The music routine below, Program 13-5, is much more complex; it plays three-voice music from data written in approximately conventional notation, so music is fairly easy to write, compared with programs like the last, with their unnaturally formatted music data. The drawback is the time taken by BASIC to convert it into ML-readable bytes; however, once calculated, these can be stored for later use.

Program 13-5. Music Program

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 POKE 56,88:CLR:GOSUB 4500                  :rem 53
20 INPUT "IS MUSIC DATA IN RAM";A$:IF A$="Y" THEN
   {SPACE}1019                                :rem 134
30 GOSUB 5000:GOSUB 4000                      :rem 38
1000 FOR VN=0 TO 2:RA(VN)=0.50:DU=16:OC=4:P=22528+
     VN*6144                                  :rem 150
1005 READ VA$:IF VA$="Z" THEN POKE P,255:NEXT:GOTO
      1019                                     :rem 41
1007 IF LEFT$(VA$,1)="O" THEN READ RA(VN):GOTO 100
     5                                        :rem 235
1008 IF LEFT$(VA$,1)="W" THEN POKE P,253:P=P+1:REA
     D WV:POKEP,WV:P=P+1:GOTO 1005             :rem 16
```

```
1009 IF LEFT$(VA$,1)="P" THEN GOSUB 1400:GOTO 1005
                                           :rem 143
1010 IF LEFT$(VA$,1)="R" THEN VA$=MID$(VA$,2):GOSU
     B 2130:PI=0:GOTO 1014                 :rem 100
1012 GOSUB 2000:PI=PI+OC*12                :rem 249
1013 IF P>40950 THEN PRINT "TOO MUCH DATA":END
                                            :rem 46
1014 POKE P,PI:P=P+1:POKE P,DU*RA(VN):P=P+1:rem 32
1016 POKE P,DU-INT(DU*RA(VN)):P=P+1:GOTO 1005
                                           :rem 108
1019 INPUT "TEMPO (1-255)";TM:POKE 56325,TM
                                           :rem 134
1020 POKE 56334,0:GOSUB 1500               :rem 211
1022 POKE 165,0:POKE 167,0:POKE 169,0      :rem 180
1023 POKE 166,88:POKE 168,112:POKE 170,136:rem 189
1024 POKE 788,208:POKE 789,192:POKE 254,0 :rem 156
1025 FOR J=49355 TO 49359 STEP 2:POKE J,32:NEXT
                                            :rem 28
1026 FOR J=49349 TO 49354:POKE J,0:NEXT    :rem 120
1028 FOR J=0 TO 2:POKE 49344+J*2,J*7:NEXT :rem 208
1038 POKE 56334,1                           :rem 93
1039 IF PEEK(254)=7 THEN 1020              :rem 211
1040 GOTO 1039                             :rem 203
1400 POKE P,254:P=P+1:READ RG:POKE P,RG    :rem 235
1410 P=P+1:READ PA:POKE P,PA:P=P+1:RETURN :rem 233
1500 SI=54272:PRINT "INITIALIZING SID"      :rem 25
1505 FOR J=SI TO SI+28:POKE J,0: NEXT       :rem 49
1510 POKE SI+5,17:POKE SI+12,17:POKE SI+19,17
                                           :rem 219
1520 POKE SI+6,177:POKE SI+13,177:POKE SI+20,177
                                           :rem 123
1530 POKE SI+24,4                          :rem 133
1550 RETURN                                :rem 171
2000 IF LEFT$(VA$,1)="+" THEN OC=OC+1:VA$=MID$(VA$
     ,2):GOTO 2040                         :rem 116
2005 IF LEFT$(VA$,1)="-" THEN OC=OC-1:VA$=MID$(VA$
     ,2):GOTO 2040                         :rem 125
2010 IF LEFT$(VA$,1)<"0" OR LEFT$(VA$,1)>"9" THEN
     {SPACE}2040                           :rem 195
2030 OC=ASC(VA$)-ASC("0"):VA$=MID$(VA$,2) :rem 218
2040 PI=ASC(VA$)-ASC("A")                   :rem 89
2060 PI=PC(PI):IF PI<=0 THEN PI=PI+12       :rem 82
2070 VA$=MID$(VA$,2):IF VA$="" THEN RETURN :rem 61
2090 IF LEFT$(VA$,1)="#" THEN PI=PI+1: VA$=MID$(VA
     $,2)                                   :rem 74
2110 IF LEFT$(VA$,1)="F" THEN PI=PI-1: VA$=MID$(VA
     $,2)                                  :rem 104
2130 IF VA$="" THEN RETURN                 :rem 160
2140 IF LEFT$(VA$,1)>"9" OR LEFT$(VA$,1)<"0" THEN
     {SPACE}PRINT "?":END                   :rem 34
2150 DU=VAL(VA$):RETURN                    :rem 167
```

```
4000 PRINT "INITIALIZING FREQUENCY TABLE"  :rem 58
4010 DIM FQ(95):FQ(95)=64814              :rem 201
4040 FOR J=94 TO 84 STEP -1:FQ(J)=FQ(J+1)/2↑(1/12)
     :NEXT                                :rem 207
4050 FOR J=6 TO 0 STEP -1:FOR K=1 TO 12   :rem 186
4060 P1=12*J+K-1:FQ(P1)=FQ(P1+12)/2:NEXT:NEXT
                                          :rem 93
4070 FOR J=1 TO 95:POKE 49151+J,FQ(J)-256*INT(FQ(J
     )/256)                               :rem 233
4080 POKE 49247+J,FQ(J)/256:NEXT:RETURN    :rem 55
4500 PRINT "LOADING ML"                    :rem 49
4502 FOR J=49360 TO 49537:READ X:POKE J,X:NEXT:RET
     URN                                  :rem 102
4504 DATA 162,0,32,226,192,162,2,32,226,192,162,4
                                           :rem 98
4512 DATA 32,226,192,76,49,234,189,197,192,240,18
                                          :rem 138
4524 DATA 222,197,192,240,1,96,188,192,192,189
                                          :rem 249
4530 DATA 203,192,41,254,153,4,212,96,189,198
                                          :rem 184
4542 DATA 192,240,6,222,198,192,240,1,96,161,165
                                           :rem 71
4554 DATA 240,61,201,255,208,18,188,192,192,169
                                           :rem 32
4566 DATA 0,153,4,212,138,208,2,232,138,5,254
                                          :rem 164
4572 DATA 133,254,96,201,254,240,55,201,253,240
                                           :rem 10
4584 DATA 71,168,138,72,189,192,192,170,185,255
                                           :rem 51
4596 DATA 191,157,0,212,185,95,192,157,1,212
                                          :rem 134
4608 DATA 138,168,104,170,189,203,192,9,1,153,4
                                           :rem 22
4614 DATA 212,32,123,193,161,165,157,197  :rem 188
4626 DATA 192,32,123,193,161,165,157,198,192,32
                                           :rem 32
4632 DATA 123,193,96,32,123,193,161,165,168,32
                                          :rem 231
4644 DATA 123,193,161,165,153,0,212,32,123,193
                                          :rem 213
4656 DATA 76,4,193,32,123,193,161,165,157,203,192
                                          :rem 127
4662 DATA 32,123,193,76,4,193,246,165,208 :rem 244
4674 DATA 2,246,166,96                     :rem 77
5000 PRINT "INITIALIZING PITCH ARRAY PTRS" :rem 97
5010 FOR J=0 TO 6:READ PC(J):NEXT:RETURN   :rem 84
5020 DATA -2,0,1,3,5,6,8                  :rem 127
10000 DATA W,64,P,3,3,O,0.9               :rem 112
10001 DATA R64,R,4BF8,5D,F16,4BF8,5EF,G20 :rem 197
```

```
10005 DATA BF4,A,G,F8,EF,D,C4,D,EF,D,C,4BF :rem 14
10010 DATA 5C8,C,F20,EF4,D,EF,F,EF,D,C      :rem 25
10015 DATA D8,D,G24,C8,D4,EF,F16           :rem 177
10020 DATA 4BF8,5C4,D,EF12,4G4,AF,5C,4BF,AF,G,F
                                            :rem 21
10025 DATA 5F8,D,EF20,G4,F,EF,D,C,4BF,5C   :rem 137
10030 DATA DF,4AF,G,F,EF,G,AF,BF,5C,4BF,AF,G,F,BF,
      5C,D                                 :rem 225
10035 DATA EF,C,-BF,AF,G,+D,EF,F,-BF,+EF,F,G,C,F,-
      BF,AF                                :rem 245
10040 DATA G16,R8,G,F4,AF,+D,C,D,-AF,F,AF :rem 190
10045 DATA G,BF,+EF,-BF,G,BF,EF,G,D8,F,BF16:rem 88
10050 DATA EF8,G,BF20,+C4,E,D,E,-BF,G,BF  :rem 132
10055 DATA A,+C,F,C,-A,+C,-F,A,E8,G,+C16   :rem 46
10060 DATA -F8,A,+C20,D4,F#,E,F#,C,-A,+C    :rem 7
10065 DATA -BF,+D,G,D,-BF,+D,-G,BF,+C8,-F,+F,EF4,D
                                           :rem 130
10070 DATA C,-F,G,A,BF,+C,D,EF,F8,D,-BF16 :rem 171
10075 DATA +F8,D,-AF16,+F8,D,-G,+F        :rem 249
10080 DATA EF,C,-G,+C4,D,EF8,C,-F,+EF     :rem 202
10085 DATA D8,-BF,F,BF4,+C,DF8,-BF,E,+DF  :rem 138
10090 DATA C4,-BF,A,G,F,EF,D,C,D8,BF,C,A  :rem 140
10100 DATA BF16,R,R,R                     :rem 117
19999 DATA Z                              :rem 137
20000 DATA W,64,P,10,5,O,0.8              :rem 160
20001 DATA 4EF8,G,BF16,EF8,AF,5C20          :rem 58
20005 DATA EF4,D,C,4BF8,AF,G,F4,G,AF,G,F,EF:rem 93
20010 DATA F,EF,D,C,3BF,4D,EF,F,G,F,EF,D,C,F,G,A
                                           :rem 106
20015 DATA BF12,5C4,4A12,BF4,BF8,F,BF20     :rem 53
20020 DATA AF4,G,AF,BF,AF,G,F,G8,G,5C20     :rem 95
20025 DATA 4BF4,A,BF,5C,4BF,AF,G,AF,G,AF,5C,4BF,AF
      ,G,F                                 :rem 211
20030 DATA G,F,G,BF,AF,G,F,EF,F,EF,F,AF,G,F,EF,D
                                           :rem 152
20035 DATA EF,D,C,3BF,4BF8,AF,G16,F       :rem 132
20040 DATA EF8,G,BF16,EF8,AF,5C20            :rem 9
20045 DATA EF4,D,C,-BF8,AF,G,+EF,-F,+D      :rem 14
20050 DATA -EF16,R8,EF8,D,F,BF16          :rem 210
20055 DATA EF8,G,BF16,F4,AF,+D,C,D,-AF,F,AF:rem 63
20060 DATA G,BF,+EF,-BF,G,BF,EF,D,E8,G,+C16:rem 59
20065 DATA -F8,A,+C16,-G4,BF,+E,D,E,-BF,G,BF
                                            :rem 61
20070 DATA A,+C,F,C,-A,+C,-F,E,F#8,A,+D16   :rem 79
20075 DATA -G8,BF,+EF20,D4,C,D,EF,D,C,-BF :rem 182
20080 DATA A16,R4,A,BF,+C,-BF,+D,F,EF,F,D,-BF,+D
                                            :rem 44
20085 DATA -AF,+D,F,EF,F,D,-AF,+D,-G,+D,F,EF,F,D,-
      G,+D                                 :rem 111
20090 DATA -G,+C,EF,D,EF,C,-G,+C,-F,+C,EF,D,EF,C,-
      F,+C                                 :rem 105
```

```
20095 DATA -F,BF,+D,C,D,-BF,F,BF,E,BF,+DF,C,DF,-BF
      ,E,BF                                 :rem 243
20100 DATA F,G,A,BF,+C20,-F4,D,F,G,EF,C,EF:rem 232
20105 DATA D16,R,R,R                         :rem 55
29999 DATAZ                                 :rem 138
30000 DATA W,64,P,17,7,O,0.98               :rem 227
30001 DATA 3EF16,R4,G,F,G,C16,R4,EF,D,EF    :rem 163
30005 DATA -AF16,BF,EF8,+EF,D,C             :rem 145
30010 DATA D,F,D,-BF,+EF,G,EF,C             :rem 144
30015 DATA D,EF,F,-F,BF16,R8,+D             :rem 139
30020 DATA F16,R8,AF,C16,R8,EF              :rem 104
30025 DATA G16,R8,BF,C,EF,AF16              :rem 108
30030 DATA -BF8,+D,G16,-AF8,+C,F,AF16       :rem 176
30035 DATA -AF8,G,+C,-BF,+BF16,AF8           :rem 32
30040 DATA G,BF,G,EF,AF,+C,-AF,F            :rem 213
30045 DATA G,AF,BF,D,EF,C,-AF,BF            :rem 238
30050 DATA EF,F,G,EF,BF16,R8,+D             :rem 165
30055 DATA EF16,R8,-G,BF16,R8,+D            :rem 203
30060 DATA EF16,R8,G,C16,R8,E                :rem 43
30065 DATA F16,R8,-A,+C16,R8,E               :rem 61
30070 DATA F16,R8,A,D16,R8,F#                 :rem 6
30075 DATA G16,R8,+C,-BF,A,BF,EF            :rem 208
30080 DATA F,-F,+F,EF,D16,R8,+D             :rem 118
30085 DATA C16,R8,-C,-B16,R8,+B             :rem 103
30090 DATA +C16,R8,2BF,A16,R8,+A            :rem 169
30095 DATA BF16,R8,AF,G16,R8,-G             :rem 163
30100 DATA A,+C,-A,F,BF,+D,EF,F             :rem 109
30105 DATA -BF16,R,R,R                      :rem 169
39999 DATA Z                                :rem 139
```

To use Program 13-5, after entering the program and saving, type RUN and press RETURN. Answer N to the question IS MUSIC DATA IN RAM? and wait for the frequency table to be set up (about a minute and a half). To begin with, enter a tempo of 160 to hear the music. Press RUN/STOP to leave the routine, and enter RUN to restart, answering Y to IS MUSIC DATA IN RAM? this time. Enter a new tempo to hear the music again.

This program uses an interrupt. The music can play during BASIC program running without much slowing effect. The ML sets frequency values, gates, and so on. Different rhythms between voices are handled correctly, which BASIC cannot do except at a very slow tempo.

Music encoding system. Notes can be specified by data items of this form:

[Octave 1–7 or + or −] Note A–G [# or F] [duration 1–255]

No spaces are allowed between parameters. The items in square brackets are optional: When omitted, current values are used, or defaults, where no value has yet been input. Octave defaults to 4, the octave starting on middle C; 1 to 7 sets the octave; + or − can indicate a change to the next octave up or down. Note letters are, of course, A–G, with optional # (sharp) or F (flat) indicators. Duration is in terms of interrupts. A smaller duration means faster tempo; the default is 16. Some examples of valid data are:

C	C in current octave with current duration.
3D	Change octave to 3. Play D.
+DF8	Change to next octave up. Play D-flat, duration 8.
4F#24	Change to octave 4, duration 24. Play F-sharp.

And, for example, DATA C,D,E,F,G,A,B,+C,−B,A,G,F,E,D,C defines an ascending major scale followed by a descending major scale.

R defines a rest. If no duration value is included, the current value is used. R8, R are examples.

O resets the on/off ratio, the fraction of the note for which the gate is held on. It must be followed by a number from 0 to 1.0. DATA O,0.1 is an example. The default is 0.5. Don't confuse the letter O with the number 0.

W allows POKEs to the control register; W,128 selects noise and W,20 triangular wave with ring mod bit on. The default waveform is sawtooth.

P allows POKEs to any SID register, which gives freedom to control filters, pulse widths, and other low-level SID features. Two parameters are necessary: for example, DATA P,24,6 sets SID register 24 to 6, POKEing volume to 6.

Z marks the end of a voice's data. So DATA C16,D,E,F,Z,B8,A,G,F,E,D,C,D,Z,Z defines two music parts, with the second voice moving at twice the rate of the first, and the third silent.

The program up to, but not including, line 10000 is the processing part; lines 10000 on hold the data, which in our example is part of the first Bach Organ Sonata, in E-flat major. Its data illustrates all these notations. W,64 selects pulse wave; P,3,3 initializes the pulse-width high register; O,0.9 sets the on/off ratio to 0.9, appropriately for an organ sound. The three voices have intentional slight differences in quality: each has a different pulse width and on/off ratio. R64, a rest of duration 64, corresponds to a complete bar or measure. The quarter note has been defined as 16 clock units in this piece, and there are four of these notes per bar. Most of the DATA statements contain exactly one bar of information.

The program can be used to experiment with rapid, highly controlled modifications to the registers as a sophisticated sound-effects generator. For example, edit the first data line to appear as follows:

```
10000 DATA O,1,W,64,C,P,3,9,C,P,3,1,C,P,3,4,C,P,3,8,Z,Z,Z
```

This sets the on/off ratio to 1 so that the gate is permanently held on, selects the pulse wave, and plays four C notes, changing the pulse width between each note. A note with a continuously changing timbre will be heard.

How the Music Program Works

ML data for the voices is stored at $5800–$6FFF, $7000–$87FF, and $8800–$9FFF. It's put there by BASIC, which traps most format errors. Locations $C000–$C0BF have frequency data for 96 notes. Line 4010 determines their value and allows tuning. The ML interrupt routine starts at $C0D0. All notes occupy three bytes: a pitch number, 1–95, allowing eight octaves of 12 semitones, as well as an *on* value and an *off* value, representing the number of interrupts before gating. Each voice can hold about 2000 notes maximum. Rests are stored with pitch 0. A code of 253 is followed by a waveform byte; 254 is followed by a SID register and new value; and 255 is the end-of-data marker. The BASIC lines from 1000 on convert music data into this format.

The ML subroutine at $C0E2 processes this data. It's called three times each interrupt, once for each voice, and uses indirect addresses ($A5), ($A7), and ($A9). On and off values for the three notes are stored at $C0C5–$C0CA, to time down at each interrupt. Location 254 is set to 7 when the end of data has been reached in all three voices.

On RUN, if data hasn't yet been read, parsed, and POKEd into RAM, lines 1000 on and 2000 on do this. The tempo is POKEd into the CIA timer controlling interrupts: Use 160–180 for the Bach piece. (Any tempo can be obtained by choosing an appropriate combination of tempo value and note duration values in the music data.) Line 1025 sets the default waveform. The subroutine at 1500–1550 initializes the envelope of each voice. The Bach piece has A, D, and R set at 1, and sustain at 11. Note that ADSR values should not be zero with this program.

Organ Keyboard

Program 13-6 below turns the 64 into a simple two-voice organ. It also handles three notes as far as the keyboard permits—some groups of three keys cannot be decoded unambiguously by the 64. The keyboard design makes it impossible. The QWERTY row sounds white notes from G to the E, almost two octaves above. Keys 1, 2, 3, 5, 6, and so on, play F-sharp, G-sharp, A-sharp, and so on. Keys R, Y, and I together play C major; T, U, and O give D minor; T, *, and O give D major.

Program 13-6. Organ Keyboard

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
5 DATA 120,160,0,132,252,132,253,132,254,162,3,142
  ,74,192,169,254,141                         :rem 52
6 DATA 0,220,162,8,72,173,1,220,205,1,220,208,248,
  74,176,21,72,142,75                         :rem 50
7 DATA 192,174,74,192,185,129,235,149,251,206,74,1
  92,240,20,104,174,75                        :rem 142
8 DATA 192,200,192,65,176,12,202,208,224,56,104,42
  ,141,0,220,208,205                          :rem 255
9 DATA 104,104,88,96                          :rem 224
20 FOR J=49152 TO 49225:READ X:POKE J,X:NEXT
                                              :rem 220
40 GOSUB 3000                                 :rem 167
45 PW=400:GOSUB 2000                           :rem 93
50 GOSUB 4000                                 :rem 169
60 GOSUB 5000                                 :rem 171
70 POKE 53281,8                               :rem 253
74 BL$="{3 SPACES}{BLK}{RVS}{RIGHT}1{RIGHT}2
   {RIGHT}3{3 RIGHT}5{RIGHT}6{3 RIGHT}8{RIGHT}9
   {RIGHT}0{3 RIGHT}-{RIGHT}£"               :rem 59
75 PRINT "{CLR}" BL$:PRINT BL$                :rem 251
79 WH$="{4 SPACES}{WHT}{RVS}{RIGHT}Q{RIGHT}W
   {RIGHT}E{RIGHT}R{RIGHT}T{RIGHT}Y{RIGHT}U{RIGHT}
   I{RIGHT}O{RIGHT}P{RIGHT}@{RIGHT}*{RIGHT}↑"
                                               :rem 53
```

```
80 PRINT "{UP}" WH$:PRINT WH$:PRINT WH$     :rem 161
90 L1=252:L2=253:L3=254:SR=49152            :rem 253
95 WN=33:WF=32                               :rem 47
100 SYS SR                                   :rem 53
110 N1=MA%(PEEK(L1)):N2=MA%(PEEK(L2)):N3=MA%(PEEK(
    L3))                                     :rem 37
115 IF N1<>0 THEN POKE SI+ 4,WN:POKE SI,LQ%(N1):PO
    KE SI+1,HQ%(N1)                          :rem 87
116 IF N2<>0 THEN POKE SI+11,WN:POKE SI+7 ,LQ%(N2)
    :POKE SI+8,HQ%(N2)                       :rem 242
117 IF N3<>0 THEN POKE SI+18,WN:POKE SI+14,LQ%(N3)
    :POKE SI+15,HQ%(N3)                      :rem 89
131 IF PEEK(197)=4 THEN WN=33:WF=32         :rem 156
132 IF PEEK(197)=5 THEN WN=17:WF=16         :rem 162
133 IF PEEK(653)=1 THEN WN=65:WF=64:GOSUB 2000
                                             :rem 30
150 IF PEEK(L1)=0 THEN POKE SI+4,WF         :rem 167
152 IF PEEK(L2)=0 THEN POKE SI+11,WF        :rem 216
153 IF PEEK(L3)=0 THEN POKE SI+18,WF        :rem 225
160 GOTO100                                  :rem 97
2000 REM **** ALTER PULSE WIDTH ****        :rem 119
2005 PW=(PW+50) AND 4095:PL=PW AND 255:PH=PW/256
                                            :rem 131
2010 FOR I=2 TO 16 STEP 7:POKE SI+I,PL:POKE SI+I+1
     ,PH:NEXT                                :rem 50
2020 RETURN                                 :rem 164
3000 REM                                    :rem 167
3003 SI=54272                               :rem 163
3005 FOR J=SI TO SI+24:POKE J,0:NEXT         :rem 42
3010 POKE SI+ 5,16*5+11                     :rem 110
3011 POKE SI+12,16*5+11                     :rem 157
3012 POKE SI+19,16*5+11                     :rem 165
3020 POKE SI+6,16*15+12                     :rem 162
3021 POKE SI+13,16*15+12                    :rem 209
3022 POKE SI+20,16*15+12                    :rem 208
3030 POKE SI+24,3                           :rem 129
3040 POKE SI+ 4,0                            :rem 77
3041 POKE SI+11,0                           :rem 124
3042 POKE SI+18,0                           :rem 132
3050 RETURN                                 :rem 168
4000 REM                                    :rem 168
4005 DIM FQ(95),LQ%(95),HQ%(95)              :rem 79
4010 FQ(95)=64814                            :rem 95
4020 FOR J=94 TO 84 STEP -1:FQ(J)=FQ(J+1)/(2↑(1/12
     )):NEXT                                 :rem 30
4030 FOR J=6 TO 0 STEP -1:FOR K=1 TO 12     :rem 184
4040 P1=J*12+K-1:FQ(P1)=FQ(P1+12)/2:NEXT:NEXT
                                             :rem 91
4045 FOR P1=1 TO 95                         :rem 180
4050 LQ%(P1)=FQ(P1)-256*INT(FQ(P1)/256):HQ%(P1)=FQ
     (P1)/256                               :rem 164
```

```
4060 NEXT:RETURN                                  :rem 35
5000 REM                                         :rem 169
5005 DIM MA%(255)                                 :rem 68
5010 FOR J= 0 TO 22:READ V$,V:MA%(ASC(V$))=V:NEXT
                                                 :rem 169
5020 RETURN                                      :rem 167
5500 DATA 1,42,Q,43,2,44,W,45,3,46,E,47,R,48,5,49,
     T,50,6,51,Y,52,U,53,8,54,I,55                 :rem 4
5510 DATA 9,56,O,57,0,58,P,59,@,60,-,61,*,62,£,63
     ,↑,64                                       :rem 228
```

Running the program sets the waveform to sawtooth. Pressing f3 selects the triangular wave, and SHIFT selects the pulse waveform and, if held, alters its duty cycle; it can operate when a note or chord is held, making audible the resulting change in timbre. Pressing f1 reselects sawtooth.

An ML routine is essential to determine which keys are pressed, as the keyscan routine returns just one value. This program's ML puts ASCII values in 254, or 254 and 253, or 254, 253, and 252, depending on whether one, two, or three keypresses were detected. BASIC PEEKs these locations and converts them to pitches.

Programmable Rhythm Box

Program 13-7 has six voices available, with the waveform and envelope defined by DATA in lines 3040–3090. Each voice definition has eight bytes, one for each of the seven SID registers plus an extra one. As it stands, this program uses no filtering, ring, or sync controls, partly because six voices would be harder to generate. But these and other features can be added without much difficulty. Since the SID chip has only three voices, the six instruments cannot be played together: 1 and 2 are played by SID voice 1; 3 and 4 by voice 2; and 5 and 6 by voice 3, so voices 1 and 2 can't sound simultaneously. However, they can alternate, so the rhythm might be 1, 1, 2, 2, 1, 1, or whatever.

Program 13-7. Rhythm Box

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
1 GOSUB 2000:NB=32:INPUT "NUMBER OF BEATS
  {3 SPACES}(1-32: DEFAULT=32)";NB          :rem 122
2 TM=5:INPUT "TEMPO{13 SPACES}(1-50: DEFAULT= 5)";
  TM                                        :rem 121
3 A$="N":INPUT "CLEAR RHYTHMS?{3 SPACES}(Y/N: DEFA
  ULT IS N)";A$                              :rem 58
4 IF A$="Y" THEN GOSUB 6000                 :rem 119
5 A$="N":INPUT "LOAD RHYTHMS{5 SPACES}(Y/N: DEFAUL
  T IS N)";A$                               :rem 182
6 IF A$="Y" THEN GOSUB 7000                 :rem 122
8 GR$=" B+++B+++B+++B+++B+++B+++B+++B+++"  :rem 142
9 PS$=" ................................":rem 111
10 PRINT "{CLR}{2 DOWN}" LEFT$(GR$,NB+1)    :rem 92
20 FOR J=1 TO 3:PRINT LEFT$(PS$,NB+1):NEXT:rem 150
30 PRINT "{HOME}";:POKE 53280,6:POKE 650,128
                                             :rem 26
```

```
50 GOSUB 3000                                    :rem 168
60 GOSUB 5000                                    :rem 171
70 N$="000000"                                   :rem 122
100 PRINT " ": PRINT "{UP}";                      :rem 57
110 TI$=N$:POKE 56334,17                           :rem 4
120 PRINT " V{LEFT}";:POKE 252,BT:SYS 49200:rem 14
130 IF TI<TM GOTO 130                            :rem 106
140 BT=BT+1:IF BT=NB THEN BT=0:GOTO 100          :rem 188
150 GOTO 110                                      :rem 97
1999 REM                                         :rem 192
2000 SI=54272                                    :rem 159
2010 POKE SI+4,8:POKE SI+11,8:POKE SI+18,8 :rem 68
2020 FORI=SI TO SI+28:POKE I,0                   :rem 175
2030 POKE SI+24,8                                :rem 133
2040 RETURN                                      :rem 166
2999 REM                                         :rem 193
3000 REM                                         :rem 167
3010 REM                                         :rem 168
3030 REM 1 FREQ LOW                              :rem 251
3031 REM 2 FREQ HI                               :rem 156
3032 REM 3 PW{3 SPACES}LO                         :rem 33
3033 REM 4 PW{3 SPACES}HI                         :rem 25
3034 REM 5 WAVEFORM + GATE                       :rem 150
3035 REM 6 ATTACK +{3 SPACES}DECAY                :rem 46
3036 REM 7 SUSTAIN +{2 SPACES}RELEASE             :rem 58
3037 REM{2 SPACES}PLUS ONE 'DUMMY' BYTE TO GIVE AN
      8-BYTE BLOCK                                :rem 70
3039 DATA "INSTRS"                                :rem 16
3040 DATA 000,010,000,000,017,024,008,0           :rem 76
3050 DATA 000,015,000,000,017,023,007,0           :rem 80
3060 DATA 000,015,000,000,129,006,000,0           :rem 79
3070 DATA 075,001,000,008,065,025,008,0          :rem 103
3080 DATA 000,226,000,000,129,025,008,0           :rem 94
3090 DATA 255,253,000,000,129,040,008,0          :rem 104
3100 RESTORE:FOR J=1 TO 1E9:READ X$:IF X$<>"INSTRS
     " THEN NEXT                                  :rem 57
3110 FOR J=0 TO 6*8-1:READ X:POKE 49152+J,X:NEXT
                                                 :rem 138
3120 REM                                         :rem 170
3130 RETURN                                      :rem 167
4998 REM                                         :rem 194
4999 DATA"ML"                                    :rem 214
5000 DATA 165,252,141,245,192,9,32,141,246,192,73,
     96,141,247,192,32,228                       :rem 182
5001 DATA 255,201,0,240,54,141,243,192,162,3,221,2
     39,192,208,17,189,244                       :rem 162
5002 DATA 192,170,169,63,157,92,193,169,46,32,214,
     192,76,124,192,202,208                      :rem 243
5003 DATA 231,173,243,192,56,233,49,201,6,176,14,4
     2,42,42,72,42,41,96,5                       :rem 166
```

```
5004 DATA 252,170,104,157,92,193,174,245,192,189,9
     2,193,201,63,240,9,168                 :rem 245
5005 DATA 32,209,192,162,0,32,182,192,174,246,192,
     189,92,193,201,63,240                  :rem 181
5006 DATA 9,168,32,209,192,162,7,32,182,192,174,24
     7,192,189,92,193,201                   :rem 151
5007 DATA 63,240,9,168,32,209,192,162,14,32,182,19
     2,96,185,4,192,41,254                  :rem 188
5008 DATA 157,4,212,169,7,141,244,192,185,0,192,15
     7,0,212,200,232,206,244                  :rem 3
5009 DATA 192,208,243,96,106,106,106,105,49,72,138
     ,201,32,144,9,201,64                   :rem 122
5010 DATA 144,3,24,105,8,105,8,170,104,157,121,4,1
     69,1,157,121,216,96,133                :rem 249
5011 DATA 134,135                            :rem 62
5015 RESTORE: FOR J=1 TO 1E9: READ X$: IF X$<>"ML"
      THEN NEXT                             :rem 246
5020 FOR J=49200 TO 49394: READ X: POKE J,X: NEXT:
     RETURN                                  :rem 92
5999 REM                                    :rem 196
6000 FOR J=49500 TO 49500+95:POKE J,63:NEXT:rem 53
6010 REM                                    :rem 171
6020 POKE 49500+32,16                        :rem 28
6021 POKE 49500+40,16                        :rem 28
6022 POKE 49500+48,16                        :rem 37
6023 POKE 49500+56,16                        :rem 37
6040 RETURN                                 :rem 170
6998 REM ---------------------------        :rem 176
6999 DATA "RHYTHMS"                         :rem 110
7000 DATA 63,63,63,63,63,63,63,63            :rem 93
7002 DATA 63,63,63,63,63,63,63,63            :rem 95
7004 DATA 08,63,00,63,63,63,00,63            :rem 78
7006 DATA 08,63,08,63,08,63,08,63            :rem 95
7007 REM                                    :rem 178
7010 DATA 16,63,63,63,63,63,63,63            :rem 92
7012 DATA 16,63,63,63,63,63,63,63            :rem 94
7014 DATA 16,63,63,63,63,63,63,63            :rem 96
7016 DATA 16,63,16,63,63,63,63,63            :rem 96
7017 REM                                    :rem 179
7020 DATA 40,63,63,63,40,63,40,63            :rem 80
7022 DATA 40,63,63,63,40,63,63,63            :rem 87
7024 DATA 32,63,63,63,32,63,32,40            :rem 82
7026 DATA 40,63,40,32,32,40,32,63            :rem 74
7030 RESTORE:FOR J=1 TO 1E9:READ X$:IF X$<>"RHYTHM
     S" THEN NEXT                           :rem 139
7040 FOR J=49500 TO 49500+95:READ X:POKE J,X:NEXT:
     RETURN                                 :rem 241
```

The program displays a grid of 3 rows and as many as 32 columns; the rows represent the SID voices and the columns stand for beats in the bar. An X character moves horizontally over the grid showing which beat is being played.

The program first asks the user to specify the number of beats to be used, the tempo, whether the rhythms currently in RAM should be cleared, and whether the rhythms defined in lines 7000–7026 should be loaded into memory. Press RETURN in response to the first three prompts to select the defaults, and Y to the last prompt to load the built-in rhythms.

Typing 1–6 plays the designated instrument at that point in the bar and adds it to the display and to the rhythm stored in RAM so that it will be heard at that point in the bar from now on. Pressing f1, f3, or f5 as the program plays will delete voices 1 and 2, 3 and 4, or 5 and 6, respectively, from the display and from memory at the position in the bar where they are typed. Thus, instruments can be added to and removed from the rhythm. If you press RUN/STOP, then run the program again, you can keep the new values by being careful not to clear them or overwrite them.

DATA in lines 7000–7026 holds the start-up rhythms. There are three groups of DATA statements, each of 32 values, one for each beat of the bar. A value of 63 indicates that no note is present at that beat. Values of 0 and 8 in the first group mean that instrument 1 or 2 has a note on that beat; values of 16 and 24 in the second group mean that instrument 3 or 4 has a note on that beat; and values of 32 and 40 in the third group mean that instrument 5 or 6 has a note on that beat. The data may be edited to provide accurately programmed rhythms, and rhythms can be edited as the system plays.

Music Programming Aids

A number of monitors and synthesizer programs exist, some commercial, others in the public domain (free). Typically, these allow parameters to be altered relatively easily; for example, F may alter frequency, P pulse width. Ideally, they should include gate timing, true frequencies in Hz, smooth gliding through values, indications of connections between ring, sync, and filter settings or registers, and facilities to use the read-only registers. They should be able to report how sounds they're producing are made. Otherwise, you can be put in the frustrating position of being unable to reconstruct some attractive sound. Some indication of waveform, or even a waveform analyzer/synthesizer, would be useful.

BASIC extensions are difficult to write for the 64. Some versions of BASIC, like Simons' BASIC use the command WAVE 1,00010000 to set voice 1 to a triangular wave, and ENVELOPE to set up a voice's ADSR values. Both therefore require knowledge of the SID chip to use. The MUSIC command on some utilities has two parameters: one to set the tempo and a second to select notes. The string uses letters A–F (SHIFTed to indicate sharps; sometimes there is no notation for a flatted note available, which is irritating to some musicians). This is followed by a number 0–8 specifying the octave, with function keys to set durations, and a couple of other control characters. It calculates a set of frequencies corresponding to the notes. PLAY plays the tune previously created using MUSIC; but only one line can be played at a time. VOL, which sets the volume nybble, is the only other sound command. Three-part harmonies and so on aren't supported by Simons' BASIC.

Chapter 14

Tape Storage

- Loading and Saving BASIC Programs with Tape
- Handling Tape Data Files
- Loading and Saving Machine Language
- Tape Hardware Notes
- Advanced Tape Programming
- Copy Protection for Tape

Chapter 14

Tape Storage

Tape is a popular storage medium for the Commodore 64. This chapter discusses tape operations completely; it will give you the information you need to handle practically any tape operation.

Loading and Saving BASIC Programs with Tape

Loading and saving programs with tape is easy. 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 64; 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 the 64, 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	8	16	25
4K	1-1/2 min	1	3	6	9
8K	2-3/4 min	—	1	3	5

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 functions 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.

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 the 64 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 IRQ interrupt, locking out the keyboard. However, the RUN/STOP key subroutine is called at intervals, so RUN/STOP and RUN/STOP–RESTORE still work. Without this, if tape reading failed in some way, the 64 would have to be switched off. Note that the TI clock is turned off 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 num-*

ber. 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 an 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 GOTO is executed, pointing to the first line of the new program. Screen prompts don't appear when PLAY is pressed on the cassette. This 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.

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 Tape Data Files

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.

The 64'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 to save 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.

Files need a buffer. Unlike a program, which has a place allocated in memory at one time, files need to be written piecemeal, 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.

Commodore 64 tape files are inevitably sequential. Since data has to be written (and read back) in order, 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. 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, 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 64 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 64 supports only one tape drive, the normal 64 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, the screen blanks and before text returns to the screen, 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.

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.

Program 14-1. Using Files

```
10 OPEN 1:REM OPENS FILE#1 TO TAPE,TO READ
20 INPUT#1,X$:REM INPUT A STRING FROM FILE #1
30 PRINT X$:REM PRINT THE SAME STRING TO THE SCREE
   N
40 CLOSE 1:REM CLOSE THE TAPE FILE
```

Using Files Effectively

Note the 10 OPEN 1,1,0,"TESTING" is necessary if you wish to name the file to be read; the default parameters have to be put in. PRINT# is generally 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.

The full syntax of CLOSE 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 read-back.

ST will become 4 or 8 if a program is mistakenly read as a file. The errors mean that the program is either too short or too long to fit the buffer.

Loading and Saving 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. Data files aren't necessary. BLOCK LOAD and BLOCK SAVE, discussed in Chapter 6, provide methods (with examples) for doing this reliably.

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 the syntax SAVE "ML",1,1. Loading into an ML area changes BASIC pointers; use the BASIC command (*not the disk command*) NEW to set them back to normal.

A more easily understood method to save blocks of data is simply to PEEK individual bytes and write them as files. Reconstructing the data requires the reverse process of reading back the file and POKEing individual bytes to RAM. This method is slower than a BLOCK SAVE, though. Programs 14-2 and 14-3 illustrate the method, applied to bytes from $8000 to $9FFF. Other address ranges can of course be used instead:

Program 14-2. Writing Bytes to Tape

```
10 OPEN 1,1,1,"ROM AREA FILE"
20 FOR J=8*4096 TO 10*4096-1
30 P=PEEK(J)
40 PRINT#1,CHR$(P);:REM ; IS ESSENTIAL
50 NEXT
60 CLOSE 1
```

Program 14-3. Reading Bytes from Tape

```
10 OPEN 1
20 FOR J=8*4096 TO 10*4096-1
30 GET#1,P$
40 POKE J,ASC(P$+CHR$(0))
50 NEXT
60 CLOSE 1
```

Tape Hardware Notes

Many 64 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 64. 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 64 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 64 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 64 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 pinch wheel 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 pinch wheel is disengaged and one of the spools 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

You can connect an ordinary tape recorder to your 64, but 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 tape 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 the right-hand pin of the cassette port, looking from the 64'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 64, 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 64'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.

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. Pin D is connected to CIA 1. Pins E and F are connected to the 6510's built-in I/O port. Pin C (power for the motor) is also controlled by the 6510 I/O port.

Figure 14-1. 64 Tape Port

Pin	A	B	C	D	E	F
Function	Ground	+5 Volts	Motor On/Off	Read	Write	Cassette Key On/Off

The tape port is programmable. It is controlled by bit 5 of port 1; when this is 0, the motor is on (9 volts are delivered). There's a complication in that the keyboard interrupt servicing checks for a cassette keypress; if it finds one, it turns the motor on and otherwise it turns the motor off. However, this feature can be disabled. In fact, it has to be, to allow files to work properly.

Provided a cassette key is pressed, POKE 192,1: POKE 1, PEEK(1) OR 32 turns the motor off, and POKE 1, PEEK(1) AND 31 turns the motor on. When a cassette key isn't pressed, the interrupt always sets 192 to 0.

The cassette keypress can be detected by testing location 1. If you PEEK(1) AND 16 and the value returned is 0, a key is pressed; if PEEK(1) AND 16=1, a key isn't pressed. It follows that WAIT 1,16 waits until no cassette key is detected, and WAIT 1,16,16 waits for a cassette keypress. In ML programming use JSR $F82E followed by BNE if you want to detect if a key is pressed.

As Chapter 3 mentions, an accidental POKE can alter location 0, perhaps reconfiguring the data direction. Bit 3 is the tape write line; if it is set for input, SAVE won't work. RUN/STOP–RESTORE (or POKE 0,57) returns to normal.

Pin B of the cassette port is sometimes used to power equipment, such as low-power amplifiers and printers. This is a 5-volt power source.

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 64'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 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.

The tape buffer, which holds headers and file data, normally extends from $033C to $03FB (828–1019), a total of 192 bytes. You can change the location of the buffer 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 64 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 63553 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

The BASIC program below, Program 14-4, will identify and list programs and files on tape. It's easily modified if you wish. Note its use of SYS 63553. This is part of OPEN, but doesn't skip blocks starting with bytes other than 1, 3, 4, or 5. The program uses PRINT statements to display data on the screen; if the data contains control characters like 147 (clear screen) or 13 (carriage return), the display will be altered accordingly.

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

Program 14-4. Tape Directory

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
1Ø SYS 63553:PRINT"{WHT}"                          :rem 118
2Ø P=PEEK(828)                                       :rem 7
3Ø ON P GOTO 1ØØ,2ØØ,3ØØ,4ØØ,5ØØ                    :rem 24
4Ø PRINT "PROGRAM OR ML BLOCK":GOTO 3ØØØ            :rem 4Ø
99 REM TYPE 1                                      :rem 2Ø1
1ØØ PRINT "RELOCATABLE PROGRAM ";                  :rem 211
11Ø GOSUB 1ØØØ:GOSUB 2ØØØ:GOTO 4ØØØ                :rem 134
199 REM TYPE 2                                     :rem 251
2ØØ PRINT "BUFFER OF DATA ":GOTO 3Ø2Ø                :rem 4
299 REM TYPE 3                                     :rem 253
3ØØ PRINT "FORCED-LOAD PROGRAM ";:GOTO 11Ø:rem 188
399 REM TYPE 4                                     :rem 255
4ØØ PRINT "DATA FILE ";:GOSUB 1ØØØ:GOTO 4ØØØ
                                                   :rem 14Ø
499 REM TYPE 5                                       :rem 1
5ØØ PRINT" END-OF-TAPE MARKER";:GOTO 4ØØØ :rem 138
999 REM PRINT NAME FROM HEADER                      :rem 26
1ØØØ PRINT CHR$(34);:FOR J=833 TO 848:PRINT CHR$(P
     EEK(J));                                      :rem 227
1Ø1Ø NEXT:PRINT CHR$(34):RETURN                    :rem 155
```

```
1999 REM PRINT START AND END ADDRESSES      :rem 35
2000 PRINT "{2 SPACES}START=" PEEK(829) + 256*PEEK
     (830)                                  :rem 122
2010 PRINT "{4 SPACES}END=" PEEK(831) + 256*PEEK(8
     32)                                    :rem 191
2020 RETURN                                 :rem 164
2999 REM PRINT 192 BYTES OF DATA IF REQUIRED
                                            :rem 16
3000 INPUT "{CYN}VIEW THE PROGRAM";YN$      :rem 112
3010 IF YN$="N" GOTO 4000                   :rem 234
3020 PRINT "{WHT}" CHR$(34);: FOR J=828 TO 1019: P
     RINT CHR$(PEEK(J));: NEXT              :rem 212
3999 REM AWAIT KEYPRESS                     :rem 174
4000 PRINT:PRINT "{CYN}PRESS C TO CONTINUE"
                                            :rem 211
4010 GET X$:IF X$="C" THEN RUN               :rem 78
4020 GOTO 4010                              :rem 196
```

Another way to inspect header storage is POKE 178,0: POKE 179,4 to move the start-of-buffer to the start-of-screen. Then, OPEN 1 (for data and programs, SYS 63553) will load 192 bytes directly into the screen. FORJ=55296 TO 55296+192: POKEJ,1: NEXT will make all the bytes visible in white. Remember to POKE 178,60: POKE 179,3 or reset with SYS 64738 to return the buffer to normal.

To summarize, executing either a LOAD or an OPEN will cause 192 bytes to be read from whatever block is next on tape. If the first byte is 1 or 3, a program header is assumed found; if 2 or 4, a data file; and if 5, an end-of-tape marker. If BASIC or ML programs happen to start with a byte in the range 1–5, they'll give a spurious appearance as a header, data file, or end-of-tape marker. This can happen if LOAD or OPEN misses the header. In fact, this isn't likely to occur, since it's impossible for normal BASIC's first byte, which is part of the link address of the first line, to be less than 7, and the only ML commands in the range 1–5 are uncommon ORAs. Note that 0 isn't used as a marker because the earliest PETs saved the very first null byte as part of BASIC, so the headers, to avoid confusion, started with 1.

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.

When files are written or read, there's a pause between blocks; in this short interval, the VIC-II chip is reenabled and simultaneously the red LED on the recorder (if it has one) goes out. This periodic flashing at six-second intervals is typical of 64 files. (Programs have a single short screen enable just after the first copy is written or read.)

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 $E168. It uses the Kernal LOAD routine at $FFD5, which jumps to $F49E. All the parameters are set, and several branches test for the device number of 1; tape LOAD itself is at $F539. Normally, all programs have a header, and $F7EA finds a named header, while $F72C finds the next header of any kind. The routine at $F5D2 reads the program itself from tape into the correct part of RAM.

A typical loader designed to run an ML program follows:

```
LDA  #$01
TAX
TAY
JSR  $FFBA  ; File number, device, secondary address all 1
LDA  #$00
JSR  $FFBD  ; Filename irrelevant—load next tape program
JSR  $FFD5  ; Forced LOAD to stored address
JMP  $1000  ; Or other start address of ML to be loaded
```

This loads whatever program it finds next on tape with a forced LOAD, then jumps to address $1000.

SAVE's ROM entry point is $E156; its Kernal routine is $FFD8, which jumps to $F5DD. Tape saving is handled from $F65F; $F76A writes the header and $F867 the program.

ML saving might look like this:

```
LDA  #$01
STA  $BA    ; Device 1 (tape)
STA  $B9    ; Secondary address = 1 (forced LOAD in header)
LDX  #$00
LDY  #$20   ; End address is $2000 here
LDA  $FB    ; Start address presumed in ($FB)
JSR  $F5DD  ; 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 14-5 first loads the header, then loads the remaining program independently to start at any new address you choose.

Program 14-5. Load Anywhere

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
2 REM LINE 20 CONTROLS NEW START; EXAMPLE IS 1024
                                           :rem 54
10 OPEN 1:REM LOADS HEADER (OR SYS 63276) :rem 247
20 S=1024:REM START ADDRESS IS SCREEN     :rem 199
30 L=PEEK(831)-PEEK(829) + 256*(PEEK(832)-PEEK(830
   ))                                     :rem 216
40 E=S+L:REM COMPUTE NEW END ADDRESS FROM LENGTH
                                          :rem 168
50 POKE 830,S/256:POKE 829,S-INT(S/256)*256:REM S
   {SPACE}IN HEADER                         :rem 5
60 POKE 832,E/256: POKE 831,E-INT(E/256)*256:REM E
    IN HEADER                             :rem 201
70 POKE 781,3:REM FOR FORCED LOAD         :rem 109
80 SYS 62820:REM LOAD REST OF PROGRAM     :rem 146
```

Programmers using ML monitors can use the following ML routine to load a block into RAM.

```
LDA  #$00
STA  $93    ;LOAD, not SAVE
LDA  #$20
STA  $C2    ;Example start address
LDA  #$00
STA  $C1    ;is $2000, and
LDA  #$30
STA  $AF    ;example end address
LDA  #$00
STA  $AE    ;is $3000
JSR  $F5A2  ;or $F5A5 omitting the loading message
BRK
```

Writing Tape Data Anywhere from RAM

This is best done with ML, using a routine like that to load tape blocks. Program 14-6 uses a good method:

Program 14-6. Save Anywhere

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
100 FOR J=320 TO 356:READ X:POKE J,X:NEXT  :rem 51
110 FOR J=828 TO 1018:POKE J,32:NEXT      :rem 204
200 PRINT "WRITE HEADER/FILE TO TAPE; OR EXIT"
                                          :rem 201
210 INPUT "(H/F/X)"; YN$                    :rem 2
220 IF YN$="X" THEN END                   :rem 205
230 IF YN$="F" GOTO 500                   :rem 132
299 REM THIS WRITES A HEADER ONLY TO TAPE :rem 151
300 PRINT "ENTER FIRST 5 PARAMETERS OF HEADER"
                                          :rem 209
310 PRINT " EXAMPLE: 3,0,192,0,208 MEAN"   :rem 69
320 PRINT " FORCED PROGRAM LOAD FROM C000-D000"
                                           :rem 89
330 INPUT A,B,C,D,E                        :rem 37
340 POKE 828,A:POKE 829,B:POKE 830,C:POKE 831,D:PO
    KE 832,E                              :rem 174
350 PRINT "PROGRAM NAME TO BE PUT IN HEADER":rem 5
360 INPUT N$                              :rem 155
370 FOR J=1 TO LEN(N$):POKE 832+J,ASC(MID$(N$,J)):
    NEXT                                  :rem 253
380 SYS 320:REM WRITE HEADER TO TAPE       :rem 78
390 RUN                                   :rem 145
499 REM THIS WRITES ANY BLOCK OF MEMORY TO TAPE
                                           :rem 46
500 INPUT "START ADDRESS OF BLOCK TO BE SAVED";S
                                           :rem 40
510 INPUT "{2 SPACES}END ADDRESS OF BLOCK TO BE SA
    VED";E                                :rem 100
520 POKE 321,20                           :rem 234
530 POKE 325,S/256:POKE 329,S-INT(S/256)*256
                                          :rem 129
540 POKE 333,E/256:POKE 337,E-INT(E/256)*256
                                           :rem 86
550 SYS 320:REM WRITE BLOCK TO TAPE        :rem 15
560 RUN                                   :rem 144
996 REM FOLLOWING ML IS SET UP FOR HEADER :rem 147
997 REM LINE 1000'S 105 = LENGTH OF TONE; :rem 139
998 REM LINE 1000'S 3 AND 60 SET START=$033C;
                                           :rem 76
999 REM LINE 1010'S 3 AND 252 SET END =$03FC.
                                          :rem 208
1000 DATA 169,105,133,171,169,3,133,194,169,60,133
     ,193                                 :rem 103
```

```
1010 DATA 169,3,133,175,169,252,133,174,169,1,133,
     184,133                                 :rem 251
1020 DATA 186,169,0,133,183,169,255,133,185,76,107
     ,248                                    :rem 119
```

The ML routine contains five parameters which control the header tone. Any user-defined header can be written to tape, for example, to cause a forced LOAD into a normally difficult area of memory. And any consecutive block of bytes can be written to tape, allowing great flexibility in program construction.

Copy Protection for Tape

Security is an interesting aspect of tape programs. Before taking it too seriously, it's worth remembering that it may be possible to copy tapes 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 REM, 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 control characters that clear the screen or change the cursor color, or it could even be something like ?LOAD ERROR. All that's needed is SAVE "NAME" + CHR$(147) + "ERROR" + CHR$(31) or other analogous strings.

Using the Screen Positions

ML programs are sometimes designed to load into the default screen position set by the 64. An ML jump to $0400, for instance, makes the program impossible to stop in the usual way, as it will be corrupted or entirely erased by pressing RUN/STOP–RESTORE. This type of program can be developed either by moving the screen to a nonstandard position or by writing the header separately from the program.

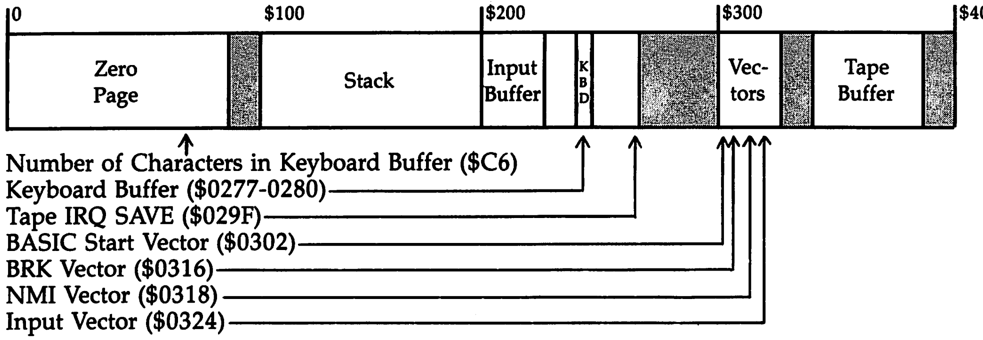
Using Headerless Programs

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
89 bytes $02A7–$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 $013E if cautious about tape reading.

BASIC warm start vector in ($0302). Normally $A483, this can be directed either into the header or into the loaded program, perhaps spanning $02A7–$0303. Then ML LOAD and RUN will automatically run the BASIC program that comes afterward.

Input vector in ($0324). Again, a loader might span $02A7–$0325, so the altered input vector might jump to $02A7.

Tape interrupt vector at ($029F). This vector is difficult to use on the 64, since it is followed by operating system variables.

A BASIC Autoloader

The autoload routine in Program 14-7 will run the BASIC program immediately following it on tape. All that's required is to enter LOAD. It disables RUN/STOP and RUN/STOP–RESTORE, and scrambles LIST, to give some program protection.

First, add these few program lines to the program which writes any data to tape. Run the program, with a rewound tape in the recorder, and prepare to write a header: the parameters to put in are 3, 167, 2, 4, and 3. (Forced LOAD into $02A7–$0304. The very last address is unused, so the ML will straddle $02A7 through $0303.)

Program 14-7. BASIC Autoloader

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
100 FOR J=320 TO 356:READ X:POKE J,X:NEXT  :rem 51
110 FOR J=828 TO 1018:POKE J,32:NEXT      :rem 204
200 PRINT "WRITE HEADER/FILE TO TAPE; OR EXIT"
                                          :rem 201
210 INPUT "(H/F/X)"; YN$                    :rem 2
220 IF YN$="X" THEN END                   :rem 205
230 IF YN$="F" GOTO 500                   :rem 132
299 REM THIS WRITES A HEADER ONLY TO TAPE :rem 151
300 PRINT "ENTER FIRST 5 PARAMETERS OF HEADER"
                                          :rem 209
310 PRINT " EXAMPLE: 3,0,192,0,208 MEAN"   :rem 69
320 PRINT " FORCED PROGRAM LOAD FROM C000-D000"
                                           :rem 89
330 INPUT A,B,C,D,E                        :rem 37
340 POKE 828,A:POKE 829,B:POKE 830,C:POKE 831,D:PO
    KE 832,E                              :rem 174
350 PRINT "PROGRAM NAME TO BE PUT IN HEADER":rem 5
360 INPUT N$                              :rem 155
370 FOR J=1 TO LEN(N$):POKE 832+J,ASC(MID$(N$,J)):
    NEXT                                  :rem 253
380 SYS 320:REM WRITE HEADER TO TAPE       :rem 78
390 RUN                                   :rem 145
499 REM THIS WRITES ANY BLOCK OF MEMORY TO TAPE
                                           :rem 46
500 INPUT "START ADDRESS OF BLOCK TO BE SAVED";S
                                           :rem 40
510 INPUT "{2 SPACES}END ADDRESS OF BLOCK TO BE SA
    VED";E                                :rem 100
520 POKE 321,20                           :rem 234
530 POKE 325,S/256:POKE 329,S-INT(S/256)*256
                                          :rem 129
540 POKE 333,E/256:POKE 337,E-INT(E/256)*256
                                           :rem 86
545 FOR J=828 TO 854:READ X:POKE J,X:NEXT  :rem 80
546 FOR J=855 TO 922:POKE J,PEEK(J-149):NEXT
                                          :rem 210
547 POKE 919,167:POKE 920,2               :rem 158
550 SYS 320:REM WRITE BLOCK TO TAPE        :rem 15
560 RUN                                   :rem 144
996 REM FOLLOWING ML IS SET UP FOR HEADER :rem 147
997 REM LINE 1000'S 105 = LENGTH OF TONE; :rem 139
998 REM LINE 1000'S 3 AND 60 SET START=$033C;
                                           :rem 76
999 REM LINE 1010'S 3 AND 252 SET END =$03FC.
                                          :rem 208
1000 DATA 169,105,133,171,169,3,133,194,169,60,133
     ,193                                 :rem 103
1010 DATA 169,3,133,175,169,252,133,174,169,1,133,
     184,133                              :rem 251
```

```
1020 DATA 186,169,0,133,183,169,255,133,185,76,107
     ,248                                   :rem 119
2000 DATA 169,234,141,40,3,169,131,141,2    :rem 159
2010 DATA 3,169,164,141,3,3,169,131,141     :rem 114
2020 DATA 119,2,169,1,133,198,108,0,160     :rem 117
```

When the header is written, prepare to write data. Use a starting address of 828 and an ending address of 921. A short ML program is put at the start of the buffer, which will be loaded and run at $02A7. It disables RUN/STOP, restores ($0302) to normal, puts #$83 (SHIFT–RUN/STOP) into the keyboard buffer, and executes a JMP ($A000). This is enough to force the next program to load, then run. The final bytes of the buffer hold the modified address, $02A7, which will be force-loaded into ($0302) and hence start the whole process.

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.

There is considerable scope for ingenuity in using encoding routines, nonstandard 6510 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 Disk Storage

- Introduction to Disk Storage
- Basic Disk Commands
- Handling Disk Files with BASIC
- Summary of Disk Commands and Messages
- Commodore Utility Programs
- Hardware Notes
- Disk Data Storage
- Machine Language Disk Programming

Chapter 15

Using Disk Storage

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.

Introduction to Disk Storage

The Commodore 64'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 model was 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 Commodore 64.

The 64's disk units store data on 5-1/4-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 64, and then any printer, but the order 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 64'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 64 tape systems.

So-called black boxes are available to allow several 64s 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 (approximately 40 feet).

The 64's disk drive is autonomous. In other words, it is largely independent of the 64. In fact, it has as much ROM as the 64 itself. When the drive receives a command from the 64, 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 64 to read them. This explains how it is possible for the 64 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 64 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 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 64 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, quotation marks and names are used. For example, to simply read the disk directory you must type in LOAD "$",8 then LIST. The DOS 5.1 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 64 and the disk drive.

Formatting a Disk

Switch on the disk drive and the 64, insert a new, *blank* disk (or one that contains information you no longer wish to keep) in the drive, and close the drive door. You're now 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 1541-format 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.

To format a disk, type in the following command and press RETURN:

OPEN 15,8,15,"NEW0:*NAME,ID*":CLOSE 15

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.

If you want to change the disk's name, you can use a shorter formatting command. Omit ,ID from the formatting command; the name will be changed and the data apparently will all be deleted, exactly like full formatting, but the ID isn't changed. This takes about ten seconds.

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 processed.

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 below. 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, if PROGRAM is the first name in the directory.

LOAD *"filename"*,8,1 is necessary for a nonrelocatable LOAD. Machine language, graphics definitions, VIC-II chip registers, and any data which needs to be replaced at the point from which it was saved uses this syntax.

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. You can use S as the abbreviation for SCRATCH.

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

OPEN 15,8,15:PRINT#15,"I":CLOSE 15

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.
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 the vector at (43–44) with the low byte and high byte of the starting address, and POKE the vector at (45–46) 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.

General Copier for Programs

For straightforward programs, the copying methods above are fine. Where these methods fail, or if you simply want an easy copying method, use something like Program 15-1, which reads a program from one disk, storing it above BASIC in RAM, then writes it back to another disk. Program 15-1 does not work properly, however, if the program to be copied is written in BASIC and ends with a semicolon. (Speed can be improved by reading and writing in ML.)

Program 15-1. General Program Copier

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 POKE 55,0: POKE 56,12: CLR: REM FREE RAM STARTS
    $0C00                                       :rem 204
20 PRINT "NAME OF PROGRAM TO BE COPIED"        :rem 223
30 INPUT N$                                     :rem 101
40 PRINT "{DOWN}NAME OF PROGRAM AFTER COPYING"
                                                :rem 159
50 INPUT M$                                     :rem 102
60 OPEN 15,8,15                                 :rem 244
100 PRINT "{DOWN}INSERT DISK HOLDING ORIGINAL PROG
    RAM"                                        :rem 229
110 PRINT" THEN PRESS RETURN"                   :rem 255
120 GET X$:IF X$<>CHR$(13) GOTO 120              :rem 48
130 PRINT "{DOWN}OK... READING " N$:PRINT :rem 205
200 OPEN 1,8,2,N$+",P,R"                        :rem 190
300 FOR J=3072 TO 40960                         :rem 115
310 GET#1,X$:IF ST>0 GOTO 400                    :rem 27
320 POKE J,ASC(X$+CHR$(0))                      :rem 139
330 NEXT:PRINT "TOO LONG":END                    :rem 19
400 CLOSE 1:PRINT "{DOWN}INSERT DESTINATION DISK"
                                                :rem 153
410 PRINT " THEN PRESS RETURN"                    :rem 2
420 GET X$:IF X$<>CHR$(13) GOTO 420              :rem 54
430 PRINT "{DOWN}OK... WRITING " M$              :rem 50
440 PRINT#15,"I"                                :rem 103
500 OPEN 1,8,2,M$+",P,W"                        :rem 197
600 FOR K=3072 TO J                             :rem 190
610 PRINT#1,CHR$(PEEK(K));                      :rem 242
620 NEXT                                        :rem 215
700 CLOSE 1:CLOSE 15:PRINT "COPY COMPLETED":rem 55
```

Quite a number of copy utilities are on the market. Single-drive copiers need only one drive. There are two basic types: Some copy an entire disk, reading as much as possible into RAM, then copying an exact image onto a new disk. More sophisticated versions copy only those parts of the disk on which data is stored. One disk typically takes 15 minutes and six disk changes. The other type copies individual programs or files, allowing the user to select only those worth copying. This second type (the program above is like this) is fine when small numbers of programs are to be copied, but tedious with large numbers of programs, because the disks typically are changed between each read/write operation.

Two-drive copiers use two daisychained drives. The technique is to switch on *one* drive, load and run a disk-device number-change program to reassign it as drive 9 (unless the drive's been changed in hardware—see below), turn the other drive on, and load the copy program. "COPY/ALL" on the demo disk is this type of program. Again, disks can be copied in entirety or copied one file at a time by utilities.

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 understand this better, enter and run the following one-line BASIC program:

```
10 OPEN15,8,15:INPUT#15,E,E$,T,S:PRINT E;E$;T;S:CLOSE15: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. (Assuming PROGRAM is still present on the disk.) RUN yields:

   ```
   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 yields:

   ```
   1 FILES 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 drive door if there's a disk present; this insures that no magnetic glitch can occur on the disk. Turn on the disk drive and immediately enter RUN. 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 the DOS 5.1 wedge from the demo disk, 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 64'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 64's tape unit. The tape system can use only a single track of tape, whereas a disk uses a multitrack system.

Types of File Organization

The 64'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 programmers to build their 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 programmers. 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, 64 sequential files—whether on tape or disk—aren't usually quite so free from structure. This is 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 64, 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 64.

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. The 64'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–Z plus a full-length relative file works well in some applications.

Writing and Reading Disk Files

OPEN. You can OPEN a disk file by 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:. This chapter ignores 0:, but it is often suggested that the prefix should be used. Regardless, readers with access to PET/CBM machines should keep this syntax in mind.

The other part of the command string uses commas as separators and causes a sequential (S) file, called ORDINARY FILE, to be set up for writing (W). 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 0. 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 not be 0, 1, or 15. This is because 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 ten spaces (in effect, tabulating 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. The two program lines, PRINT#2,CHR$(N); and GET#1,X$: N=ASC(X$+CHR$(0)), are exact mirror images. One writes a single character (the semicolon prevents unwanted RETURNs) and the other reads the character back, also converting it back into its ASCII value, allowing for the null-character bug in the 64's ASC command.

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#. This reads individual characters from a file, with no exceptions. It will fetch null characters written as CHR$(0), quotation 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 ***file number***

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" below 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 bits 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 an easy 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. Usually, it does not mean that something horrible has happened to the disk.

The red light combines several informative functions. 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 written only 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 buffer 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 *?FILE NOT FOUND.* Type the correct version, LOAD "PROGRAM",8, and loading will proceed normally with no problems. The same sort of thing applies in a program: 10 OPEN 2,8,2,"FIEL,S,R" might 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

```
1 OPEN 15,8,15,"I":REM INITIALIZE DISKETTE
2 OPEN 2,8,2,"PROGRAM FILE DEMO,P,R":REM OPEN PRG
  {SPACE}FILE FOR READ
3 GET#2,X$:REM GET A FILE CHARACTER
4 IF ST>0 THEN CLOSE 2:END
5 PRINT ASC(X$+CHR$(0));:REM PRINT ASCII VALUE
6 GOTO 3
```

Line 2 must include the name of the program to be examined. Alternative 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 8	14 8	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 manipulations are possible. The link address need not be correct, since LOAD will relink it.

It's possible to write BASIC directly 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, you may use Program 15-3.

Program 15-3. Finding ML or Memory Dump LOAD Addresses

```
10 INPUT "PROGRAM NAME";N$
20 OPEN 2,8,2,N$+",P,R"
30 GET#2,X$,Y$
40 PRINT ASC(X$+CHR$(0))+256*ASC(Y$+CHR$(0))
50 CLOSE 2
```

Analyzing ML 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

```
10 OPEN 2,8,2,"ML FILE1,P,R"
20 OPEN 3,8,3,"ML FILE2,P,W"
30 GET#2,X$,X$
40 PRINT#3,CHR$(0)CHR$(192);
50 GET#2,X$:IF X$="" THEN X$=CHR$(0)
60 S=ST:PRINT#3,X$;
70 IF S=0 GOTO 50
80 CLOSE 2:CLOSE 3
```

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 $C000. Note in line 50 how a character which GET# regards as null must be converted into CHR$(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 64 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 $02A7. The autorun feature is caused by changing the vector at $0302–$0303 to $02A7, where the ML program is loaded by name (in effect, with LOAD "ML",8,1) and then jumped to.

To use this program, you need a BASIC program on disk, Program 15-5 (below) in memory, a new name for the program, plus a two-line message. When run, the program adds ML from $02A7 to the start of BASIC. In other words, LOAD "NEWNAME",8,1 puts BASIC into its normal position, but prefaced by about five blocks of ML which autoruns. It also straddles the screen, so the message which is input appears before the program runs (it's timed to stay for about five seconds and appears black-on-white). RUN/STOP and RUN/STOP–RESTORE are automatically disabled. Some of the ML simulates plug-in ROM at $8000, which means that even a hardware reset cannot break into the program as it runs.

Autorun assumes BASIC starts at $0801, as it normally does; VIC-20 is more awkward to autorun than the 64 is, because its starting position varies with memory expansion. Test the new program, then delete the pure BASIC version from the disk; now you have an autorunning BASIC program.

Program 15-5. ML Autorun

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
20 REM DATA MARKED WITH ASTERISKS PROTECTS AGAINST
                                           :rem 132
21 REM HARDWARE RESET; IF THIS ISN'T REQUIRED,
                                            :rem 12
22 REM OMIT THE DATA. ALSO OMIT IF BASIC USES
                                            :rem 67
```

```
23 REM RAM AFTER $8000.                          :rem 181
24 REM                                            :rem 74
100 PRINT "MESSAGE, STATEMENT, OR TITLE"         :rem 151
110 INPUT M$                                     :rem 147
120 INPUT "PRESENT PROGRAM NAME";N$              :rem 110
130 INPUT "AUTORUN PROGRAM NAME";P$              :rem 126
140 PRINT "{CLR}{2 DOWN}": PRINT M$               :rem 83
200 OPEN 15,8,15,"I"                             :rem 217
210 OPEN 1,8,3,N$+",P,R"                         :rem 192
220 OPEN 2,8,4,P$+",P,W"                         :rem 202
230 FOR J=679-2 TO 9999                          :rem 143
240 READ X: IF X<0 THEN 300                       :rem 89
250 PRINT#2,CHR$(X);:NEXT                          :rem 3
300 FOR K=J TO 2048                              :rem 189
310 IF K=770 THEN PRINT#2,CHR$(167);:NEXT         :rem 42
320 IF K=771 THEN PRINT#2,CHR$(2);:NEXT          :rem 192
330 IF K=808 THEN PRINT#2,CHR$(234);:NEXT         :rem 41
340 PRINT#2,CHR$(PEEK(K));:NEXT                  :rem 108
400 GET#1,X$,X$                                   :rem 24
410 GET#1,X$:IF ST=64 THEN CLOSE 1:CLOSE 2:CLOSE 1
    5:END                                        :rem 103
420 PRINT#2,CHR$(ASC(X$+CHR$(0)));:GOTO 410
                                                 :rem 138
500 DATA 167,2,169,131,141,2,3,169,164,141 :rem 12
510 DATA 3,3,169,1,133,43,169,8,133,44,169 :rem 20
520 DATA 0,168,153,0,216,136,208,250,169  :rem 173
530 DATA 1,141,33,208,133,162,165,162,208,252
                                                 :rem 157
540 DATA 162,5,189,15,253,157,3,128,221,3: REM ***
    *                                            :rem 168
550 DATA 128,208,251,202,208,242,169,2,141: REM **
    **                                           :rem 213
560 DATA 1,128,169,254,141,3,128,169,245,141,0: RE
    M ****                                       :rem 159
570 DATA 128,169,188,141,2,128,169,128,133,56: REM
     ****                                        :rem 132
580 DATA 169,234,141,40,3,32,89,166,76,174,167,-1
                                                 :rem 121
```

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 64 tape, is 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 64.

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 need not be 2, and the device number does not have to 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*"

The above combines two or more 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

```
10 PRINT "NAME OF SEQ FILE TO BE DISPLAYED"
20 INPUT N$
30 OPEN 1,8,2,N$:REM OPEN SEQ FILE (DEFAULT= READ)
40 GET#1,X$
50 PRINT X$;
```

```
60 IF ST>0 THEN CLOSE 1:END:REM STOP AT END OF FIL
   E
70 GET X$:IF X$="" THEN 70
80 GOTO 40
```

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 S=ST) and change ST in line 60 to S; the program then tests for all error conditions.

After making the above changes, the program will look as follows:

```
5 OPEN 15,8,15
10 PRINT "NAME OF SEQ FILE TO BE DISPLAYED"
20 INPUT N$
30 OPEN 1,8,2,N$:REM OPEN SEQ FILE (DEFAULT= READ)
35 GOSUB 10000
40 GET#1,X$
42 S=ST
45 GOSUB 10000
50 PRINT X$;
60 IF S>0 THEN CLOSE 1:END:REM STOP AT END OF FILE
70 GET X$:IF X$="" THEN 70
80 GOTO 40
10000 INPUT#15,E,E$,T,S:IF E<20 THEN RETURN
10010 PRINT "***DISK WARNING":PRINT E;E$;T;S:CLOSE
      1
```

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 work only 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, followed by 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$(L) to open a relative file. As usual, the file number and channel may take a range of values, and the device number may not be 8. Using L is compulsory when the file is set up for the first time; it is followed by the record length, which must allow for a RETURN character. For example, use CHR$(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 tricky: 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 above. The low/high byte format 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 read of the error channel. To end the program, enter a record number of 99999.

Program 15-7. Handling Relative Files

```
10 OPEN 15,8,15
20 PRINT "REL. FILENAME":INPUT N$
30 PRINT "RECORD LENGTH":INPUT L
40 OPEN 1,8,2,N$+",L,"+CHR$(L+1)
50 INPUT "RECORD#";R
60 IF R=99999 THEN CLOSE 1:END
70 RH%=R/256:RL=R-RH%*256
80 PRINT "RECORD":INPUT R$
90 R$=LEFT$(R$,L)
100 PRINT#15,"P"+CHR$(2)+CHR$(RL)+CHR$(RH%)+CHR$(1
    )
110 PRINT#1,R$
120 GOTO 50
```

Line 40 opens a relative file, named by the user, and assigns a record length one greater than the value entered (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 could 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 0 exists). One thousand records of length 21 occupy 21,000 bytes; these are stored in 254-byte sectors, so the data occupies 83 sectors. (Commodore 64 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 1 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 apparently does not have this fault.

Summary of Disk Commands and Messages

The following list summarizes the 64 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. The DOS 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 disk drive 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 64'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. Be careful with this command.

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 Validate 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 below. Validate is also useful for cleaning up a heavily used disk. If you have scratched and resaved programs many times (for example, during program development), the disk may contain more free blocks than are shown by LISTing the directory. Validate will reorganize such a disk and free up all of the unused blocks.

Pattern Matching

Disk commands involving loading or opening for reading 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 question mark (?) 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.

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 hang-up. Try turning your printer off.

See Table 15-1 for a summary of the messages generated by the disk drive.

File problems. Unclosed files are signaled with an asterisk in the directory entry (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 reading, or recover some of the file data, using OPEN 2,8,2,"*filename*,M" which enables unclosed files of all types to be read as far as possible.

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 relative 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	

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 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, using BASIC's pointers at locations 45 and 46 to locate the end.

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

The 64 wedge. Use this to make direct mode disk commands simpler. The wedge will not coexist with some other utilities. It adds these direct mode commands:

@ alone reads the disk status and prints it,
@$ reads and displays the directory, without affecting BASIC,
/PROGRAM loads PROGRAM.

Some versions allow a LOAD and RUN option, and an abbreviated SAVE command. All versions allow > (a wedge) as an alternative to @.

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
This program simplifies disk commands for the VIC-20, as DOS 5.1 does for the 64.

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 15. At least four drives can be daisychained together, so in principle, a four-drive system would be feasible. Given the right hardware, a single 64 can also share a disk drive with other 64s.

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.

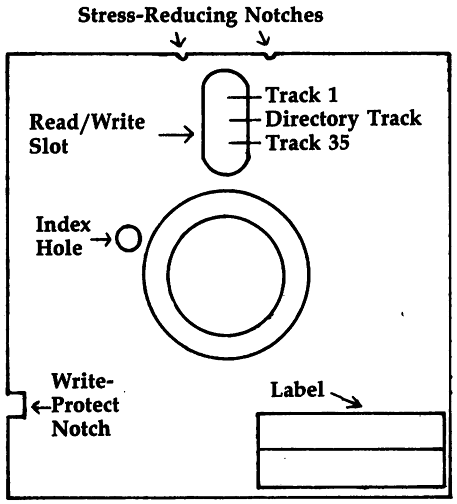
These drives use 35 tracks. The actual magnetized zones are about 1/60 inch wide; the clutch mechanism which grips the diskette has to position it within that tolerance.

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
1540/1541 disk drives use 5-1/4-inch 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.*

Figure 15-3. A Typical Diskette



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 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 that both sides are usable. The index hole isn't a factor; 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 *continuous* 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 cross-over 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.

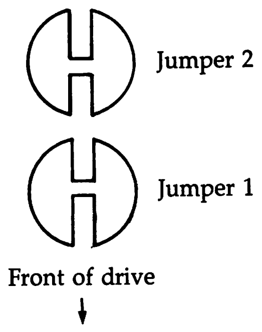
Software conversion is easily done using CHANGE DISK ADDR on the demo disk. This program, which works with any Commodore disk, writes the new device number into two disk RAM locations. Commodore disks vary a great deal internally, so the program also has to work out the type of disk drive. With the 64, 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 8–15.

When two drives are used, they must be turned on separately. Typically, one drive is turned on, then the 64 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 respective 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. Note that opening the drive case to do this will probably void your warranty. *To avoid severe electrical shock,* do not attempt any such operation until you have turned the drive off and unplugged every connector. If you're not sure what's involved, get help from someone who understands electronics.

Disk ROM

Commodore disk drives have internal ROM from $C000 to $FFFF and RAM from $0 to $07FF. 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 the result of using GET#, which in this case 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.

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 64 is used for any serious purpose.

Hardware 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 Data Storage

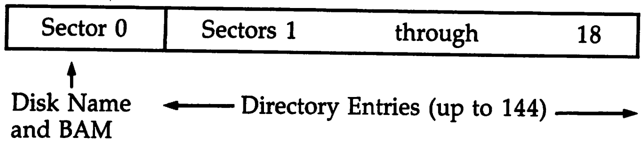
As stated earlier, Commodore disks have 35 tracks. Of those tracks, 17 have 21 sectors each, 7 have 19 sectors each, 6 have 18 sectors each, and 5 have 17 sectors each. That gives a total of 683 sectors. Track 18 holds the directory information. Subtracting 19 for the directory gives 664 blocks free, as reported by the directory for an empty disk. And 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. 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
```

The value of the first byte is 21 ($15), which means that all 21 sectors of track 1 are free. The hex value in the second and third bytes is $FF (bit pattern 1111 1111), showing that sectors 0–7 and 8–15 are all free. The hex value of the fourth byte is $1F (bit pattern 0001 1111), meaning sectors 16 through 20 are unused as well. 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 2 bytes of the sector, which serve as a 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 Validate. $80 is a scratched unclosed file, a type to be avoided.
3–4	Track and sector pointer to first block of file
5–20	Filename + shifted spaces ($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 side sector (of a possible six), 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.

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

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.)

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. Since 1541 disk drives allow five channels, 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 and before the filename, show the track and sector of the first block. DISPLAY T&S outputs hex numbers and has been modified from earlier versions to automatically read chained blocks when desired.

Program files (type $82) can be either BASIC or ML dumps. The first two bytes are the LOAD address (for example, 01 08, $0801, for 64 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 and other, more sophisticated disk examination programs will allow you to explore, and the system concepts will soon be easier to understand.

The Disk Directory

Both the entire disk directory track and the directory program can be read from BASIC. The information here will help you examine or modify disk programs, files, or directory entries by writing directly onto the disk.

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 "$:64*",8 lists all programs and files beginning with 64.
LOAD "$:??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,"$". Use of 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

```
1Ø OPEN 1,8,Ø,"$"
2Ø GET#1,X$,X$
3Ø GET#1,X$,X$,X$,X$
4Ø IF ST THEN CLOSE 1:END
5Ø GET#1,X$: IF X$="" THEN PRINT:GOTO 3Ø
6Ø IF X$=CHR$(34) THEN Q=NOT Q
7Ø IF Q THEN PRINT X$;
8Ø GOTO 5Ø
```

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

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
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
2ØØØ N=3: GOSUB 1ØØØØ: K=K+1: GET#1,N1$: GET#1,N2$
     : IF ST>Ø GOTO 2ØØØØ                     :rem 24
2Ø1Ø D$(K) = STR$(ASC(N1$+CHR$(Ø)) + 256*ASC((N2$)
     +CHR$(Ø))) + " "                         :rem 21
2Ø2Ø FOR J=1 TO 27: GET#1,X$                 :rem 133
2Ø3Ø D$(K)=D$(K)+X$: NEXT                     :rem 42
2Ø4Ø GOTO 2ØØØ                               :rem 193
1ØØØØ FOR J=1 TO N: GET#1,X$: NEXT: RETURN :rem 42
2ØØØØ CLOSE 1: CLOSE 15                      :rem 175
3ØØØØ SYST:D                                 :rem 196
4ØØØØ OPEN 4,3: REM OR OPEN 4,4 TO DISPLAY TO PRIN
      TER                                    :rem 19Ø
4ØØØ5 PRINT#4,CHR$(147)                      :rem 247
4ØØ1Ø FOR J=Ø TO K-1: PRINT#4,"{1Ø SPACES}" D$(J):
       NEXT                                  :rem 233
4ØØ2Ø FOR J=1 TO 1Ø: PRINT#4: NEXT            :rem 48
4ØØ3Ø CLOSE 4                                :rem 161
4ØØ4Ø CLR: GOTO 1ØØØ                          :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

```
20000 OPEN 100,8,0,"$:U=U"
20010 FOR J=1 TO 35:GET#100,X$:NEXT
20020 GET#100,Y$:CLOSE 100
20030 BF=ASC(X$+CHR$(0)) + 256*ASC(Y$+CHR$(0))
20040 PRINT BF"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 the BAM

```
10 Z$=CHR$(0):OPEN 2,8,2,"$"
20 GET#2,X$,X$
30 FOR J=1 TO 35:GET#2,A$,B$,C$,D$
40 PRINTJ ASC(A$+Z$) ASC(B$+Z$) ASC(C$+Z$) ASC(D$+
   Z$)
50 NEXT:CLOSE 2:END
```

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

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 DIM X$(30)                               :rem 107
20 OPEN 15,8,15,"I"                         :rem 169
30 OPEN 3,8,3,"#"                            :rem 30
40 OPEN 2,8,2,"$"                            :rem 30
50 FOR J=1 TO 254:GET#2,X$:NEXT             :rem 210
100 FOR J=1 TO 8                             :rem 11
110 FOR K=1 TO 30:GET#2,X$(K):NEXT          :rem 100
120 IF X$(1)<>CHR$(130) GOTO 200             :rem 75
```

```
130 FOR K=4 TO 19:PRINT X$(K);:NEXT          :rem 215
140 PRINT#15,"U1:";3;0;ASC(X$(2)+CHR$(0));ASC(X$(3
    )+CHR$(0))                               :rem 211
150 GET#3,X$,X$,L$,H$                         :rem 80
160 PRINT ASC(L$+CHR$(0)) + 256*ASC(H$+CHR$(0))"
    {LEFT}"                                  :rem 125
200 IF J<8 THEN GET#2,X$,X$                  :rem 147
210 NEXT J                                    :rem 28
220 IF ST=0 GOTO 100                           :rem 1
300 CLOSE 2:CLOSE 3:CLOSE 15:END              :rem 69
```

Line 50 skips the BAM sector, and line 110 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 after 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 the commands that give the 64 direct control of the disk drive. There are three types of direct access commands: those that read or write on individual disk 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, which Commodore does not publish. 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 reallocates them in the BAM and leaves them at risk of being overwritten.

Direct access commands are powerful; some of them can garble an entire disk if misused. If you want to experiment with them, use an unimportant disk until you have gained some experience.

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 block read (B-R) and block write (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 instead of 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 15,8,15:OPEN 1,8,2,"#"
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 analogous to the P parameter used with relative files.

Program 15-14. Using Block Write

```
10 OPEN 15,8,15:OPEN 1,8,2,"#"
20 PRINT#15,"U1";2;0;18;1
```

```
30 PRINT#15,"B-P";2;2
40 PRINT#1,CHR$(130+64);
50 PRINT#15,"U2";2;0;18;1
60 CLOSE 1:CLOSE 15:END
```

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, the directory will never be read into the buffer; as a result, 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 the BAM low, to show that a sector is in use. A bit value of 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, the BAM can accurately reflect blocks written to disk by Block Write.

B-F (Block Free). The block free command sets a bit in the 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 1–8 with the directory entries in track 18, can be used to read from 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 a routine in ROM, or a routine in RAM 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 bytes and the other parameters are the starting address in RAM and the number of bytes.

Machine Language Disk Programming

LOAD and SAVE

BLOCK LOAD and SAVE are discussed in Chapter 6. These work from within a program without disturbing its sequence of operations.

The autorunning loader in the section on program files uses Kernal subroutines, as shown below. Note that a name is necessary with disks, even if it's only "*".

```
LDA  #$01      ;FILE NUMBER
LDX  #$08      ;DEVICE NUMBER
LDY  #$00      ;SECONDARY ADDRESS
JSR  $FFBA     ;SETLFS
LDA  #LENGTH   ;NAME LENGTH
LDX  #LOW      ;START OF NAME
LDY  #HIGH
JSR  $FFBD     ;SETNAM
LDA  #$00
STA  $0A       ;LOAD/VERIFY FLAG 0ff
JSR  $FFD5     ;LOAD            SAVE = FFD8
JMP  START
```

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 a pointer to 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.

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  #$01
     JSR  $FFC6  ;OPEN FILE 1 FOR INPUT
     JSR  $FFCF  ;INPUT A BYTE (LIKE GET#)
     PHA         ;STORE IT
     LDY  $90    ;LOAD ST
     LDX  #$02
     JSR  $FFC9  ;OPEN FILE 2 FOR OUTPUT
     PLA         ;RECOVER BYTE
     JSR  $FFD2  ;OUTPUT IT (LIKE PRINT#)
     CPY  #$00
     BEQ  LOOP   ;CONTINUE IF ST IS 0
     LDA  #$01
     JSR  $FFC3  ;CLOSE 1
     LDA  #$02
     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 I/O to normal operation.

Program 15-15 gives another, shorter example. It is POKEd from BASIC, so try it if you're inexperienced in ML. It displays 256 bytes from an open file 1 on a Commodore 64. 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. More SYS 828 commands read further, and 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.

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

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
110 DATA 207,255,153,0,4,169,1,153,0
120 DATA 216,200,208,242,76,204,255
```

OPEN and CLOSE in ML

OPEN uses SETLFS to set the parameters for *logical, first, and secondary addresses,* typified by 1, 8, and 2 in OPEN 1,8,2"*filename*". These three parameters are often referred to as file number, device number, and channel number, respectively. Use the following:

```
LDA  # file number
LDX  #8
LDY  # channel number
JSR  $FFBA
```

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 numbers 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  #$08
      LDY  #$0F
      JSR  $FFBA   ;SET 15,8,15
      LDA  #$00    ;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  $FFD2   ;PRINT TO SCREEN
      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 (CLRCHN) 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 Control Ports

- Joysticks
- Paddles and Other Analog Devices

The Control Ports

This chapter explains how to program controllers which fit the 64's control ports. Fast ML routines are supplied which can be used to write more efficient programs.

Joysticks

The 64 has two control ports, commonly called joystick ports, each with the standard nine-pin D-connector. Joysticks are the most popular external controller, and the 64 can use two. The four- or eight-direction control plus fire button is fine for games, but less suitable for complicated inputs like graphics and words. However, the 64 keyboard can be used along with joysticks, subject to a few restrictions. Commodore joysticks are interchangeable with those for Atari, Coleco, and several models of videogame machines.

Joysticks are based on a simple principle. The central stick is connected to electrical ground, so moving it makes contact with the sensors positioned up, down, left, or right. The button grounds another wire when pressed. So, the cable to the 64 contains six lines, one of them ground and the other five normally high but capable of being grounded. The 64 tests for one or more wires being grounded. Most joysticks are designed so that intermediate positions (northeast or up and right, and so on) ground two wires at once. Thus, the 64 may detect up to three wires of either joystick low simultaneously, counting the fire button. Some combinations aren't normally possible, of course, like north and south at the same time.

Internally, the most common arrangement is a grounded metal ring and pressure-sensitive, dimpled-metal switches which give way and make contact when the stick moves. Heavy-duty models have other arrangements; some even use microswitches. Some models have two fire buttons, and/or a button on top of the stick, so they can be used in either hand (converting some types for left-hand operation isn't hard). Joysticks tend to break down easily, often because the cable contains the thinnest possible strands of wire, which may break just inside the casing. To test a joystick, try it with one of my programs to verify that all eight directions can be located easily.

Programming Information

Two locations are relevant when programming the control ports for use with joysticks. Programming is easy if you know how to manipulate bits with AND and OR.

		Bit #: 7	6	5	4	3	2	1	0
JOY1 (front)	Keyboard Row: $DC01 (56321)				Fire	East	West	South	North
JOY2 (rear)	Keyboard Column: $DC00 (56320)				Fire	East	West	South	North

PEEKing locations 56320 and 56321 reads the joysticks. Nothing else is needed. Bits 0–4 are normally set to 1. The joystick's grounding action sets them to 0. The order of the registers is reversed from what might seem natural. Joystick 1 is read from a register with a higher address than joystick 2. It's easy to incorrectly assume 53260 is joystick 1.

Keyboard interference. The control ports are wired to the keyboard circuitry, which is an economical arrangement. Generally, the keyboard won't be used with a joystick. But if it is, there are a few side effects to watch for.

The keyboard scan routine sets the column (a bit in location 56320), then reads the row (location 56321); so joystick 1 causes spurious characters to appear on the screen occasionally because it mimics a row. In fact, the 1 key, back arrow, CTRL, 2 key, and space bar are interchangeable with S, N, W, E, and fire for joystick 1. Thus, if you press joystick 1 in the east direction, it outputs a 2; if you press it to the west, it slows screen scrolling, like the CTRL key does. Actually, locations 53261 and 145 ($91) can be interchanged in joystick programming, because 145 holds a copy of the default row, for testing the RUN/STOP key.

Joystick 2 doesn't generate spurious characters and is usually the better one to use with programs requiring only one joystick. But it does alter the keyscan. Pressing joystick 2 west while typing the X key has the same effect as SHIFT–RUN/STOP, for example.

Hardware. Looking at the 64, pins 1–5 are on top and 6–9 underneath, in left-to-right order. Joysticks use pins 1 (north), 2 (south), 3 (west), 4 (east), 6 (fire), and 8 (ground). It's easy to experiment with these ports, but be careful with pin 7, which carries 5 volts.

Using WAIT. WAIT loops until a bit or bits at some location change, so this command is useful for starting or restarting games. For example, WAIT 56320,16,16 waits until joystick 2's fire button is pressed, and WAIT 56321,16 waits until joystick 1's button is not pressed.

Program Examples

BASIC routines to handle joysticks tend to be long, which is hard to avoid since all possible directions must be separated out for processing. Program 16-1 demonstrates the method of combining bits into one value.

Program 16-1. BASIC Joystick Routine

```
1 REM RETURNS P=-41 TO +41
10000 PP=PEEK(56320)
10010 P=((PPAND4)=0) - ((PPAND8)=0) + 40*((PPAND1)
      =0) - 40*((PPAND2)=0)
10020 RETURN
```

Add a line 100015 PRINT P:GOTO 10000 to change the subroutine to a demonstration you can simply run. Program 16-1 checks only joystick 2 and doesn't check the fire button.

Using ML speeds joystick reading greatly, but the results often still need to be PEEKed by BASIC. The following ML routine reads both sticks; if either is active, ST is set to a nonzero value. Program 16-2 returns the joystick information in locations 2 and 3.

Program 16-2. ML Joystick Routine

```
1 REM ML JOYSTICK READER FOR THE 64
2 REM
3 REM USE SYS 828. ST<>Ø MEANS JOY PRESSED
4 REM PEEK(2) RETURNS 1,2,4,8,16 (OR MIX)
5 REM FOR N,S,W,E,FIRE OF JOY 1
6 REM PEEK(3) SAME FOR JOY 2
7 REM
1Ø FOR J=828 TO 848:READ X:POKE J,X:NEXT
2Ø DATA 173,Ø,22Ø,41,31,73,31,133,3,173
3Ø DATA 1,22Ø,73,255,133,2,5,3,133,144,96
4Ø SYS 828:PRINT ST PEEK(2) PEEK(3):GOTO 4Ø
```

Where all eight directions are needed, they can be combined together by a routine similar to the following, which uses the ML routine above. Delete line 40 in Program 16-2 and add the lines shown in Program 16-3. Replace the FIREBUTTON in line 1010 and the direction indicators in line 1020 with the line numbers of the routines in your program that process those joystick operations. *Program 16-3 will not run properly unless you replace these with valid line numbers.*

Program 16-3. ML Joystick Interpreter

```
1ØØØ SYS 828:IF ST=Ø GOTO 1ØØØ
1Ø1Ø IF (PEEK(3) AND 16)>Ø THEN FIREBUTTON
1Ø2Ø ON PEEK(3) GOTO N,S,,W,NW,SW,,E,NE,SE
```

Other machine language techniques include using interrupts to read joysticks and process the results. For example, it is possible to retain a previous value even when the joystick is back in the neutral position, or to allow optional keyboard or joystick operation.

Paddles and Other Analog Devices

Game paddles are less popular than joysticks, because, for many purposes the simple style of joystick movement is easier to use than the rotating knobs on the paddles. Commodore's paddles consist of two separate handheld units, each with a knob and fire button, which plug together into the same control port. Since the 64 has two game ports, two pairs of paddles could be used between four people, but this is relatively unusual. This discussion deals mainly with using one pair of paddles.

Counterclockwise rotation of the paddle knob increases the value read by the 64, and vice versa. It may be worth labeling the paddles since one (the X paddle) is read by SID chip location $D419 (54297), while the other (the Y paddle) is read at $D41A (54298). These registers are each eight bits wide, so there's maximum resolution of 1 in 256.

If both ports are used, POKEs into bits 6 and 7 of location 56320 ($DC00) select whether port 1 or port 2 is to be read. Obviously, something like this is necessary to enable both ports (four paddles) to be used, since the SID chip has only two analog-to-digital conversion registers.

Paddles are analog devices: they sweep through a continuous range of values. Differences between devices may cause slight compatibility problems because of this, unlike joysticks, where grounding is a simple on-or-off alternative. Pin 7 of each control port is connected to the 5-volt power supply. Pins 5 (Y) and 9 (X) are connected to this, via the paddles. Rotating the paddles alters the resistance of the internal potentiometers (variable resistors), each connected to the knob so that, as it turns, the resistance changes. The SID chip's POT X and POT Y registers measure the changing voltage levels produced by this changing resistance, performing an analog-to-digital (A/D) conversion which relates the voltage level (0–5 volts) to a number 0–255.

It's simple to use the same principle with other resistances. Commodore paddles are rated at 470K ohms (the K stands for *thousand*), and their minimum resistance is a few hundred ohms. They are approximately linear, changing in even steps of about 1000 ohms, so the overall range is large. To avoid damaging the SID chip while experimenting, keep minimum resistances of several hundred ohms between the 5-volt line and the POT inputs of the SID chip. Commodore documentation suggests hardware smoothing with a 1000pF capacitor between the POT inputs and ground, but this is already built in on the 64. Commodore paddles have a fire button on each unit, wired to pins 3 and 4 of the controller port for the X and Y paddles, respectively, which is connected to ground (pin 8) when pressed, just like joystick contacts for the west and east directions.

Atari paddles can also be used with a Commodore 64, but since the maximum resistance of the potentiometers in the Atari unit is about twice that of Commodore paddles—1M (1 million ohms) versus 470K ohms—the Atari paddles are very sensitive. A half turn of an Atari paddle is approximately equal to a full turn of a Commodore paddle.

Programming Information

Paddles can be read in BASIC or ML. The following locations are important:

		Bit #: 7	6	5	4	3	2	1	0
POT X	$D419 (54297)	All Bits							
POT Y	$D41A (54298)	All Bits							
Port 1 Buttons	$DC01 (56321)					Y	X		
Port 2 Buttons	$DC00 (56320)					Y	X		

Note that the same location reads the paddle potentiometer values irrespective of the port, but the fire button locations vary, and there's no hardware switch between them. Note also that the default is port 1, so if you want to use the simplest approach, use this port and simply PEEK the POT X and POT Y registers.

Program 16-4 selects and reads the paddle connected to port 2. Add a line 80 PRINT X,Y,FB to see the values on the screen.

Program 16-4. Reading Paddle 2

```
10 POKE 56333,127:REM IRQS OFF
20 POKE 56322,192:REM SET PINS 6,7 FOR OUTPUT
30 POKE 56320,128:REM SELECT PORT 2 (64=PORT 1)
40 X=PEEK(54297)
50 Y=PEEK(54298):FB=PEEK(56320)AND 12:REM USE 5632
   1 FOR PORT 1
60 POKE 56322,255:POKE 56333,129
100 RUN
```

The program selects port 2 in line 30. Interrupts have to be turned off to prevent the keyscan routine from altering values. After reading the registers, line 60 restores normal keyboard operation. If you don't need the keyboard, line 60 can be omitted.

Paddle buttons generate spurious characters when connected to port 1. Pressing the button on paddle X is equivalent to pressing the CTRL key, which slows processing. In port 2, the keyscan is altered (try X with fire button). As noted, port 2 cannot be read at the same time as the keyboard because of the conflict over the use of $DC00. This means that, with paddles in port 1, PEEKing values is easy, but the fire buttons will have to be avoided in some circumstances.

ML programming is fast enough to disable the keyboard without noticeable effect. Program 16-5 allows four paddles and four buttons to be read with virtually no problems. The fire buttons register in the status variable ST, so IF ST>0 is a simple test for a button press.

Program 16-5. ML Paddle Reader

```
10 DATA 120,162,2,169,192,141,2,220,41,128,141,0
11 DATA 220,160,208,136,208,253,173,25,212,149,2
12 DATA 173,26,212,149,3,169,64,202,202,16,232,173
13 DATA 1,220,74,74,9,252,133,144,173,0,220,9,243
14 DATA 37,144,73,255,133,144,169,255,141,2,220,96
15 FOR J=49152 TO 49211:READ X:POKE J,X:NEXT
```

To activate the routine, use SYS 49152. The results will be left in the following locations:

	Paddle X	Paddle Y	Fire X	Fire Y
Port 1	PEEK(2)	PEEK(3)	ST Bit 0 Set	ST Bit 1 Set
Port 2	PEEK(4)	PEEK(5)	ST Bit 2 Set	ST Bit 3 Set

The POT registers are usually updated about every 500 processor cycles. Extra time is necessary when the ports are shifted, so in ML it's best to allow a loop like LDX #$D0 : LOOP DEX : BNE LOOP before reading from the SID paddle registers.

Accuracy. The 64's A/D conversion is better than the VIC-20's. There's less crosstalk and no need to correct for the other paddle's value. With paddles, resolution is limited, and you may find intermediate values can't be read. Occasionally, values returned are actually very different from other recent values. Generally, don't try to aim for accuracy beyond the eight-bit limit imposed by the SID chip. It's possible to smooth values, but use a moving average method, taking a running total at regular intervals. With devices other than paddles, smoothing isn't suitable: graphics pads, where a stylus can jump from point to point at will, need a good estimate of each point's coordinate, not an average value between unrelated points.

Other Analog Devices

Other devices that make use of variable resistances may be interfaced to the 64 through the paddle inputs. One of the most common is the graphics tablet. Contact of a stylus on a carbon film pad reduces resistance in both the X and Y direction. Well-designed graphics tablets should return X and Y values which reflect the position of the stylus on the pad accurately. Programming is identical to that for paddles. Generally, these are used to draw on the screen or to select from a menu by positioning a cursor at some option and pressing a button.

Light Pens

A light pen is a pen-shaped device, fitted with a cable, which plugs into control port 1. The line carrying the light pen's signal is tied to the keyboard, so keys B, C, M, Z, f1, the left SHIFT key, and period won't work while a light pen is plugged in, unless the light pen has a switch on it. Watch for this if you want to input from the keyboard and use a light pen simultaneously.

The internal electronics include a light-sensitive component, usually a phototransistor, which allows current to pass only when exposed to light. Light pens use the 5-volt and ground lines, plus a line into the VIC-II chip. When fairly intense light—such as the electron beam that creates the video display in a television or monitor—is detected, this line is grounded and two VIC chip registers are frozen, or *latched*, and remain unaltered until the next exposure to light. The two registers hold the horizontal and vertical positions of the pen inferred from the distance of the electron beam from its starting position.

Whenever a range of alternatives is to be selected from a screen display, a light pen may be useful. Selecting alternative answers to multiple-choice questions and selecting options from a menu are examples. Also, games like chess can make good use of light pens. Numbers can be input with a numeric 0–9 pad on the screen, and a light pen can help sketch on a screen.

The drawbacks to using light pen input are the limited accuracy of the pens, computer limitations, and the fact that more people own joysticks than light pens. The glass in front of the TV tube and general lack of precision make accuracy and repeatability not as good as with analog devices like graphics tablets. Another difficulty is that some colors, such as black, won't trigger the pen.

The light pen programming registers are read-only (they cannot be POKEd): $D013 (53267) is the horizontal position register and $D014 (53268) is the vertical position register. The following one-line program displays both registers:

```
10 PRINT PEEK(53267): PRINT PEEK(53268): PRINT "{CLR}": GOTO 10
```

As you move the pen across the screen, the first value varies, increasing as you move the pen from left to right. As you move it down, the second reading increases. The readings are taken from the VIC chip as it monitors and controls the TV. Values across vary from about 30 to 190, ignoring the border area, and values down range from about 50 to 250 (on U.S. televisions). The monitor scans each picture twice, interlacing the pictures, which is why there are only 200 separately distinguished raster lines. To convert these ranges into 0–39 and 0–24 for column and row equivalents is easy. Just subract 30, then divide by 4, to get the horizontal position, and subtract 50 and divide by 8 for the vertical. Resolution is in principle two dots horizontally and one dot vertically.

Program 16-6. BASIC Light Pen Program

```
20 X=PEEK(53267)
30 Y=PEEK(53268)
40 X=(X-30)/160*40
50 Y=(Y-50)/200*25
60 PRINT "{HOME}":FOR J=1 TO X:PRINT "{RIGHT}";:NE
   XT
70 FOR J=1 TO Y:PRINT "{DOWN}";:NEXT:PRINT "Q";:GO
   TO 20
```

This simple program will draw a small ball on the screen when it detects a light pen reading. You'll probably find your pen's readings show quite a bit of jitter, even when held still, removing any chance of serious high-resolution work.

Program 16-7 uses an ML subroutine to read the pen and convert its readings to a screen position. It POKEs a character into the screen, then loops back for more. As you'll see, this is much faster than BASIC. Color change and character change demonstrations are included, and you can modify these to suit your requirements.

Program 16-7. ML Light Pen Draw

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
2 REM SYS 49152 READS, PLOTS UNTIL KEYPRESS:rem 73
3 REM DEMO SETS WHITE BALL; USES KEYS      :rem 206
4 REM 1-8 AND SHIFT 1-8 TO CHANGE COLOR     :rem 93
5 REM & F7 TO SWITCH TO PLOTTING BLOCKS      :rem 3
6 REM (CHAR IS IN 49215, COLOR IN 49230)     :rem 5
10 FOR J=49152 TO 49236:READ X:POKEJ,X:NEXT
                                           :rem 221
20 PRINT "{CLR}"                           :rem 198
100 SYS 49152                              :rem 149
110 A=PEEK(780):REM PEEK (197) ALSO OK      :rem 35
120 A$=CHR$(A)                             :rem 200
130 IF CHR$(A)>="1" AND CHR$(A)<="8" THEN POKE 492
    30,A-49                                :rem 104
140 IF CHR$(A)>="!" AND CHR$(A)<="(" THEN POKE 492
    30,A-25                                 :rem 67
150 IF CHR$(A)="{F7}" THEN POKE 49215,160 :rem 231
160 GOTO 100                                :rem 97
500 DATA 32,228,255,240,1,96,173,19,208,56,233,32
                                           :rem 109
501 DATA 144,242,201,160,176,238,170,173,20,208,23
    3                                      :rem 197
```

```
502 DATA 48,144,230,201,200,176,226,41,248,168,169
                                              :rem 162
503 DATA 0,133,3,132,2,6,2,38,3,6,2,38,3,152,101
                                              :rem 18
504 DATA 2,133,2,169,4,101,3,133,3,138,74,74,168
                                              :rem 50
505 DATA 169,81,145,2,165,2,133,4,165,3,24,105,212
                                              :rem 146
506 DATA 133,5,169,1,145,4,208,173,240,171 :rem 16
```

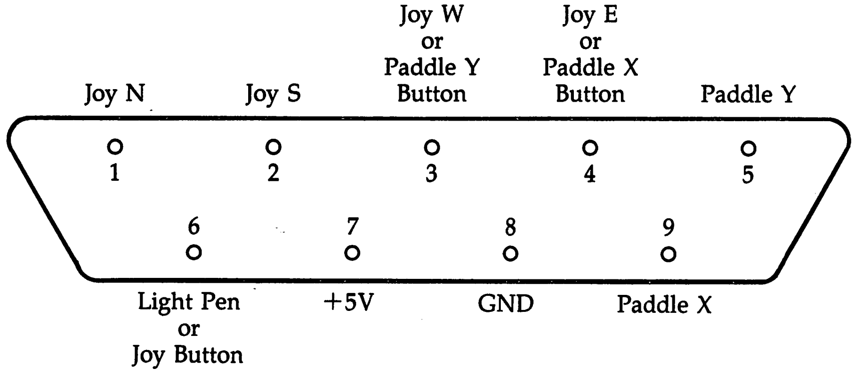
The ML checks to see that the light pen values are in range. If so, it calculates the position of the screen character, and plots. You may prefer to read the screen and detect a character as the pen points to it. Modify Program 16-7 as follows. Replace each of the first six numbers in line 500 with 234, then replace line 505 with 505 DATA 169,81,177,2,96, and delete line 506. RUN and PRINT J to find the correct upper limit to use after TO in the FOR-NEXT loop in line 10, and replace line 100 with 100 SYS 49152: PRINT "{HOME}" PEEK(780): GOTO 100. Now the screen PEEK value (32 for space, 1 for A, 2 for B, etc.) of the character selected by the light pen appears in the top left. The program's calculations are unaltered, but the screen is no longer POKEd. Values 32, 160, 48, and 200 in the DATA check the limits, subtracting 32 and checking to see that the result doesn't exceed 160 horizontally; these can be fine-tuned.

Light pens and other devices which ground the appropriate line can be programmed to generate interrupts (the end of Chapter 12 explains how). This has the advantage of signaling every activation of the device, but otherwise isn't very useful.

The Control Port Socket

Figure 16-1 shows the assignments of the nine pins of the control port sockets of the 64. The two ports are identical, except that the light pen input (shown at pin 6) is available only for port 1. Remember that control port devices will affect each other just as they affect the keyboard. A light pen program may not work if paddles are connected, for example.

Figure 16-1. Typical Control Port Pinout



Chapter 17

Major Peripherals

- Printers
- Plotters
- Modems
- The RS-232 Interface
- The Serial Port

Chapter 17

Major Peripherals

This chapter covers printers, plotters, modems, and the 64's interfaces. Simple program examples are included for quick reference.

Printers

Simple Commands

Commodore printers designed for the 64 plug into the serial port, the round port at the back of the 64 next to the video output, or at the back of the disk unit when daisychaining. At the simplest level, printers are controlled with OPEN 4,4 (which opens file number 4 to the printer), PRINT#4,"HELLO" (which prints a message to file 4), and CLOSE 4 (which closes the file). Any number of PRINT#4 statements can be issued. PRINT statements can still be used to send output to the screen.

The 64 has no LIST#4 statement. To LIST programs to the printer, use OPEN 4,4: CMD 4: LIST (which opens file 4, directs output to that file instead of to the screen, and lists). Follow this with PRINT#4: CLOSE 4 (which disengages CMD and closes the file). The PRINT#4 is needed to close the file properly, so get in the habit of using it before CLOSE whenever you use CMD.

With machine language monitors, all output can be sent to a printer with OPEN 4,4: CMD 4: followed by a SYS to the entry point of the monitor. Then enter M 1000 1200, for example, and output for the desired memory dump will be made to the printer. The commands may have to be typed in blind, since they may not echo to the screen, but this isn't a big problem. Enter X or E to exit the monitor, then PRINT#4: CLOSE 4 to redirect output to the screen.

Non-Commodore Printers

For most applications, many non-Commodore printers use commands identical to those for Commodore printers. Also, many interfaces are available which emulate Commodore printer features in addition to allowing you to use the special features of the non-Commodore printer. Printers which use the RS-232 port at the back left of the 64, usually with an RS-232 converter cartridge, use device number 2, instead of 4. To use such a device, the following sequence is typical:

```
OPEN 2,2,0,CHR$(6):PRINT#2,"HELLO"
```

This opens file 2 with baud rate 300, then sends HELLO to the device. PRINT#2: CLOSE 2 closes the file. OPEN 2,2,0: 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 the 64.

Easy Printing

PRINT# statements are similar to ordinary PRINT statements, but there is a distinction between the carriage-return character, CHR$(13), and the linefeed character, CHR$(10), which advances the paper in the printer. Commodore 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 marker 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-Commodore printer over-prints lines on top of each other, use a file number of 128 or greater (OPEN 128,4, for example, with PRINT#128,"HELLO") to cause the 64 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 likely to contain a number of PRINT# statements which are meaningful only with reference to the printer in use.

Choosing and Using Printers

A printer is simply a device to convert a stream of bytes into text or graphics. Unlike other Commodore devices, non-Commodore printers can often substitute for Commodore equipment. This is worthwhile where special faster or higher quality print is required (it is best to *see the product in action*), or where a user needs to be able to print in foreign languages. In all these cases, some sort of interface will be necessary, because neither of the 64's printer ports is standard.

Commodore printers for the 64 include the 1515, the 1525, the MPS-801 (which is quite similar to the 1525 except that it uses cartridge ribbon instead of a reel), and the more versatile 1526. Their features are summarized in Table 17-1. The 1515, 1525, and MPS-801 are designed to be compatible with both the VIC and the 64; the 1526 is designed specifically for the 64 and is not completely compatible with the VIC. They are made by Seikosha. (Commodore doesn't make any of its own printers.)

Table 17-1. Selected Commodore Printers for the 64

	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, although none prints characters that are *identical* to the 64's characters. The 1515's reverse characters, for example, lack the solid underline of the screen characters. In addition, lowercase letters like *g* lack true descenders, so they seem to float up on the line. 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 by Commodore without warning, so there's no guarantee that one model won't differ from others. Commodore's printers haven't consistently used identical commands in the past, either. These commands are discussed further in the next section.

Each printer allows 80 normal-width characters to the line (except that the 1515 uses a smaller typeface and nonstandard 8-inch-wide paper). It is possible to use 8-1/2-inch paper on the 1515 by loosening the paper guide, removing the lid and the guide, and taking out the bar so that 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. Usually, a total of 66 lines, including linefeeds, has to be arranged per page if neat output is desired. Six lines per inch is standard.

In addition to the standard Commodore 64 graphics characters, the 1515, 1525, and MPS-801 printers have a graphic mode in which individual columns of dots can be programmed. This is demonstrated in the manuals by reproducing the Commodore symbol. A page of graphics can be printed continually redefining the dot pattern. This makes it easy to print high-resolution graphics. The 1526 lacks a graphic mode, but a similar effect can be achieved using this printer's single user-defined character. However, only one redefined character is allowed per line printed, so multiple overprints must be made to reproduce a complete line of graphics. Thus, the 1526 is less suitable for high-resolution graphics printing than the other models.

Most software assumes device 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 Commodore printers have a self-testing facility, a loop in internal ROM which outputs the character set (except for reverse characters, which may cause overheating if used excessively). They also have a power-on sequence. The older 1515 can 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 Commodore printers include a series of printers for the earlier PET/CBM machines. All PET/CBM printers require an IEEE interface connected to the 64's normal printer port to operate. The 4022 is the main PET/CBM printer; it has 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 a Centronics interface, which is a parallel interface using multiwire flat-ribbon cable. RS-232 serial interfaces are also common. IEEE interfaces are rarer, and current loop interfaces are another relatively uncommon type. All of these can be connected to the 64, with the proper interfacing.

It is always advisable to test the compatibility of equipment before purchasing, particularly if packaged software is to be used. Most good word processors make allowances for printer type, but other programs may not work correctly with all printers, particularly with features like margin and tab.

Printer Types

Several different kinds of printers are now available for the 64. They are described below.

Teletypes. These are old-fashioned terminals, uppercase only, which communicate with computers via RS-232. Since they have been replaced by video terminals in industry, they can sometimes be found very cheaply. Of course, they may cost you more in the long run, and are severely limited in their capability.

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. Ball 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 inexpensive, but the paper they use is relatively costly and generally not the best quality.

Dot-matrix printers. These are by far the most widely used computer printers. The print head typically has 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 are 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 international alphabets, as well as the ability to print graphics. Some dot-matrix printers have double strike, emphasized, and correspondence-quality modes, and are able to print in several type widths.

Daisywheel printers. A daisywheel has approximately 100 radial spokes, each of which holds a character at the tip. The wheels have low rotational inertia so they can be spun rapidly, and common letters are clustered together to reduce search time. A solenoid drives the letter against ribbon and paper. Commonly used spokes will eventually wear and the wheel will need replacing. Wheels and ribbons aren't standardized to any extent. These printers are usually more expensive than dot-matrix units and are often slower, but the print quality is very good. Some daisywheel printers offer double strike, proportional, and shadow printing.

General Remarks

Printers normally use continuous fanfold paper. Letterhead stationery is available in 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. *Friction feed* indicates that rolls or sheets of unperforated paper 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. Some of these printers use standard typewriter ribbons (like the popular Gemini printers), which makes the ribbon less costly to replace.

External switches can range from simple paper control (linefeed, formfeed, and online buttons) up to complete control over baud rate, parity, horizontal and vertical spacing, and so on. Some printers, like Epson's RX-80, 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 of which is a very satisfactory description. A lot depends on the density of the text to be output. Moreover, the figures quoted are often inaccurate. *As usual, it is best to test the printer in the conditions you plan to use it before you buy.*

Commodore 64 compatibility is difficult with regard to graphics and upper/lowercase switching. Few printers offer the entire 64 character set, and interfaces may not handle the 64 upper/lowercase switch. However, in some cases, the interfaces themselves are programmable to allow for this or contain their own ROM character definitions for the Commodore graphics and reverse-field characters.

Programming for Printers

The following discussion is not intended to replace printer manuals, since 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 G shows the conventional meanings of codes 0–31, most of which are more relevant to Teletypes than to printers. The ESCape code, CHR$(27), is widely used with non-Commodore printers. Anything following ESCape is treated by a special routine independent of the rest and can be used to set the features of the printer. In that respect, it works much like channel 15 of Commodore disk drives. Commodore printers could have used this method rather than secondary addresses.

Some 64 control characters, like {CLR} and {DEL}, mean nothing to Commodore printers and may cause them to hang up. A number of the ASCII control characters are irrelevant. The special characters controlling 64 printers therefore are chosen from those characters. The characters and their functions are given in Table 17-2.

Table 17-2. Common 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 Commodore 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 have varied as well: the 1515 uses OPEN 4,4,7 to set lowercase mode; earlier models used this for uppercase.

To avoid such problems, you should let the 64 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, since they have almost identical effects. For example, after OPEN 4,4, the commands PRINT#4,"HELLO" and CMD4: PRINT "HELLO" each print HELLO to file 4. The difference is that CMD leaves the printer in a listening mode, so future PRINT statements are output to the printer.

However, CMD isn't really implemented properly. Although it works well with LIST (OPEN 4,4: CMD4: LIST), if CMD 4 is followed by a program with PRINT statements, it isn't reliable. GET, for example, makes the printing revert to the screen. Thus, it's usually best to use PRINT#. If you wish to divert some output to the screen, use something like the routine shown below.

```
10 PRINT "OUTPUT TO PRINTER OR SCREEN (P/S)";: INPUT X$
20 IF X$="P" THEN D=4
30 IF X$ ="S" THEN D=3 :REM SCREEN IS DEVICE 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 require.

As noted above, you should use PRINT#4 after CMD4 to "unlisten" the printer, and return everything to normal, followed by CLOSE4. 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 Lowercase

CBM printers don't generally behave like VIC and 64 printers, since they remain in either uppercase or lowercase mode until changed. The CBM models 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—use 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. This 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 (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

The 1525 and MPS-801 use CHR$(8) to enter graphic mode, in which the dot pattern of the printed character can be defined. Since these printers form characters in a 6 × 7 matrix, six columns of seven dots have to be defined. It's also necessary to add 128 to the value for each column. All that's needed is PRINT#4, CHR$(8) followed by the bytes which define the columns. You can use CHR$ to define the bytes for the column values—for example, PRINT#4,CHR$(8) CHR$(150) CHR$(182) CHR$(224) CHR$(224) CHR$(182) CHR$(150). You can also use PRINT#4,X$ where X$ is built from values in a DATA statement, starting with 8. Remember to PRINT#4, CHR$(15) after graphic printing to return the printer to normal text mode.

The 1526 has a single definable character, CHR$(254), specified as eight columns of eight dots by opening a file to the printer with a secondary address of 5. Thus the 1526 equivalent for the example above would be OPEN 5,4,5: PRINT#5, CHR$(0) CHR$(22) CHR$(54) CHR$(96) CHR$(96) CHR$(54) CHR$(22) CHR$(0): CLOSE 5 to define the character, followed by OPEN 4,4:PRINT#4,CHR$(254): CLOSE 4 to print it.

An interesting use for this definable character capability is to dump a high-resolution screen to the printer. Multicolor mode is not as easy, since the printer can't distinguish the four colors. Program 17-1 slowly dumps a bitmap screen starting at 8192 ($2000). It will not work with all printers, due to differences in the printer commands and features. (It cannot be used with the 1526 printer.)

Program 17-1. Graphics Screen Dump

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
100 BA=8192:REM BASE ADDRESS MAY DIFFER    :rem 251
105 OPEN 4,4                                :rem 92
110 FOR J=0 TO 7:P(J)=2↑J:NEXT             :rem 191
115 FOR X=0 TO 319 STEP 7                  :rem 246
120 X$=""                                  :rem 144
125 FOR Y=199 TO 0 STEP-1:V=0               :rem 34
130 FOR BT=0 TO 6                           :rem 87
140 X1=X+BT:X2=X1 AND 7:IF BT AND X2 THEN 150
                                           :rem 137
145 BY=BA + (Y AND 248)*40 + (X1 AND 504) + (Y AND
     7)                                     :rem 25
150 V=V + P(BT)*SGN(PEEK(BY) AND P(7-X2)) :rem 183
155 NEXT                                   :rem 218
160 PRINT V;                               :rem 181
165 X$=X$ + CHR$(V+128)                    :rem 106
170 NEXT                                   :rem 215
175 PRINT#4,CHR$(8)X$                      :rem 179
180 NEXT:CLOSE4                            :rem 188
```

Printing a copy of a normal text screen is considerably less complicated than reproducing graphics. Program 17-2 is an ML screen dump which works with most printers, as long as the screen display uses only the ordinary ASCII characters with no 64 graphics. It includes tests for the screen start position, and for lower- or upper-case mode. Use OPEN 4,4: CMD 4: SYS 830: PRINT#4: CLOSE 4 to activate the routine.

Program 17-2. ML Character Screen Dump

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
1 REM 64 SCREEN DUMP                           :rem 117
2 REM USE OPEN 4,4:CMD4:SYS 83Ø:PRINT#4:CLOSE4
                                               :rem 225
3 REM ALLOWS FOR U/L CASE & SCREEN POSN        :rem 226
4 REM                                           :rem 24
5 FOR J=83Ø TO 932:READ X:POKE J,X:NEXT        :rem 221
1Ø DATA 169,Ø,133,3,133,4,133,5,173,136,2,133,6,16
   9                                           :rem 188
11 DATA 64,133,2,23Ø,3,165,3,2Ø1,41,2Ø8,15,23Ø,4
                                                :rem 23
12 DATA 165,4,2Ø1,24,24Ø,67,169,1,133,3,32,215,17Ø
                                               :rem 137
13 DATA 162,Ø,161,5,41,127,36,2,2Ø8,6,36,129,24Ø
                                                :rem 41
14 DATA 19,2Ø8,33,36,129,2Ø8,9,72,169,2,44,24,2Ø8
                                               :rem 116
15 DATA 2Ø8,13,1Ø4,169,35,2Ø8,16,72,169,2,44,24,2Ø
   8                                           :rem 2Ø7
16 DATA 2Ø8,5,1Ø4,9,64,2Ø8,3,1Ø4,9,96,32,21Ø,255
                                                :rem 54
17 DATA 23Ø,5,2Ø8,177,23Ø,6,2Ø8,173,76,215,17Ø
                                               :rem 218
```

The above routine substitutes the dummy character # for any graphics found in the screen display. Alter the 35 in line 15 to change the dummy character to some other symbol.

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 (X, which must be in the range 1–255), specifies character definition (255 gives a solid column of dots), and returns to normal graphics.

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 64 is left free to run programs normally, except that accessing the serial bus is prohibited. In principle this seems easy, since the disk can talk and the printer can listen, but there is no simple way to accomplish it. Some printers and some interfaces include a printer buffer, which accepts data from the computer and holds it until the printer can process it. Once all the data has been sent to the buffer, the computer returns to other operations.

Plotters

Plotters are most commonly used commercially for technical drawings. Plotters have two stepper motors controlling pen or paper movement (or both) vertically and horizontally, with a mechanism to lift the pen off the paper. Typically, eight directions of motion 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 inexpensive plotters is roughly three inches per second, so be prepared for long delays, particularly if the interface is slow and if commands are sent with BASIC.

Commodore's 1520 plotter uses 4-1/2-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 vertical 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, and can also draw geometrical patterns. Yellow, magenta, and cyan pens could give an imitation of color-separation printing.

Lines

Program 17-3 is a plotter drawing 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

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
100 M=YD/XD:YP=0:REM M IS THE GRADIENT. Y-POSITION
     STARTS 0.                              :rem 113
110 FOR XP=1 TO XD: PRINT#N,EAST: REM EXACT FORM V
    ARIES WITH PLOTTER                      :rem 86
120 IF M*XP>YP THEN PRINT#N,NORTHEAST: YP=YP+1:XP=
    XP+1:IF XP<XD GOTO 120                  :rem 150
```

```
130 NEXT                                          :rem 211
140 IF YP-1<YD THEN PRINT#N,NORTHEAST: YP=YP+1:GOT
    O 140:REM FINISH                               :rem 33
```

Circles

There are several methods to plot circles. One useful circle plotting routine is given in Program 17-4.

Program 17-4. Circle Plotter

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
500 REM Q=DEGREES SUBTENDED BY EACH STRAIGHT LINE
    {SPACE}SEGMENT.                                :rem 31
505 REM Q=10 PLOTS A 36-SIDED FIGURE                :rem 1
510 G=R:H=0: REM R=RADIUS. G AND H ARE INTERMEDIAT
    E VALUES                                      :rem 131
520 N=360/Q: REM N=NUMBER OF SIDES=NUMBER OF REPET
    ITIONS OF LOOP                                :rem 122
530 F=COS(Q*↑/180): I=SIN(Q*↑/180):REM TRIG PARAME
    TERS                                           :rem 24
540 FOR J=0 TO N                                   :rem 40
550 C=G*F-H*I:A=G*I+H*F: REM THESE ARE THE X AND Y
     COORDINATES OF THE NEXT PT.                  :rem 164
560 REM DRAW THE STRAIGHT LINE SEGMENT TO THE POIN
    T X=C,Y=A                                      :rem 24
570 G=C:H=A                                        :rem 99
580 NEXT J                                         :rem 38
```

Program 17-5 demonstrates the Commodore 1520 plotter's ability to draw graphs.

Program 17-5. Plotter Demo

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
10 OPEN 4,6                                        :rem 41
20 OPEN 6,6,1                                     :rem 137
30 PRINT#6,"H"                                      :rem 1
40 FOR X=20 TO 460 STEP 2                         :rem 237
50 Y=SIN((X-20)*4*↑/440)                          :rem 188
60 Y=Y+(1/3)*SIN((X-20)*4*3*↑/440)                :rem 172
70 Y=Y+(1/5)*SIN((X-20)*4*5*↑/440)                :rem 177
80 PRINT#6,"D",X,Y*70                             :rem 156
90 NEXT                                           :rem 168
100 PRINT#6,"M",479,0                              :rem 96
110 PRINT#6,"D",0,0                               :rem 228
120 PRINT#6,"D",0,110                              :rem 71
130 PRINT#6,"M",0,0                               :rem 239
140 PRINT#6,"R",0,-100                            :rem 131
150 PRINT#4,"WAVEFORM WITH 2 ODD HARMONICS ADDED"
                                                  :rem 140
160 PRINT#4:CLOSE 4:PRINT#6:CLOSE 6               :rem 101
```

Modems

Most 64 modem users have either the model 1600 VICModem or the 1650 Automodem. Both are designed for the U.S. phone system. The 1650 plugs into the base of the phone, while VICModem requires that you use a phone with a modular plug handset in order to connect correctly.

When the 64 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 64 is always set to originate while talking to the system.

To use a modem, first connect the modem to the computer with the 64 turned off. Plug the Commodore modem into the user port (and connect it to the phone, if it is a direct-connect model like the 1650), 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, tape, or disk. For the VICModem, you may use *64 Term* and for the Automodem, use *Term 64*, provided with the modem package.

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, dial the number (either by dialing yourself or by inputting the number into the 64 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.

Now, wait for the carrier-detected indicator (red light) to come on. Your software may print something like 64 CONNECTED. Either signifies that your modem has recognized the incoming frequency. Again, the actual procedure varies between modems; it's automatic with the Automodem, but acoustic modems require you to put the handset into the cups of the modem and switch to data (online) mode.

Wait for the system's first welcome frame to appear. Public systems often ask for a password. This allows them to charge for access to the system. 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.

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 must 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 that it can reliably detect individual bits.

Certain technical parameters are also important. Phone companies maintain control over certain technical details of their lines. They do not permit excessive voltages to get to exchanges, and some tones and signals are reserved for diagnostic use. Such standards vary internationally. As a result, modems in different countries are likely 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 international compatibility problems.

The most common American modem convention is the Bell 103, which is used by Commodore modems and by many computers. However, this is slow, as characters are generally transmitted at 300 bits per second. 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. Still, this is faster than many people can read or talk.

Bell 103 uses a system called frequency shift keying. It means that an *on* bit is transmitted with one frequency tone, while an *off* bit is transmitted with a tone of another frequency. The tone of the signal carries information. In order that both ends of the line can talk, Bell 103 uses four tones altogether, which means 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 frequencies in originate mode are 1270 Hz to transmit a mark signal, 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 an abbreviation of the words *modulator* and *demodulator.*

At 300 baud (or, more properly, 300 bits per second), the 64'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 64'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 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 sum of the individual bit values 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 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 okay. OPENing an RS-232 file allows these variables to be controlled by the programmer, within limits.

Note that since Commodore 64 characters use eight bits, 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.

Converting bytes into bits and sending them, and the converse process of assembling bits into bytes, can be performed in software (as the 64's RS-232 handling does) or by chips like the UART (Universal Asynchronous Receiver-Transmitter—*asynchronous* 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 conforming to the check, and such events constitute undetected errors. 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 echo-plexing feature, a verification system which returns characters when they're received. Thus, if characters appear double, use this switch or software which verifies the echoed characters. True half duplex requires a line like RS-232's secondary channel to be able to interrupt unwanted messages.

Smart terminal software can download programs (load 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 Bell 202 and 212A, which are faster than the 103 standard. The 212A can work with the 103, but the 202 and 212A are currently less popular than the 103, mainly because the faster modems are significantly more expensive. Incidentally, the 103 system can operate at 600 bits per second, which may be worth trying.

One problem with acoustic modems may be getting the two cups which are supposed to fit the handset into place. A few modems forgo the rigid body in favor of a pair of cups on leads, so they can fit many phone styles. Incidentally, over short distances it's not necessary to use a modem—two VICs or 64s can be connected by three lines between their user ports, or with RS-232 adapters. *As always, leave this work to someone with sufficient technical experience.*

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 64 must be allowed for. Both are fairly straightforward, though they may appear difficult.

Opening an RS-232 file. Only one RS-232 file can be opened at any one time, and the syntax is typically OPEN 2,2,0,CHR$(6). The device number must be 2, so 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. The value shown assigns 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 duplex. Three-line handshaking is assumed.

This is the most common combination. Use OPEN 2,2,0,CHR$(38)CHR$(96) to assign a seven-bit word and even parity instead.

Transmitting and receiving characters. All that's needed is a loop to get characters from the keyboard and send them through the modem using PRINT#2, and to get characters from the modem (using GET#2). BASIC may need delay loops in its output to avoid sending characters too fast. For most purposes, some characters have to be converted, and BASIC provides an adaptable and quite easy means to do this. One reason for the conversion is that 64 ASCII is slightly different from true ASCII, so unless you're happy with strange-looking lettering, conversion is necessary. The other reason is that it's useful to define some keys so that they perform modem-specific functions.

Program 17-6 is a good example of a program for use with a modem.

Program 17-6. 64 Terminal Program

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
100 OPEN 2,2,0,CHR$(6):REM OPENS FILE 2; 300 BAUD,
     8 BIT, NO PARITY                          :rem 39
101 REM OPEN 2,2,0,CHR$(38)+CHR$(96) FOR ASCII 7 B
    ITS + EVEN PARITY                          :rem 84
200 DIM F%(255),T%(255)                        :rem 86
210 FOR J=32 TO 64:T%(J)=J:NEXT               :rem 193
220 FOR J=65 TO 90:T%(J)=J+32:NEXT:REM LOWER-CASE
                                               :rem 71
230 FOR J=91 TO 95:T%(J)=J:NEXT               :rem 204
240 FOR J=193 TO 218:T%(J)=J-128:NEXT:REM VIC UPPE
    R-CASE                                    :rem 202
250 T%(133)=27:T%(134)=127:T%(135)=3:T%(136)=17
                                               :rem 97
251 REM THESE ARE TRUE ASCII:I.E. ESC,DEL,CTRL-C B
    REAK,CTRL-Q                               :rem 139
260 T%(137)=17:T%(138)=144                     :rem 24
300 FOR J=0 TO 255                            :rem 112
310 IF T%(J)>0 THEN F%(T%(J))=J                :rem 43
320 NEXT                                      :rem 212
400 PRINT CHR$(147) CHR$(14):REM CLEAR; LOWER-CASE
                                               :rem 88
```

```
500 IF PEEK(669)<PEEK(670) THEN 500          :rem 82
501 REM IF (PEEK(37151) AND 64)=64 THEN 501
                                              :rem 122
510 GET OUT$:IF OUT$>"" THEN PRINT#2,CHR$(T%(ASC(O
    UT$)));: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, while line 101 shows an alternative OPEN statement. Lines 200–260 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 simply be looked up. Integer arrays save space. Lines 300–320 convert the *from* array into the inverse of the *to* array. Line 500 verifies that output data has actually been sent.

The status byte, ST, can also be tested. 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. Remember that the 64 keyboard allows control characters to be typed without special programming.

ML conversion to true ASCII. ML programmers may need this routine, which converts a 64 ASCII character into true ASCII. It interchanges upper- with lowercase:

```
      CMP  #$41
      BCC  END
      CMP  #$5B
      BCS  LABEL
      ORA  #$20
      BCC  END
LABEL CMP  #$61
      BCC  END
      CMP  #$7B
      BCS  END
      AND  #$DF
END   continue...
```

The RS-232 Interface

RS-232-C is a communications standard established by the Electronic Industries Association. Its voltage convention is as follows: A negative voltage indicates a *mark* (bit value 1), while a positive voltage indicates a *space* (bit value 0). The standard also establishes a 25-pin connector to be used in RS-232 equipment. (This connector is not used by the 64, which provides RS-232 communication through the user port.) Pin numbering is from 1 to 13 (top) and 14 to 25 (bottom). It is sometimes helpful to know the standard functions of the pins, 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

In the 64's RS-232 system, an OPEN to the RS-232 device 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 routine (at $F409 in the 64) 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: OPEN to RS-232 lowers the top of memory by 512 bytes, making room for two 256-byte FIFO (first-in, first-out) buffers. BASIC pointers are altered to clear variables (equivalent to a CLR statement), so it's best to OPEN the RS-232 file early in the program to avoid losing variable values. The baud rate is controlled by reference to tables in ROM, which in 64 has ten usable values.

ML programmers may want to alter the NMI vector into RAM so that the tables can be changed. Alter the first tabled value to generate the new baud rate. After OPEN, remember 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 Channel

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. As stated earlier, 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.

Figure 17-1. The Control Register

128	64 32		8 4 2 1
Number of Stop Bits	Word Length (Excluding Parity)	Unused	Baud Rate Control
0=single 1=two	00=8 bits 01=7 bits (e.g., ASCII) 10=6 bits 11=5 bits (e.g., Baudot)		0001= 50 0010= 75 0011=110 0100=134.5 0101=150 0110=300 0111=600 1000=1200 1001=1800 1010=2400

Figure 17-2. The Command Register

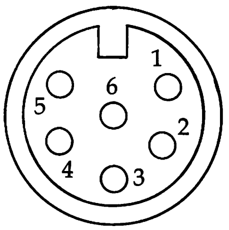
128 64	32	16		1
Parity Type	Parity Bit/ No Parity Bit	Duplex	Unused	Handshake Type
00=Odd Parity 01=Even Parity 10=Mark Bit 11=Space Bit	0=None 1=Parity Bit	0=Full 1=Half		0=3-Line 1=X-Line

Finally, six bits of location 663 ($297) report conditions resulting from RS-232 use. If bit 0 is set, there is a parity-bit error. Bit 1 being set indicates an error in the structure of received bits, perhaps due to noise. Bit 2 is set when the receiver buffer is full (that is, when data is coming in too fast). Bit 3 is set to indicate when the receiver buffer is empty. Bit 4 is set when the clear-to-send signal is off (when the remote terminal is not ready to receive). Bit 5 is unused, and bit 6 is set when the data-set-ready signal is missing (the remote terminal is not ready to send). When bit 7 is set, a break has been detected.

The Serial Port

The 64's serial port, adapted from the IEEE-488 standard interface of PET/CBM machines, is peculiar to Commodore. The 64 uses a simplified, nonstandard version, which carries serial (as opposed to parallel) data and is comparatively slow. Figure 17-3 shows the port's six connections as they appear looking at the 64 from the back (the serial port is next to the cassette port).

Figure 17-3. 64 Serial Port



Pin 1	SRQ in	FLAG of CIA 1
Pin 2	Ground	
Pin 3	ATN in	Pin 9 of user port
Pin 3	ATN out	PA3 of CIA 2
Pin 4	CLK in	PA6 of CIA 2
Pin 4	CLK out	PA4 of CIA 2
Pin 5	Data in	PA7 of CIA 2
Pin 5	Data out	PA5 of CIA 2
Pin 6	RESET	Connected to 64 reset line

Pin 6 is connected to the 64's reset line, which is why the disk drive and printer reset when the 64 is switched on. Data is transferred in and out through pin 5. Pin 1, the service request line, allows devices to request service from the 64. CLK is a clock signal and ATN (attention) is described below.

Both CIAs are used in processing. The part of ROM handling this can be inspected in detail by looking at the places where bit 7 of $DD00 (CIA 2's Port A) is used; this line inputs data bits. Data is transmitted from bit 5 of CIA 2 Port B, so $DD00 also controls data output. Other important functions of that location are bits 4 and 6, which provide input and output clock signals.

Briefly, the serial bus is controlled by the 64. Devices on the bus are talkers, listeners, or both. For example, printers listen and disk drives 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; and $60–$7F indicate secondary addresses. This is why secondary addresses are stored in the 64 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

Appendix A

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 Commodore 64s.

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. Common mistakes are substituting the letter O for the numeral 0, a lowercase l for the numeral 1, or an uppercase B for the numeral 8. Also, be sure to 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 programs in machine language, while 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 RUN/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 may still be 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 make a mistake? You can always retype the line, but at least you need to know how to backspace. Do you know how to enter reverse video, lowercase, and control characters? It's all explained in your 64's manual.

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 and cursor keys 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").

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, Commodore 64 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 (for example, {10 N}), you should type the key as many times as indicated. In this case, you would enter ten SHIFTed N's. One exception to this is that {SHIFT-SPACE} means to hold down the SHIFT key and type the space bar.

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.

Occasionally, you will see a single character within braces. These characters are entered by pressing CTRL while typing the letter indicated. For example, {A} is entered by pressing CTRL–A.

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 quote followed by a few characters, then try to change it by moving the cursor left, you'll only get a bunch of reverse-video characters. These are the symbols for cursor left. The only editing key that isn't programmable is the INST/DEL key, so you can still use INST/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 use INST/DEL to 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:
{CLR}	SHIFT	CLR/HOME	♥
{HOME}		CLR/HOME	S
{UP}	SHIFT	↑ CRSR ↓	●
{DOWN}		↑ CRSR ↓	Q
{LEFT}	SHIFT	← CRSR →	┃
{RIGHT}		← CRSR →	]
{RVS}	CTRL	9	R
{OFF}	CTRL	0	■
{BLK}	CTRL	1	■
{WHT}	CTRL	2	E
{RED}	CTRL	3	£
{CYN}	CTRL	4	◥
{PUR}	CTRL	5	▒
{GRN}	CTRL	6	↑
{BLU}	CTRL	7	←
{YEL}	CTRL	8	π

When You Read:	Press:		See:
{ 1 }	COMMODORE	1	✚
{ 2 }	COMMODORE	2	╭
{ 3 }	COMMODORE	3	✕
{ 4 }	COMMODORE	3	○
{ 5 }	COMMODORE	5	┴
{ 6 }	COMMODORE	6	▌
{ 7 }	COMMODORE	7	●
{ 8 }	COMMODORE	8	╋
{ F1 }		f1	■
{ F2 }	SHIFT	f1	■
{ F3 }		f3	■
{ F4 }	SHIFT	f3	■
{ F5 }		f5	■
{ F6 }	SHIFT	f5	■
{ F7 }		f7	■
{ F8 }	SHIFT	f7	■
←	←		←
↑	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, when entering a BASIC 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

Many 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 from 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 since proper spacing occasionally *is* important, 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.

Here'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 tap 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 64 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 64 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 three 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'll get the message FOUND PROOFREADER, but not the familiar LOADING. Don't worry; the Proofreader is now in memory. 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 ten spaces.

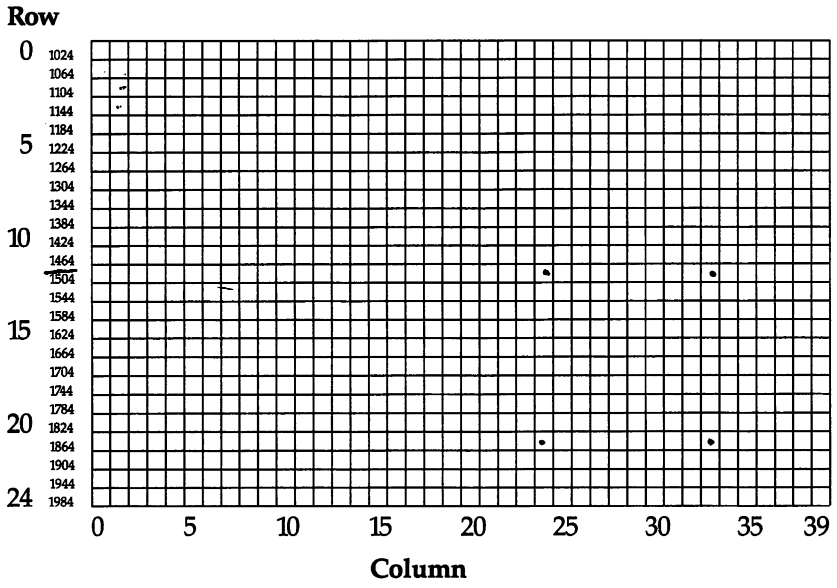
You can now reload the Proofreader into memory whenever LOAD or SAVE destroys it, restoring your personal typing helper.

Automatic Proofreader for Commodore 64 and VIC

```
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
```

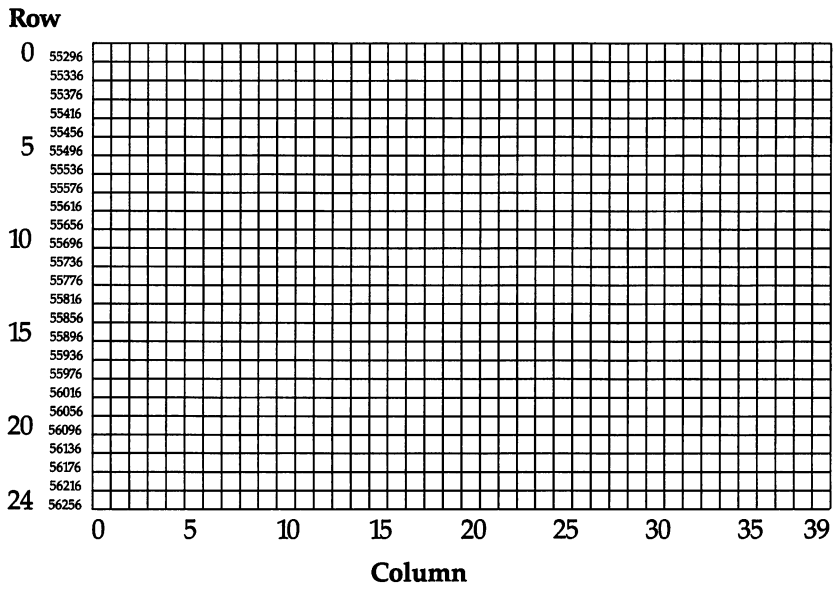
Screen Location Table

$ 0400



Screen Color Memory Table

$D800



Screen Color Codes

Color:	Black	White	Red	Cyan	Purple	Green	Blue	Yellow
Code:	0	1	2	3	4	5	6	7

Color:	Orange	Brown	Light Red	Dark Gray	Medium Gray	Light Green	Light Blue	Light Gray
Code:	8	9	10	11	12	13	14	15

Appendix G

ASCII Codes

Dec	Hex		Meaning	Dec	Hex	Meaning	Dec	Hex	Meaning	Dec	Hex	Meaning
0	00	NUL	- Null character	32	20	SPACE	64	40	@	96	60	`
1	01	SOH	- Start heading	33	21	!	65	41	A	97	61	a
2	02	STX	- Start text	34	22	″	66	42	B	98	62	b
3	03	ETX	- End text	35	23	#	67	43	C	99	63	c
4	04	EOT	- End transmission	36	24	$	68	44	D	100	64	d
5	05	ENQ	- Enquiry	37	25	%	69	45	E	101	65	e
6	06	ACK	- Acknowledge	38	26	&	70	46	F	102	66	f
7	07	BEL	- Ring bell	39	27	'	71	47	G	103	67	g
8	08	BS	- Backspace	40	28	(	72	48	H	104	68	h
9	09	HT	- Horizontal tabulation	41	29	)	73	49	I	105	69	i
10	0A	LF	- Linefeed	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	- Vertical tabulation	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	- Formfeed	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	- Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	- Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	- Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	- Data link escape	48	30	0	80	50	P	112	70	p
17	11	DC1	- Device control 1	49	31	1	81	51	Q	113	71	q
18	12	DC2	- Device control 2	50	32	2	82	52	R	114	72	r
19	13	DC3	- Device control 3	51	33	3	83	53	S	115	73	s
20	14	DC4	- Device control 4	52	34	4	84	54	T	116	74	t
21	15	NAK	- Negative acknowledge	53	35	5	85	55	U	117	75	u
22	16	SYN	- Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	ETB	- End transmission block	55	37	7	87	57	W	119	77	w
24	18	CAN	- Cancel	56	38	8	88	58	X	120	78	x
25	19	EM	- End medium	57	39	9	89	59	Y	121	79	y
26	1A	SUB	- Substitute	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	- Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	- File separator	60	3C	<	92	5C	\	124	7C	¦
29	1D	GS	- Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	- Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	US	- Unit separator	63	3F	?	95	5F	_	127	7F	DEL

ASCII (American Standard Code for Information Interchange) is largely followed by the 64—more closely than in earlier CBM equipment; alphabetic characters, numerals, and punctuation are generally similar, although the 64 has uppercase and lowercase letters switched with respect to standard ASCII. Standard ASCII, however, has a parity bit (bit 7) set to make the number of 1's in the byte even. This is why ASCII has only 128 characters.

Appendix H

Commodore ASCII Codes

Hex	Dec	Character	Hex	Dec	Character
05	5	WHITE	37	55	7
08	8	DISABLE	38	56	8
		SHIFT-COMMODORE	39	57	9
09	9	ENABLE	3A	58	:
		SHIFT-COMMODORE	3B	59	;
0D	13	RETURN	3C	60	<
0E	14	LOWERCASE	3D	61	=
11	17	CURSOR DOWN	3E	62	>
12	18	REVERSE VIDEO ON	3F	63	?
13	19	HOME	40	64	@
14	20	DELETE	41	65	A
1C	28	RED	42	66	B
1D	29	CURSOR RIGHT	43	67	C
1E	30	GREEN	44	68	D
1F	31	BLUE	45	69	E
20	32	SPACE	46	70	F
21	33	!	47	71	G
22	34	"	48	72	H
23	35	#	49	73	I
24	36	$	4A	74	J
25	37	%	4B	75	K
26	38	&	4C	76	L
27	39	'	4D	77	M
28	40	(	4E	78	N
29	41	)	4F	79	O
2A	42	*	50	80	P
2B	43	+	51	81	Q
2C	44	,	52	82	R
2D	45	–	53	83	S
2E	46	.	54	84	T
2F	47	/	55	85	U
30	48	0	56	86	V
31	49	1	57	87	W
32	50	2	58	88	X
33	51	3	59	89	Y
34	52	4	5A	90	Z
35	53	5	5B	91	[
36	54	6	5C	92	£

Hex	Dec	Character	Hex	Dec	Character
5D	93	]	85	133	f1
5E	94	↑	86	134	f3
5F	95	←	87	135	f5
60	96	─	88	136	f7
61	97	♠	89	137	f2
62	98	│	8A	138	f4
63	99	─	8B	139	f6
64	100	─	8C	140	f8
65	101	▔	8D	141	SHIFT-RETURN
66	102	▁	8E	142	UPPERCASE
67	103	▏	90	143	BLACK
68	104	▕	91	145	CURSOR UP
69	105	╮	92	146	REVERSE VIDEO OFF
6A	106	╰	93	147	CLEAR SCREEN
6B	107	╯	94	148	INSERT
6C	108	⌞	95	149	BROWN
6D	109	╲	96	150	LIGHT RED
6E	110	╱	97	151	GRAY 1
6F	111	⌜	98	152	GRAY 2
70	112	⌝	99	153	LIGHT GREEN
71	113	●	9A	154	LIGHT BLUE
72	114	▁	9B	155	GRAY 3
73	115	♥	9C	156	PURPLE
74	116	▏	9D	157	CURSOR LEFT
75	117	╭	9E	158	YELLOW
76	118	╳	9F	159	CYAN
77	119	○	A0	160	SHIFT-SPACE
78	120	♣	A1	161	▌
79	121	▕	A2	162	▄
7A	122	♦	A3	163	▔
7B	123	┼	A4	164	▁
7C	124	▒	A5	165	▏
7D	125	│	A6	166	▒
7E	126	π	A7	167	▕
7F	127	◥	A8	168	▒
81	129	ORANGE	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, 130–132, and 143 have no effect.
2. 192–223 same as 96–127, 224–254 same as 160–190, 255 same as 126.

Screen Character 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.

Appendix J

The VIC-II Chip

Hex Address	Decimal Address	Offset	Function							
D000	53248	0	Sprite 0 X-Position (low 8 bits)							
D001	53249	1	Sprite 0 Y-Position							
D002	53250	2	Sprite 1 X-Position (low)							
D003	53251	3	Sprite 1 Y-Position							
D004	53252	4	Sprite 2 X-Position (low)							
D005	53253	5	Sprite 2 Y-Position							
D006	53254	6	Sprite 3 X-position (low)							
D007	53255	7	Sprite 3 Y-Position							
D008	53256	8	Sprite 4 X-Position (low)							
D009	53257	9	Sprite 4 Y-Position							
D00A	53258	10	Sprite 5 X-Position (low)							
D00B	53259	11	Sprite 5 Y-Position							
D00C	53260	12	Sprite 6 X-Position (low)							
D00D	53261	13	Sprite 6 Y-Position							
D00E	53262	14	Sprite 7 X-Position (low)							
D00F	53263	15	Sprite 7 Y-Position							
D010	53264	16	High Bit of Sprite X-Position							
			Sprite 7	Sprite 6	Sprite 5	Sprite 4	Sprite 3	Sprite 2	Sprite 1	Sprite 0
D011	53265	17	Bit 8	Extended Color mode 1 = on	Bitmap mode 1 = on	Screen Blanking 0 = Blank	No. of Rows 1 = 25 0 = 24	Vertical Screen Position	0–7 pixels	
D012	53266	18	Raster Scan Line and Write Register for Raster Interrupts							
			Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
D013	53267	19	Light Pen Horizontal Position							
D014	53268	20	Light Pen Vertical Position							
D015	53269	21	Enable Sprites (1 = on, 0 = off)							
			Sprite 7	Sprite 6	Sprite 5	Sprite 4	Sprite 3	Sprite 2	Sprite 1	Sprite 0
D016	53270	22	1	1	Chip Reset 0 = Normal	Multicolor Mode 1 = Enable	No. of Columns 1 = 40 0 = 38	Horizontal Screen Position	0–7 Pixels	

Hex Address	Decimal Address	Offset	Function							
D017	53271	23	Sprite Vertical Expansion (1 = on, 0 = off)							
			Sprite 7	Sprite 6	Sprite 5	Sprite 4	Sprite 3	Sprite 2	Sprite 1	Sprite 0
D018	53272	24	Screen Base Address				Character Definitions Base Address			
			Bit 13	Bit 12	Bit 11	Bit 10	Bit 13	Bit 12	Bit 11	(1)
D019	53273	25	Interrupt Flags and Write Register to Clear Flags							
			1 = Interrupts 0 = No Interrupts	1	1	1	Light Pen	Sprite-Sprite Collision	Sprite-Data Collision	Raster Scan
D01A	53274	26	Interrupt Enable (1 = enabled, 0 = not enabled)							
			1	1	1	1	Light Pen	Sprite-Sprite Collision	Sprite-Data Collision	Raster Scan
D01B	53275	27	Sprite-Data Priority (1 = data, 0 = sprite)							
			Sprite 7	Sprite 6	Sprite 5	Sprite 4	Sprite 3	Sprite 2	Sprite 1	Sprite 0
D01C	53276	28	Sprite Multicolor Mode (1 = multicolor, 0 = hi-res)							
			Sprite 7	Sprite 6	Sprite 5	Sprite 4	Sprite 3	Sprite 2	Sprite 1	Sprite 0
D01D	53277	29	Sprite Horizontal Expansion (1 = on, 0 = off)							
			Sprite 7	Sprite 6	Sprite 5	Sprite 4	Sprite 3	Sprite 2	Sprite 1	Sprite 0
D01E	53278	30	Sprite-Sprite Collision Register (cleared only when read)							
			Sprite 7	Sprite 6	Sprite 5	Sprite 4	Sprite 3	Sprite 2	Sprite 1	Sprite 0
D01F	53279	31	Sprite-Data Collision Register (cleared only when read)							
			Sprite 7	Sprite 6	Sprite 5	Sprite 4	Sprite 3	Sprite 2	Sprite 1	Sprite 0
D020	53280	32	1	1	1	1	Border Color (0–15)			
D021	53281	33	1	1	1	1	Background Color 0 (0–15)			
D022	53282	34	1	1	1	1	Background Color 1			
D023	53283	35	1	1	1	1	Background Color 2			
D024	53284	36	1	1	1	1	Background Color 3			
D025	53285	37	1	1	1	1	Sprite Multicolor 0			
D026	53286	38	1	1	1	1	Sprite Multicolor 1			
D027	53287	39	1	1	1	1	Sprite 0 Color			

Continued on next page.

Hex Address	Decimal Address	Offset	Function				
D028	53288	40	1	1	1	1	Sprite 1 Color
D029	53289	41	1	1	1	1	Sprite 2 Color
D02A	53290	42	1	1	1	1	Sprite 3 Color
D02B	53291	43	1	1	1	1	Sprite 4 Color
D02C	53292	44	1	1	1	1	Sprite 5 Color
D02D	53293	45	1	1	1	1	Sprite 6 Color
D02E	53294	46	1	1	1	1	Sprite 7 Color

Appendix K

The SID Chip

	Hex Address	Decimal Address	Offset	Function								
	D400	54272	0	Frequency Control (low byte)								
	D407	54279	7									
	D40E	54286	14									
	D401	54273	1	Frequency Control (high byte)								
	D408	54280	8									
	D40F	54287	15									
	D402	54274	2	Pulse Width (bits 7–0)								
	D409	54281	9									
	D410	54288	16									
Voice 1	D403	54275	3	not used				Pulse Width (bits 11–8				
Voice 2	D40A	54282	10					bit 11	bit 10	bit 9	bit 8	
Voice 3	D411	54289	17									
	D404	54276	4	Control Register (1 = on, 0 = off)								
	D40B	54283	11	noise	pulse	sawtooth	triangle	test	ring	sync	gate	
	D412	54290	18									
	D405	54277	5	Attack (0–15)				Decay (0–15)				Write-only Registers
	D40C	54284	12									
	D413	54291	19									
	D406	54278	6	Sustain Level (0–15)				Release (0–15)				
	D40D	54285	13									
	D414	54292	20									
	D415	54293	21	not used					Filter Control (bits 2–0)			
									FC2	FC1	FC0	
	D416	54294	22	Filter Control (bits 10–3)								
				FC10	FC9	FC8	FC7	FC6	FC5	FC4	FC3	
	D417	54295	23	Resonance (0–15)				Filtered Voice(s) (1 = on, 0 = off)				
								External	Voice 3	Voice 2	Voice 1	
	D418	54296	24	Cutout Voice 3 (0=not cut 1=cut)	Filter Type			Master Volume (0–15)				
					high pass	band pass	low pass					
	D419	54297	25	Potentiometer X (0–255)								
	D41A	54298	26	Potentiometer Y (0–255)								Read-only Registers
	D41B	54299	27	Read Waveform of Voice 3 (0–255)								
	D41C	54300	28	Read Envelope of Voice 3 (0–255)								

Appendix L

Device Numbers

Table of second parameter in OPEN. Example: OPEN 5,4 opens file 5 to printer.

0	Keyboard
1	Tape (not used in SX-64 models)
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 Conversion Table

Hex	Low Dec.	High Dec.	Hex	Low Dec.	High Dec.	Hex	Low Dec.	High Dec.	Hex	Low Dec.	High 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

Opcodes in Detail

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										BIT
								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									INC
						E8 2							INX
						C8 2							INY
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

Table of 6502/6510 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

6502/6510 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	
RRA (ROR,ADC)	**6F**	**7F**	7B	**67**	**77**		63	73	
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**
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
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 6502/6510 as opcodes. These commands are not part of the 6502/6510'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 6502/6510, it is a fact that some published software containing these codes has given no problems. All 6502/6510s 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 a monitor 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.

Converting Commodore 64, VIC-20, 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 Commodore 64

VIC-20 programs. VIC-20 disk programs should load without difficulty into the Commodore 64. However, 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 a MERGE to read it into the computer. This writes the program in smaller chunks, so loading is easier. If this fails, loading into a CBM/PET first (see below), and then into the 64, is likely to work. Alternatively, the program could even be transferred by modem.

PET/CBM programs. PET/CBM disk programs should load into the 64, 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 64 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 the 64 (the rest of the program is fine).

To load a program from one of these very early PETs into the 64, 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 64-loadable program.

Loading 64 Programs into Other Computers

Loading 64 programs into the VIC-20. This is no problem with disks. However, tape may be unsuccessful, because of timing differences. Use the same cures as you would when loading VIC programs into the 64.

Loading 64 programs into PET/CBMs. This is slightly tricky; the 64's screen is usually at the place PET/CBM BASIC starts. First move the screen, then move the start of BASIC down with POKE 44,4: POKE 1024,0: NEW; now load the 64 program and save to tape or disk. The result will load successfully into PET/CBM machines.

Alternatively, change the LOAD address to $0401 on disk or tape, by reading and writing back the program file (disk) or overwriting the header (tape is trickier!). Other methods are possible, too. The following program, "Simulate CBM/PET," modifies the 64 to resemble a CBM/PET.

Simulate CBM/PET

```
10 SIMPLE PET/CBM SIMULATOR
20 POKE 792,193:POKE 646,5:POKE 53281,0:POKE 53280
   ,0
30 POKE 56576,5:POKE 53272,4:POKE 648,128
40 POKE 1024,0:POKE 43,1:POKE 44,4:POKE 55,0:POKE
   {SPACE}56,128:PRINT"{CLR}":NEW
```

Program Conversion

Programs which are pure BASIC, even for non-CBM computers, can often be converted to run on the 64. Difficulties are likely, though, particularly if disk or tape access is needed. The 64 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 64.

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 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 the 64 in several small ways—GO TO isn't allowed as one word in the oldest PETs, for example, and DS is a reserved variable in the most recent models. Pure BASIC (without SYS, PEEK, POKE, WAIT, or USR) is compatible to a very large extent; screen problems can occur, 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 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 RUN/STOP key and manipulating the keyboard buffer are examples. You may be able to delete some commands; disabling RUN/STOP isn't very important. In addition, you may be able to replace some PEEKs amd POKEs. For instance, the 64's POKE 198,0 has a BASIC equivalent, FOR J=1 TO 10: GET X$: NEXT, which clears a ten-character keyboard buffer. CBM's POKE 59468,14 to switch to lowercase is replaceable by PRINT CHR$(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 2.0'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.

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 read joysticks, for example) can be tricky as well, 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 should help you identify the purpose of a few 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

Appendix R

Supermon 64

Supermon is a relatively short monitor for the 64. It is a public domain program, so it is free. It can be loaded like BASIC and run, and this puts it into the top of BASIC memory, leaving RAM from $C000 free for ML programs. Chapter 8 explains its operation. This version prints your input in white, and the monitor's output in cyan, for good readability.

SYS 38910 reenters *Supermon* after .X has been used to exit to BASIC, assuming *Supermon* is in its usual position in RAM and hasn't been disconnected by RUN/STOP–RESTORE. SYS to an address containing zero (SYS 13, for example) will reenable the monitor as well.

The following instructions tell you how to enter and save *Supermon.* The first step is to switch on the 64 and type in the entire program (not necessarily all at once) and save "Supermon Data" to tape or disk. Note that complete accuracy is required in entering the data. For security, a simple checksum is included. It's often most efficient to have a friend call out the numbers as you type in programs.

Next, turn the computer off, then on again, and type in:

POKE 43,1: POKE 44,18: POKE 18*256,0: NEW

to move BASIC up out of the way. After this, LOAD "SUPERMON DATA",8 (or ,1 for tape). Run the Supermon Data program (which takes about 25 seconds), then SAVE "SUPERMON 64",8 (or ,1 for tape). Now, "Supermon 64" becomes the primary version of *Supermon;* just load and run it.

Supermon Data

For mistake-proof program entry, be sure to use the "Automatic Proofreader," Appendix C.

```
1Ø FOR J=2Ø48 TO 4587: READ X: POKE J,X: T=T+X: NE
   XT                                        :rem 24
2Ø IF T<>283295 THEN PRINT "CHECKSUM ERROR": END
                                            :rem 234
3Ø POKE 43,1: POKE 44,8: POKE 45,235: POKE 46,17:
   {SPACE}CLR: LIST                          :rem 27
4ØØ DATA Ø,26,8,1ØØ,Ø,153,34,147,18,29,29,29,29,83
    ,85,8Ø,69,82,32                         :rem 134
4Ø1 DATA 54,52,45,77,79,78,Ø,49,8,11Ø,Ø,153,34,17,
    32,32,32,32,32                           :rem 69
4Ø2 DATA 32,32,32,32,32,32,32,32,32,32,Ø,75,8,12Ø,
    Ø,153,34,17,32                           :rem 3Ø
4Ø3 DATA 46,46,74,73,77,32,66,85,84,84,69,82,7Ø,73
    ,69,76,68,Ø,1Ø2                         :rem 173
4Ø4 DATA 8,13Ø,Ø,158,4Ø,194,4Ø,52,51,41,17Ø,5Ø,53,
    54,172,194,4Ø                            :rem 14
4Ø5 DATA 52,52,41,17Ø,49,5Ø,55,41,Ø,Ø,Ø,17Ø,17Ø,17
    Ø,17Ø,17Ø,17Ø                           :rem 255
4Ø6 DATA 17Ø,17Ø,17Ø,17Ø,17Ø,17Ø,17Ø,17Ø,17Ø,17Ø,1
    7Ø,17Ø,17Ø,17Ø                           :rem 64
4Ø7 DATA 17Ø,17Ø,17Ø,17Ø,165,45,133,34,165,46,133,
    35,165,55,133                            :rem 34
```

```
408 DATA 36,165,56,133,37,160,Ø,165,34,2Ø8,2,198,3
    5,198,34,177,34                        :rem 15Ø
409 DATA 2Ø8,6Ø,165,34,2Ø8,2,198,35,198,34,177,34,
    24Ø,33,133,38                           :rem 49
41Ø DATA 165,34,2Ø8,2,198,35,198,34,177,34,24,1Ø1,
    36,17Ø,165,38                           :rem 43
411 DATA 1Ø1,37,72,165,55,2Ø8,2,198,56,198,55,1Ø4,
    145,55,138,72                           :rem 5Ø
412 DATA 165,55,2Ø8,2,198,56,198,55,1Ø4,145,55,24,
    144,182,2Ø1,79                         :rem 1Ø2
413 DATA 2Ø8,237,165,55,133,51,165,56,133,52,1Ø8,5
    5,Ø,79,79,79,79                        :rem 163
414 DATA 173,23Ø,255,Ø,141,22,3,173,231,255,Ø,141,
    23,3,169,128,32                        :rem 1Ø3
415 DATA 144,255,Ø,Ø,216,1Ø4,141,62,2,1Ø4,141,61,2
    ,1Ø4,141,6Ø,2                          :rem 238
416 DATA 1Ø4,141,59,2,1Ø4,17Ø,1Ø4,168,56,138,233,2
    ,141,58,2,152                           :rem 19
417 DATA 233,Ø,Ø,141,57,2,186,142,63,2,32,87,253,Ø
    ,162,66,169,42                          :rem 72
418 DATA 32,87,25Ø,Ø,169,82,2Ø8,52,23Ø,193,2Ø8,6,2
    3Ø,194,2Ø8,2,23Ø                       :rem 179
419 DATA 38,96,32,2Ø7,255,2Ø1,13,2Ø8,248,1Ø4,1Ø4,1
    69,159,32,21Ø                           :rem 36
42Ø DATA 255,169,Ø,Ø,133,38,162,13,169,46,32,87,25
    Ø,Ø,169,5,32,21Ø                       :rem 169
421 DATA 255,32,62,248,Ø,2Ø1,46,24Ø,249,2Ø1,32,24Ø
    ,245,162,14,221                        :rem 1Ø6
422 DATA 183,255,Ø,2Ø8,12,138,1Ø,17Ø,189,199,255,Ø
    ,72,189,198,255                        :rem 154
423 DATA Ø,72,96,2Ø2,16,236,76,237,25Ø,Ø,165,193,1
    41,58,2,165,194                        :rem 137
424 DATA 141,57,2,96,169,8,133,29,16Ø,Ø,Ø,32,84,25
    3,Ø,177,193,32                          :rem 8Ø
425 DATA 72,25Ø,Ø,32,51,248,Ø,198,29,2Ø8,241,96,32
    ,136,25Ø,Ø,144                          :rem 74
426 DATA 11,162,Ø,Ø,129,193,193,193,24Ø,3,76,237,2
    5Ø,Ø,32,51,248                          :rem 7Ø
427 DATA Ø,198,29,96,169,59,133,193,169,2,133,194,
    169,5,96,152,72                        :rem 177
428 DATA 32,87,253,Ø,1Ø4,162,46,76,87,25Ø,Ø,169,15
    9,32,21Ø,255,162                       :rem 189
429 DATA Ø,Ø,189,234,255,Ø,32,21Ø,255,232,224,22,2
    Ø8,245,16Ø,59                           :rem 21
43Ø DATA 32,194,248,Ø,173,57,2,32,72,25Ø,Ø,173,58,
    2,32,72,25Ø,Ø                           :rem 11
431 DATA 32,183,248,Ø,32,141,248,Ø,24Ø,92,32,62,24
    8,Ø,32,121,25Ø                          :rem 54
432 DATA Ø,144,51,32,1Ø5,25Ø,Ø,32,62,248,Ø,32,121,
    25Ø,Ø,144,4Ø,32                         :rem 77
433 DATA 1Ø5,25Ø,Ø,169,159,32,21Ø,255,32,225,255,2
    4Ø,6Ø,166,38,2Ø8                       :rem 176
```

```
434 DATA 56,165,195,197,193,165,196,229,194,144,46
    ,160,58,32,194                         :rem 130
435 DATA 248,0,32,65,250,0,32,139,248,0,240,224,76
    ,237,250,0,32                          :rem 14
436 DATA 121,250,0,144,3,32,128,248,0,32,183,248,0
    ,208,7,32,121                          :rem 5
437 DATA 250,0,144,235,169,8,133,29,32,62,248,0,32
    ,161,248,0,208                         :rem 76
438 DATA 248,76,71,248,0,32,207,255,201,13,240,12,
    201,32,208,209                         :rem 69
439 DATA 32,121,250,0,144,3,32,128,248,0,169,159,3
    2,210,255,174                          :rem 23
440 DATA 63,2,154,120,173,57,2,72,173,58,2,72,173,
    59,2,72,173,60                         :rem 80
441 DATA 2,174,61,2,172,62,2,64,169,159,32,210,255
    ,174,63,2,154                          :rem 28
442 DATA 108,2,160,160,1,132,186,132,185,136,132,1
    83,132,144,132                         :rem 64
443 DATA 147,169,64,133,187,169,2,133,188,32,207,2
    55,201,32,240                          :rem 44
444 DATA 249,201,13,240,56,201,34,208,20,32,207,25
    5,201,34,240,16                        :rem 104
445 DATA 201,13,240,41,145,187,230,183,200,192,16,
    208,236,76,237                         :rem 79
446 DATA 250,0,32,207,255,201,13,240,22,201,44,208
    ,220,32,136,250                        :rem 93
447 DATA 0,41,15,240,233,201,3,240,229,133,186,32,
    207,255,201,13                         :rem 54
448 DATA 96,108,48,3,108,50,3,32,150,249,0,208,212
    ,169,159,32,210                        :rem 130
449 DATA 255,169,0,0,32,239,249,0,165,144,41,16,20
    8,196,76,71,248                        :rem 151
450 DATA 0,32,150,249,0,201,44,208,186,32,121,250,
    0,32,105,250,0                         :rem 35
451 DATA 32,207,255,201,44,208,173,32,121,250,0,16
    5,193,133,174                          :rem 18
452 DATA 165,194,133,175,32,105,250,0,32,207,255,2
    01,13,208,152                          :rem 16
453 DATA 169,159,32,210,255,32,242,249,0,76,71,248
    ,0,165,194,32                          :rem 43
454 DATA 72,250,0,165,193,72,74,74,74,74,32,96,250
    ,0,170,104,41                          :rem 34
455 DATA 15,32,96,250,0,72,138,32,210,255,104,76,2
    10,255,9,48,201                        :rem 123
456 DATA 58,144,2,105,6,96,162,2,181,192,72,181,19
    4,149,192,104                          :rem 49
457 DATA 149,194,202,208,243,96,32,136,250,0,144,2
    ,133,194,32,136                        :rem 133
458 DATA 250,0,144,2,133,193,96,169,0,0,133,42,32,
    62,248,0,201,32                        :rem 108
459 DATA 208,9,32,62,248,0,201,32,208,14,24,96,32,
    175,250,0,10,10                        :rem 109
```

```
460 DATA 10,10,133,42,32,62,248,0,32,175,250,0,5,4
    2,56,96,201,58                         :rem 56
461 DATA 144,2,105,8,41,15,96,162,2,44,162,0,0,180
    ,193,208,8,180                         :rem 65
462 DATA 194,208,2,230,38,214,194,214,193,96,32,62
    ,248,0,201,32                          :rem 31
463 DATA 240,249,96,169,0,0,141,0,0,1,32,204,250,0
    ,32,143,250,0                         :rem 244
464 DATA 32,124,250,0,144,9,96,32,62,248,0,32,121,
    250,0,176,222                          :rem 11
465 DATA 174,63,2,154,169,159,32,210,255,169,63,32
    ,210,255,76,71                         :rem 99
466 DATA 248,0,32,84,253,0,202,208,250,96,230,195,
    208,2,230,196                          :rem 32
467 DATA 96,162,2,181,192,72,181,39,149,192,104,14
    9,39,202,208,243                      :rem 206
468 DATA 96,165,195,164,196,56,233,2,176,14,136,14
    4,11,165,40,164                       :rem 157
469 DATA 41,76,51,251,0,165,195,164,196,56,229,193
    ,133,30,152,229                       :rem 155
470 DATA 194,168,5,30,96,32,212,250,0,32,105,250,0
    ,32,229,250,0                           :rem 5
471 DATA 32,12,251,0,32,229,250,0,32,47,251,0,32,1
    05,250,0,144,21                        :rem 81
472 DATA 166,38,208,100,32,40,251,0,144,95,161,193
    ,129,195,32,5                          :rem 32
473 DATA 251,0,32,51,248,0,208,235,32,40,251,0,24,
    165,30,101,195                         :rem 52
474 DATA 133,195,152,101,196,133,196,32,12,251,0,1
    66,38,208,61,161                      :rem 181
475 DATA 193,129,195,32,40,251,0,176,52,32,184,250
    ,0,32,187,250                          :rem 33
476 DATA 0,76,125,251,0,32,212,250,0,32,105,250,0,
    32,229,250,0,32                        :rem 85
477 DATA 105,250,0,32,62,248,0,32,136,250,0,144,20
    ,133,29,166,38                         :rem 63
478 DATA 208,17,32,47,251,0,144,12,165,29,129,193,
    32,51,248,0,208                       :rem 134
479 DATA 238,76,237,250,0,76,71,248,0,32,212,250,0
    ,32,105,250,0                          :rem 17
480 DATA 32,229,250,0,32,105,250,0,32,62,248,0,162
    ,0,0,32,62,248                         :rem 44
481 DATA 0,201,39,208,20,32,62,248,0,157,16,2,232,
    32,207,255,201                         :rem 55
482 DATA 13,240,34,224,32,208,241,240,28,142,0,0,1
    ,32,143,250,0                         :rem 243
483 DATA 144,198,157,16,2,232,32,207,255,201,13,24
    0,9,32,136,250                         :rem 75
484 DATA 0,144,182,224,32,208,236,134,28,169,144,3
    2,210,255,32,87                       :rem 135
485 DATA 253,0,162,0,0,160,0,0,177,193,221,16,2,20
    8,12,200,232,228                      :rem 141
```

```
486 DATA 28,208,243,32,65,250,0,32,84,253,0,32,51,
    248,0,166,38,208                            :rem 178
487 DATA 141,32,47,251,0,176,221,76,71,248,0,32,21
    2,250,0,133,32                              :rem 62
488 DATA 165,194,133,33,162,0,0,134,40,169,147,32,
    210,255,169,159                             :rem 141
489 DATA 32,210,255,169,22,133,29,32,106,252,0,32,
    202,252,0,133                               :rem 11
490 DATA 193,132,194,198,29,208,242,169,145,32,210
    ,255,76,71,248                              :rem 112
491 DATA 0,160,44,32,194,248,0,32,84,253,0,32,65,2
    50,0,32,84,253                              :rem 65
492 DATA 0,162,0,0,161,193,32,217,252,0,72,32,31,2
    53,0,104,32,53                              :rem 41
493 DATA 253,0,162,6,224,3,208,18,164,31,240,14,16
    5,42,201,232,177                            :rem 166
494 DATA 193,176,28,32,194,252,0,136,208,242,6,42,
    144,14,189,42                               :rem 46
495 DATA 255,0,32,165,253,0,189,48,255,0,240,3,32,
    165,253,0,202                               :rem 21
496 DATA 208,213,96,32,205,252,0,170,232,208,1,200
    ,152,32,194,252                             :rem 117
497 DATA 0,138,134,28,32,72,250,0,166,28,96,165,31
    ,56,164,194,170                             :rem 147
498 DATA 16,1,136,101,193,144,1,200,96,168,74,144,
    11,74,176,23,201                            :rem 177
499 DATA 34,240,19,41,7,9,128,74,170,189,217,254,0
    ,176,4,74,74,74                             :rem 160
500 DATA 74,41,15,208,4,160,128,169,0,0,170,189,29
    ,255,0,133,42                               :rem 19
501 DATA 41,3,133,31,152,41,143,170,152,160,3,224,
    138,240,11,74                               :rem 253
502 DATA 144,8,74,74,9,32,136,208,250,200,136,208,
    242,96,177,193                              :rem 94
503 DATA 32,194,252,0,162,1,32,254,250,0,196,31,20
    0,144,241,162                               :rem 2
504 DATA 3,192,4,144,242,96,168,185,55,255,0,133,4
    0,185,119,255                               :rem 45
505 DATA 0,133,41,169,0,0,160,5,6,41,38,40,42,136,
    208,248,105,63                              :rem 57
506 DATA 32,210,255,202,208,236,169,32,44,169,13,7
    6,210,255,32,212                            :rem 173
507 DATA 250,0,32,105,250,0,32,229,250,0,32,105,25
    0,0,162,0,0,134                             :rem 71
508 DATA 40,169,159,32,210,255,32,87,253,0,32,114,
    252,0,32,202,252                            :rem 162
509 DATA 0,133,193,132,194,32,225,255,240,5,32,47,
    251,0,176,233                               :rem 23
510 DATA 76,71,248,0,32,212,250,0,169,3,133,29,32,
    62,248,0,32,161                             :rem 109
```

```
511 DATA 248,Ø,2Ø8,248,165,32,133,193,165,33,133,1
    94,76,7Ø,252,Ø                              :rem 84
512 DATA 197,4Ø,24Ø,3,32,21Ø,255,96,32,212,25Ø,Ø,3
    2,1Ø5,25Ø,Ø,142                             :rem 93
513 DATA 17,2,162,3,32,2Ø4,25Ø,Ø,72,2Ø2,2Ø8,249,16
    2,3,1Ø4,56,233                              :rem 53
514 DATA 63,16Ø,5,74,11Ø,17,2,11Ø,16,2,136,2Ø8,246
    ,2Ø2,2Ø8,237,162                           :rem 158
515 DATA 2,32,2Ø7,255,2Ø1,13,24Ø,3Ø,2Ø1,32,24Ø,245
    ,32,2Ø8,254,Ø                              :rem 248
516 DATA 176,15,32,156,25Ø,Ø,164,193,132,194,133,1
    93,169,48,157                               :rem 51
517 DATA 16,2,232,157,16,2,232,2Ø8,219,134,4Ø,162,
    Ø,Ø,134,38,24Ø                              :rem 56
518 DATA 4,23Ø,38,24Ø,117,162,Ø,Ø,134,29,165,38,32
    ,217,252,Ø,166                              :rem 67
519 DATA 42,134,41,17Ø,188,55,255,Ø,189,119,255,Ø,
    32,185,254,Ø,2Ø8                           :rem 188
52Ø DATA 227,162,6,224,3,2Ø8,25,164,31,24Ø,21,165,
    42,2Ø1,232,169                              :rem 63
521 DATA 48,176,33,32,191,254,Ø,2Ø8,2Ø4,32,193,254
    ,Ø,2Ø8,199,136                              :rem 85
522 DATA 2Ø8,235,6,42,144,11,188,48,255,Ø,189,42,2
    55,Ø,32,185,254                            :rem 137
523 DATA Ø,2Ø8,181,2Ø2,2Ø8,2Ø9,24Ø,1Ø,32,184,254,Ø
    ,2Ø8,171,32,184                            :rem 11Ø
524 DATA 254,Ø,2Ø8,166,165,4Ø,197,29,2Ø8,16Ø,32,1Ø
    5,25Ø,Ø,164,31                              :rem 73
525 DATA 24Ø,4Ø,165,41,2Ø1,157,2Ø8,26,32,28,251,Ø,
    144,1Ø,152,2Ø8                              :rem 58
526 DATA 4,165,3Ø,16,1Ø,76,237,25Ø,Ø,2ØØ,2Ø8,25Ø,1
    65,3Ø,16,246,164                           :rem 162
527 DATA 31,2Ø8,3,185,194,Ø,Ø,145,193,136,2Ø8,248,
    165,38,145,193                              :rem 96
528 DATA 32,2Ø2,252,Ø,133,193,132,194,169,159,32,2
    1Ø,255,16Ø,65                               :rem 32
529 DATA 32,194,248,Ø,32,84,253,Ø,32,65,25Ø,Ø,32,8
    4,253,Ø,169,5                               :rem 25
53Ø DATA 32,21Ø,255,76,176,253,Ø,168,32,191,254,Ø,
    2Ø8,17,152,24Ø                              :rem 73
531 DATA 14,134,28,166,29,221,16,2,8,232,134,29,16
    6,28,4Ø,96,2Ø1                              :rem 79
532 DATA 48,144,3,2Ø1,71,96,56,96,64,2,69,3,2Ø8,8,
    64,9,48,34,69                               :rem 61
533 DATA 51,2Ø8,8,64,9,64,2,69,51,2Ø8,8,64,9,64,2,
    69,179,2Ø8,8,64                            :rem 162
534 DATA 9,Ø,Ø,34,68,51,2Ø8,14Ø,68,Ø,Ø,17,34,68,51
    ,2Ø8,14Ø,68,154                            :rem 121
535 DATA 16,34,68,51,2Ø8,8,64,9,16,34,68,51,2Ø8,8,
    64,9,98,19,12Ø                             :rem 1Ø7
```

```
536 DATA 169,Ø,Ø,33,129,13Ø,Ø,Ø,Ø,Ø,89,77,145,146,
    134,74,133,157                          :rem 69
537 DATA 44,41,44,35,4Ø,36,89,Ø,Ø,88,36,36,Ø,Ø,28,
    138,28,35,93,139                        :rem 187
538 DATA 27,161,157,138,29,35,157,139,29,161,Ø,Ø,4
    1,25,174,1Ø5,168                        :rem 193
539 DATA 25,35,36,83,27,35,36,83,25,161,Ø,Ø,26,91,
    91,165,1Ø5,36                           :rem 37
54Ø DATA 36,174,174,168,173,41,Ø,Ø,124,Ø,Ø,21,156,
    1Ø9,156,165,1Ø5                         :rem 113
541 DATA 41,83,132,19,52,17,165,1Ø5,35,16Ø,216,98,
    9Ø,72,38,98,148                         :rem 153
542 DATA 136,84,68,2ØØ,84,1Ø4,68,232,148,Ø,Ø,18Ø,8
    ,132,116,18Ø,4Ø                         :rem 122
543 DATA 11Ø,116,244,2Ø4,74,114,242,164,138,Ø,Ø,17
    Ø,162,162,116                           :rem 1Ø
544 DATA 116,116,114,68,1Ø4,178,5Ø,178,Ø,Ø,34,Ø,Ø,
    26,26,38,38,114                         :rem 116
545 DATA 114,136,2ØØ,196,2Ø2,38,72,68,68,162,2ØØ,5
    8,59,82,77,71                           :rem 56
546 DATA 88,76,83,84,7Ø,72,68,8Ø,44,65,66,249,Ø,53
    ,249,Ø,2Ø4,248                          :rem 12Ø
547 DATA Ø,247,248,Ø,86,249,Ø,137,249,Ø,244,249,Ø,
    12,25Ø,Ø,62,251                         :rem 123
548 DATA Ø,146,251,Ø,192,251,Ø,56,252,Ø,91,253,Ø,1
    38,253,Ø,172,253                        :rem 16Ø
549 DATA Ø,7Ø,248,Ø,255,247,Ø,237,247,Ø,13,32,32,3
    2,8Ø,67,32,32                           :rem 17
55Ø DATA 83,82,32,65,67,32,88,82,32,89,82,32,83,8Ø
    ,Ø                                      :rem 21
```

Index

The Commodore 64 is an amazing computer. Its price makes it available to the average family. Its power and flexibility allow it to perform sophisticated tasks, from word processing to business calculations. And its sound and graphics capabilities make it the perfect fast-action arcade game machine.

Until now, complete information and advice on getting the most out of the 64 have been hard to come by. *Programming the Commodore 64: The Definitive Guide*, by noted Commodore authority Raeto Collin West, provides that information. It's the encyclopedic reference guide to the Commodore 64.

In the tradition of the renowned *Programming the PET/CBM* and *Programming the VIC*, West presents 17 chapters of dictionaries, maps, BASIC and machine language examples, and programming aids. Unlike so many computer books, which are read only once, then discarded, *Programming the Commodore 64* remains an invaluable guide to your learning. You can lose yourself in this book for weeks.

It starts with BASIC and probes more deeply with each chapter. Ready-to-type-in programs show you how to use the BASIC and Kernal ROMs, the 6502/6510 microprocessor, the CIA, VIC, and SID chips, and the hidden RAM beneath the ROM in the 64. And major peripherals—tape and disk drives, printers, plotters, and modems—are discussed as well. You'll discover many amazing things you can do with your 64. Here's a sample of what you'll find:

- Dozens of programming techniques and tricks to experiment with.
- A detailed dictionary of every Commodore 64 BASIC keyword.
- A dictionary of extensions to BASIC, including program examples.
- Complete chapters on sound and graphics.
- An annotated list of the 6502/6510 machine language instruction set.
- A thorough map of the 64's ROM; listed side-by-side with major VIC ROM differences.
- Numerous appendices, including advice on translating programs between the 64, VIC and PET/CBM machines.

Programming the Commodore 64 helps you improve your programming by giving you the technical information and advice you need. It answers the questions other reference guides leave unanswered. And, as with all COMPUTE! books, the appendices are packed with useful quick-reference information and programming aids—including "The Automatic Proofreader" and *Supermon*. No matter what your programming level, *Programming the Commodore 64* is one of the most valuable books on the 64 you could own.

ISBN 0-942386-50-7