MAPPING THE VIC

The comprehensive memory guide for the VIC-20® home computer. An indispensable source book, with programming tips, examples, and detailed explanations.

Russ Davies

MAPPING
THE
VIC

Russ Davies

VIC-20 is a registered trademark of Commodore Electronics, Ltd.

Contents

Foreword

This is a memory map of the VIC-20 home computer. It's your *road map* to the internal architecture and operating systems of your computer, showing you how memory is constructed, how it's used, and how *you* can access it.

Mapping the VIC isn't just a listing of the computer's memory locations. It explains the purpose of each location and shows you how to change the contents of many of them. You'll be able to make the computer do what *you* want it to.

You'll be able to understand other programmers' efforts far more easily when you can look up the pointers, POKEs, PEEKs, and SYS commands that they've used. Seeing how someone else has programmed is only a step away from learning new programming techniques of your own. You can use the extensive index and label cross reference to learn about a subject of interest, reading in-depth explanations of such things as tape use with the VIC, or how to call BASIC or Kernal routines from your own programs.

There are numerous programming techniques already explained in this book. How to use the block SAVE/LOAD feature of the VIC; how to make the VIC's screen into a 40-column display with a new character set; even how to control the various connectors on the back of the VIC from within your own programs.

If you're programming in BASIC, you'll appreciate the easy-to-understand explanations of how to use the advanced features of the VIC-20 in your own programs. If you're using machine language, you'll come back to *Mapping the VIC* over and over, referring to specific memory locations and routines. Whether you're a beginning programmer or a veteran, you'll have the most complete guide to the VIC-20's memory available.

Introduction

A memory map is like a road map. It can show you something you
never dreamed was there, help you find the things that you always
knew were there, and show you what you've never seen. It can help
you get to a place from where you are and it's always there to
remind you of forgotten details. A memory map shows you how
things in memory are constructed, how the memory is used, who
can use it, and what it means in the larger scheme of things.

   This book is designed to be a working resource as you program.
When you look at others' programs, you can refer to the memory
locations that you see PEEKed and POKEd to understand what is
being done. SYS locations aren't as mysterious when you have a
memory map that tells you what a routine is doing and how. The
manipulation of pointers within a program is understandable once
you can read what the pointer is used for and what the effect is
when it changes.

   And a memory map can shed light on the methods and tech-
niques used by other programmers. Understanding what their pro-
grams do will make you a better programmer yourself, for you'll
learn a number of new techniques and tricks of programming. You'll
also find it easier to modify other programs once you understand
what they do and how they do it.

   When you find yourself interested in a particular subject—
TAPE, for instance—you can look up the subject in the keyword
index in the back of this book. There you'll find all the locations and
routines used for that subject. By reading the description of these
locations and routines, as well as their many possibilities and some
little-known facts, you'll become knowledgeable about the subject.
These locations and routines will refer you to other related locations,
routines, and appendices. The keyword index also lists the location
of many sample programs and statements, as well as tables and
diagrams. An additional index is provided to help you locate a mem-
ory location quickly once you're familiar with the mnemonic label
associated with that location.

   To make full use of the memory map, you should have a work-
ing knowledge of BASIC, but extensive experience isn't really nec-
essary. The BASIC routine descriptions offer insights into the quirks
and peculiarities of each BASIC keyword. The Kernal routine
descriptions become more technical and describe the way the Kernal
handles the input and output tasks on the VIC-20. The Video Inter-
face Chip (VIC) description is thorough and explains how to use the
computer's namesake chip to customize such things as screen colors

and size, musical tones, and characters of your own design. Other advanced features, such as multicolored characters, alternate screen flipping, and bitmapping are also explained. The section on the Versatile Interface Adapters (VIAs) describes the various connectors on the back of the VIC-20, what these connectors are designed for, and how to control and use them from within your own programs.

The ROM-based BASIC and Kernal routines can be disassembled with any of several available disassembler programs, and the memory map will explain the purpose of the machine language routines.

The program and statement examples included in the book are designed to be straightforward and understandable, not tricky. Extra levels of parentheses, additional spaces, and single statements per program line can be compacted to use less of the VIC's precious memory.

Even though the memory map is comprehensive, new techniques, tricks, and inventive methods will be constantly discovered. You may find it handy to note these at the appropriate memory locations when you discover them, insuring that this single-source reference is always up-to-date with your knowledge of the VIC.

You'll find plenty of programming techniques in this book. You'll learn how nonrelocatable program tapes can be relocated, how to call BASIC and Kernal routines from a BASIC program, what the effect of a shifted space is in a disk filename, and how the Kernal prevents you from saving memory above a certain location to tape. You'll also learn why x-line handshaking on RS-232 devices doesn't work properly, how to make a custom character set that gives the appearance of a 40-column screen, how to SAVE your data from within a BASIC program and LOAD it wherever you want, and other valuable programming methods. I think you'll find this book an invaluable guide for working with your VIC-20.

I wrote this book because I needed it. The VIC-20 is a home computer with lots of potential, but there was so little documentation available at first that the only solution was to start collecting memory location usage gems as though they were rare postage stamps. How can you write a machine language program if you have no idea where anything is in the computer? The *VIC-20 Programmer's Reference Guide* was somewhat helpful when it became available much later, but not to the extent that I felt necessary to effectively use the VIC-20.

For years I've collected, sorted, experimented, expanded upon, and cross-referenced the VIC-20 memory locations and routines. Every article, book, program, idea, and explanation that I could find through daily trips to the computer stores and many long distance conversations with other VIC-20 users was grist for this mill. I disassembled BASIC and the Kernal. Programs were written to cross-reference, sort, and label the operands of that machine language

viii

code. Every instruction was examined and evaluated. PET literature was pored over, and more recently Commodore 64 documentation was digested for possible insights. No Commodore proprietary documentation was available to me in this endeavor. My primary source of information (besides the VIC-20 itself) was the manuals and magazine articles available on the retail bookstore shelves.

Acknowledgments

simply couldn't have accomplished this massive task. I am in her debt, and admire her for the strength and conviction she has shown in the face of adversity.

VIC—20 Architecture

The main body of this book is organized by memory location for convenience of reference. Because of this organization, you may encounter unfamiliar terminology or concepts in the description of a particular location.

This overview explains many of the common VIC-20 terms used throughout the book, and gives you an idea of the VIC-20's architecture.

If you're familiar with the architecture (at least at the level described in the *VIC-20 Programmer's Reference Guide* or similar reference), you may wish to skip this overview and immediately use the memory location map. A block diagram of the unexpanded and expanded VIC-20 memory structure follows this overview. The diagram will be helpful to even the most knowledgeable reader as a source of quick configuration information.

Logic elements. The primary computing logic in the VIC-20 is implemented by the *microprocessor,* a single semiconductor component, or chip, designated the 6502. The 6502 is a popular microprocessor used (in some form) in most Commodore computers, in all Atari computers, in the Apple I and II, and in several less well-known machines.

The 6502 microprocessor implements the VIC-20's *instruction set*: the internal commands issued by machine language (ML) programs. It also contains a set of *registers*: high-speed internal storage locations used by most of the instructions. The most important 6502 registers for the VIC-20 programmer are:

- .A (Accumulator for arithmetic operations)
- .X (X-Index for indexed addressing)
- .Y (Y-Index for indexed addressing)
- .P (Processor status bits for conditional branching)

Two 6522 *Versatile Interface Adapter* (VIA) chip components provide timers, shift registers, and data ports for most VIC-20 Input/Output (I/O): the process of reading or writing data to connected devices. A VIA interface is used for keyboard input and for I/O to the principal VIC-20 ports:

- Game port used for joysticks, paddles, and so on
- User port for modems and custom I/O devices
- Serial port for VIC disk drives and printers
- Tape port for the VIC Datassette

The registers used to control the 6522 VIA functions are unknown to the 6502, but are mapped into memory locations for reference by programming instructions.

Output of characters, graphics, colors, and sounds to a video monitor (or RF modulator for a TV set) is an extremely complex function that requires custom logic to produce the proper signals. This logic is implemented on the *Video Interface Controller* (VIC): the chip from which the computer gets its name. Two model numbers are used for the VIC chip, depending on the video signal standards of the country in which the VIC-20 is sold. For the United States and Canada, the NTSC standard prevails and is supported by the 6560 VIC chip. For most European and Asian countries, the PAL video standard requires the 6561 VIC chip. This difference has little effect on programming, which accesses the VIC registers through locations mapped into memory, as in the case of the VIA registers.

Memory. The 6502 microprocessor reads a memory range of locations (address space) from location 0 to location 65535. Each location is composed of eight *bits* (BInary digiTs, each containing a 1 or 0) that form a *byte*. Bytes are grouped into *pages*, each 256 bytes long, which begin on a location that is an integral multiple of 256. Some addresses, called *pointers*, are actually two bytes long. These 16-bit addresses consist of an 8-bit page address byte preceded by an 8-bit *offset* within that page. This format, using the least significant byte (LSB) preceding the most significant byte (MSB), is often referred to as low byte/high byte.

The terms *vector*, *pointer*, and *link* describe the same type of two-byte memory location containing an address of another location in memory. The terms have subtle differences of meaning that are of more interest semantically and categorically than practically. Quoting Jim Butterfield: "The term *link* is used when the address is normally used to connect adjacent code; in this case, it doesn't affect the program flow until the link is broken with a new address. A *vector*, on the other hand, is used as a jump point, and the normal program jumps somewhere else through the vector. In other words, a ROM program hits a link point and normally keeps going; it hits a vector point and branches out."

A link is there in case you want to change where the routine normally goes, and normally points to the next sequential instruction. A vector is where the routine will be branching to. When a particular address is referred to by one of these terms in other documentation, that term is also used in this map. Therefore, you'll find *VECTOR* tables, screen *POINTERS* and *LINK* address captions for consistency's sake and to aid those readers who are familiar with each term. Fundamentally, each is two consecutive bytes that represent an address somewhere in memory.

Unless otherwise noted, these two bytes are in LSB/MSB (Least Siginficant Byte/Most Significant Byte) format. As already mentioned, this format is also referred to as low byte/high byte, but there are yet other terms used, such as LB/HB, LO/HI, and displacement/page found in other works. If you are unfamiliar with all of these terms, see Appendix A for an explanation of the LSB/MSB format and other related number systems topics.

The term *offset* is used to describe the distance from the beginning address of something in memory. The beginning has an offset of zero, the next byte has an offset of one, and so forth. *Index* usually means the number of an entry within a table. The first table entry has an index of zero, the second entry has an index of one, and so on. A table entry is typically the same length as all other entries in the table. Tables that have variable length entries are a bit more complicated and are explained as they occur within the VIC-20 memory.

Most eight-bit microprocessors have a similar-sized address space (65535 / 1024 = 64K bytes). The VIC-20 is sometimes unfairly compared to other microcomputers as having 5K of memory in contrast to 16K, 32K, or even the 64K of its big brother, the Commodore 64. Actually, the 64K address space on the VIC-20 is filled with four types of memory locations:

● Random Access Memory (RAM), that can be read or written by the program
● Read Only Memory (ROM), containing static data that can only be read
● Input/Output (I/O) Registers, mapped into memory banks
● Expansion Banks, for accessory RAM or ROM

In the unexpanded VIC-20, there are 5K bytes of RAM, 1K of which is used by the VIC chip for screen color codes; 4K of ROM containing pixel maps of the standard character sets; 8K of ROM for the BASIC interpreter; and 8K of ROM for the Kernal operating system service routines. Also, 3K of memory is consumed (but not filled) by I/O registers. This leaves empty *expansion blocks* of 3K, 8K+8K+8K (contiguous), and 8K (discontiguous with other expansion blocks). Data, addressing, and control lines for all five expansion blocks are routed to the VIC-20 *expansion port* to allow the interfacing of memory expansion boards or ROM cartridges.

BASIC. If the native language, for internal purposes, of the VIC-20 is that of the 6502 microprocessor, there's no doubt that it speaks to the rest of the world in a dialect of Microsoft BASIC. Even the operating commands, such as LOAD, SAVE, RUN, and so on, are actually BASIC statements.

A ROM-resident screen editor and BASIC interpreter are normally activated when the 6502 receives a *reset* signal at power on.

Control and data areas in *page 0* are initialized and the keyboard is questioned for input. Whatever you type on the keyboard is translated into BASIC *tokens*, a process usually called *tokenization.* In *direct mode* (entered from the keyboard as opposed to *program mode*, resident within a BASIC program), the tokenized commands are passed directly to the BASIC interpreter and executed immediately. Keyboard-entered commands preceded by line numbers are saved for later execution in what is usually called the BASIC program storage RAM (a contiguous area of memory addressed by a pointer in page 0).

BASIC stores a command such as PRINT as a one-byte representative of that word in the BASIC program storage area. This technique tends to best use program storage space and also helps BASIC execute faster. Reducing a BASIC word to this one-byte shorthand is called tokenization, and the one-byte character is known as a token. When the program is LISTed, BASIC converts it back into a BASIC keyword. See Appendix B for the description of the internal format of BASIC statements and variables. Appendix C has a code chart that shows the one-byte values used for BASIC tokens. Harvey Herman in *COMPUTE!'s First Book of PET/CBM* discusses tokens in his article "Tokens Aren't Just for Subways." It may be something you'll find of interest if you want further reference on tokens and tokenization.

In program mode (when a program is executing) the screen editor can be used to obtain input from the keyboard or screen, to display that information, and to manage any program output going to the screen.

A user machine language routine called a *wedge* can insert itself into the vector of addresses of routines given control by the Kernal keyboard monitor. A wedge can examine each command, decide whether it wishes to interpret that command, and, if not, pass it on to other wedges or the BASIC interpreter.

A final point of architectural interest implemented in the BASIC interpreter is *floating point* arithmetic. The only arithmetic functions implemented on the 6502 microprocessor are fixed-point binary and decimal addition and subtraction. The BASIC language further defines multiplication, division, and exponentiation and has fixed- and floating-point binary data types. This arithmetic is simulated by BASIC ROM routines that define floating point *accumulators* in low storage. Even machine language programs frequently use these ROM routines for arithmetic.

Kernal. Another set of ROM routines in the VIC-20 provide a common set of services to the BASIC interpreter, user BASIC programs, user machine language programs, and any other type of programs that happen to be running in the computer. These Kernal services are similar in most Commodore computers. Some are

addressed indirectly through vector references whose locations are also common. The pointers, control blocks, and other data areas defined in pages 0–3 by Kernal routines are less common among Commodore machines but are quite compatible between the VIC-20 and Commodore 64.

The services provided by Kernal routines are in several categories. The most obvious and most widely used is I/O device support. The 6522 VIA I/O functions are tedious to program, involving a great deal of time-dependent bit manipulation, so nearly all programmers take advantage of the Kernal routines to accomplish byte- or block-oriented I/O functions. These functions are discussed later.

The screen editor used by BASIC and most user programs for keyboard and screen output is also provided by the Kernal. On input, the screen editor provides the familiar support for the INSert and DELete key, the cursor positioning keys, and the RETURN key. On output, the line-wrap feature and the interpretation of cursor and color control characters in the output stream are the responsibility of the screen editor. Most software uses the screen editor for character I/O to the keyboard and screen, so you tend to be most familiar with its functions. Exceptions for input are the BASIC GET command and the use by many word processors of alternate input editing conventions. For output, the most visible exceptions are games and other graphic applications that generate screen output by POKEing or otherwise updating the areas of storage that define the display.

The Kernal also provides monitoring services for the various kinds of interrupt conditions generated within the VIC. One of the most significant interrupts is the IRQ JIFFY, which controls a Kernal monitoring routine pointed to by a vector in RAM at designated time intervals, called *jiffies*. Routines chained to the jiffy vector are used for almost all asynchronous (can-happen-at-any-time) activity in VIC-20 programming.

Other Kernal functions include: autostart of plug-in ROM cartridges, memory initialization and management, and some screen-oriented service routines that assist in writing portable or translatable programs. The memory map contains a complete description of all Kernal services.

Front-ends, Preambles, and Wedges. Many times a Kernal or BASIC routine could perform some additional work for you, if you could just add some instructions to it. The routines are located in ROM and can't be changed, but by adding instructions to the beginning of the routine, you may be able to add to or change it. This addition of instructions is possible when a vector in RAM points to the beginning of the routine, and this vector is used by other BASIC or Kernal routines to get to the routine. The vector can be changed

to point to your own routines, which will perform the functions you wish, then go on to the original ROM routines. These routines placed before the standard routines are called *front-ends, preambles,* or *wedges.* The VIC-20 contains vector tables for many of the BASIC and Kernal routines. These can be modified to point to your own intercepting routines.

Input/Output. On the VIC-20, I/O is accomplished in many different ways. This can be confusing, compared to architectures which have a more regular I/O structure, but is necessary to the efficient functioning of the machine.

Devices attached to the serial port, such as the Commodore disk drives and printers, are presumed to be *intelligent* at implementing high-level control functions. The Kernal routines which support the serial port must contend with idiosyncrasies of the 6522 VIA, such as the need to shift data in and out bit by bit, but don't need to know anything about the device. The VIC 1540/1541 disk drives, for example, have their own 6502 microprocessor and their own RAM and ROM containing a high-level set of disk management logic called DOS (Disk Operating System).

At the opposite extreme, the tape port device is an extremely low-level analog electrical attachment. The Kernal routines which perform tape I/O must literally detect and generate the time-dependent changes in recording levels that represent the bits on the tape, as well as higher level data management functions.

User port and game port I/O can have aspects of either extreme. The only Kernal routines in direct support of the user port simulate a 6551 UART (Universal Asynchronous Receiver/Transmitter) chip's clocking and handshaking logic and control registers in support of devices such as modems and printer interfaces that use a protocol for I/O similar to the RS-232 standard. The 6551 UART was not actually included in the VIC, so the software simulation of its logic and registers was required in the Kernal as a way of supporting the other ROM and package software that assumed those functions. (Note that the user port does *not* provide electrical compatibility with the RS-232 standard, either with voltage levels or connector pinout; external drivers and connector adapters are required to attach RS-232 devices to the VIC.)

Some I/O is directed into memory in usable form by the 6522 VIA. For example, switch-type joystick input can be received directly by the program. Also, keyboard input can be directly observed while a key is held down. Kernal IRQ routines stabilize the keyboard input to prevent unwanted multiple keystrokes, move the key code to a low-storage location, translate the key code according to any SHIFT, CTRL, or Commodore keys depressed, and append it to a keyboard input buffer for reference by higher-level input routines.

The last I/O interface on the VIC-20, the video (and also sound) port, is such a fundamental part of the machine that it deserves its own section.

VIC Chip. If the 6502 microprocessor is the brains of the VIC-20, memory its skeleton, and the 6522 VIAs its eyes and ears, the 6560/6561 VIC chip is really its heart. No understanding of the VIC-20 architecture can be complete without some knowledge of the VIC chip functions.

As mentioned previously, the VIC chip is responsible for generating the composite video and sound signals that are sent to a color display monitor or RF modulator for transmission to a television set. Normally, the VIC chip maps the display screen into an array of *pixels* (picture elements, which are dots that are the smallest units of video information) that is normally 176 (22 x 8) pixels wide and 184 (23 x 8) pixels high. It also displays a *border* and a *background* of designated colors.

Color, mode, and other control information is supplied to the VIC chip via its 16 registers, which are mapped into memory locations as described in the memory map. Three other memory areas, addressed through the VIC chip registers, are used by the VIC chip to generate character or graphic displays.

● *Screen memory*, sometimes called the Video Matrix, is an array of memory with a byte for each character position on the screen (normally 506 [22 x 23] character positions). Each byte in screen memory contains the offset in character memory.

● *Character memory* (usually one of the banks of character ROM) of the bit map of the pixels to be displayed at the corresponding screen position. Usually, control information in the VIC chip registers has the VIC chip in single color mode. In this case, each bit in character memory selects whether the corresponding pixel color is to be the same as the background color (pixel invisible or *off*) or to be a color selected by the code contained at a position in color memory.

● *Color memory* corresponding to the screen location of the character.

Custom character sets can be designed and placed into 8 x 8 or 16 x 8 patterns in RAM addressable by the VIC chip (only built-in RAM) and then used by selecting that RAM in the VIC chip registers. What is usually called *medium-resolution* graphics can be developed using the same technique, with custom characters for the graphic images. *High-resolution* graphics on the VIC-20 are created by initializing screen memory to simply point to successive locations in a bitmap of the screen addressed as though it were character memory by the VIC chip.

Multicolor mode uses two bits in character memory to color each pixel, so it cannot make reasonable use of the built-in character

ROM. The two bits select among the background color and the character memory color for that location, as in single color mode. The border color and an *auxiliary* color are also defined in the VIC chip registers when multicolor mode is selected.

Some little-known functions of the VIC chip are the *light pen* position detection, which can be used by the program to determine the screen position in front of a light pen and *resistance value sensing*, for use in reading up to two peripheral devices, such as game paddles, that contain potentiometers for sensing or measuring position. These may also contain other variable resistance-sensing devices, such as photo-resistors or carbon microphones, for special applications.

The final function of the VIC chip is *sound*. Five of the VIC chip registers mapped into memory control sound volume, the frequency of each of the three tone generators (voices) and the noise generator.

Common Plug-In Cartridges. A wide variety of VIC-20 software is available in plug-in cartridge form. A cartridge is simply a means of adding memory, usually ROM, in one of the expansion blocks. Most of the software is application programming of one kind or another, like games, word processors, and so on, and does not affect the architecture. Several plug-in cartridges, though, are really intended as architectural extensions or supplements:

- Programmer's Aid—extensions to BASIC editing commands
- Super Expander—graphic and sound supplements to BASIC
- Machine Language Monitors—replacing the BASIC editor
- Other language monitors—that also replace BASIC

Compatibility with the Commodore 64. With attention to some variations in the memory map, it's possible to write software for the VIC-20 that can be readily translated to the Commodore 64.

This is possible because of a number of similarities between the two machines. The 6510 microprocessor used in the 64 has the same instruction set as the 6502. Its only significant extension is that a memory-mapped I/O port is added. The BASIC dialect on the Commodore 64 is identical to that on the VIC-20. Indeed, purely BASIC programs, written for the VIC-20 with no PEEKs and POKEs, should run correctly with no modifications on the 64 if statements are restricted to 80 characters, instead of the 88 allowed on the VIC-20. The Kernal service routines are fundamentally the same on both machines. Finally, much of the low-storage data defined in pages 0–3 is identical, with key differences noted in the memory map.

However, the 6560/6561 VIC chip display, graphics, and sound functions are implemented much more elaborately on the 64 with a 6566/6567 VIC-II chip and a 6581 SID (Sound Interface Device) music synthesizer chip. These new devices are not simply compatible supersets of the VIC chip, either in function or in programming.

The VIC-II chip has much more extensive graphics capability, but changes the function and location of its control registers. Programs that are intended to be translated to the 64 must isolate the logic that manipulates the VIC chip registers. Also, the VIC-II chip defines a larger normal screen (25 lines of 40 characters), so a program that POKEs directly into screen or color memory must take care to use system-defined data areas and service routines to discover the location and dimension of such memory.

The SID chip provides a high degree of control over the waveform and frequency of its several voices. However, it is quite complex to program compared to the VIC chip's voices. Even to duplicate these simple sound functions takes quite a bit of programming, so the audio effects of game programs, for instance, probably will need to be completely redesigned as well as reprogrammed when translating to the 64. Again, it would be wise to attempt to isolate the statements that control sound effects.

Actually, most programmers would agree that the incompatibilities in architecture introduced on the Commodore 64 were justified by its additional functions. Other home computer vendors have maintained more compatibility among their models but sacrificed the opportunity to enhance function, particularly in such areas as graphics and sound.

Conclusion. With this understanding of the overall VIC-20 architecture, you can now jump into the memory map and its cross reference to find out how to make the VIC-20 do amazing things for you!

Block Diagram of All VIC-20 Addressable Memory

No Expansion	3K Expansion Differences	8K–32K Expansion Differences
Decimal Hex		
65535 - - - $FFFF	....	....
8K Kernal ROM		
57344 - - - $E000	....	....
8K BASIC ROM		
49152 - - - $C000	....	49152 - - - $C000
		(User Area)
(8K Expansion)		(for POKEs,ML)
(RAM / ROM)		(if this 8K)
((autostarted))		(is added)
40960 - - - $A000	....	40960 - - - $A000
I/O EXP. B3		
39936 - - - $9C00	....	....
I/O EXP. B2		
38912 ***** $9800	....	....

xviii

Column 1:

* Color RAM *
38400 ***** $9600
I/O EXP. B0
37888 - - - $9400
VIA 1/2
37136 - - - $9110
VIC 6560
36864 - - - $9000
Character ROM
32768 - - - $8000

(8K Expansion)
(RAM / ROM)

24576 - - - $6000

(8K Expansion)
(RAM / ROM)

16384 - - - $4000

(8K Expansion)
(RAM / ROM)

08192 ***** $2000
* Screen Area *
07680 ***** $1E00
User PGM Area
3584 RAM
04096 - - - $1000
(3K expansion)
(RAM)
01024 - - - $0400
BASIC / Kernal
1K Work Areas
00000 - - - $0000

Column 2 (center):

....
....
....
....
....
....
....
....
....
....
User PGM Area
6656 RAM
01024 - - - $0400
....

Column 3 (right):

38400 ***** $9600
* Color RAM *
37888 ***** $9400
....
....
32768 - - - $8000

(User PGM Area)
(28160 if +24K)

24576 - - - $6000

(User PGM Area)
(19967 if +16K)

16384 - - - $4000

(User PGM Area)
(11776 if +8K)

04608 ***** $1200
* Screen Area *
04096 ***** $1000
(3K non-BASIC)
(when filled)
01024 - - - $0400
....

Format of the Map Descriptions

Locations and routines in this book are presented in this format:

Decimal	$Hexadecimal	Label	Values
		(Notes)	

Title of Location or Routine
Explanation of location or routine usage by all routines of
BASIC and the Kernal with suggestions for user usage or testing.

Decimal. The decimal location range (for PEEKs and POKEs).

$Hexadecimal. The hexadecimal location range preceded by $ for machine language (ML) users.

Label. Mnemonic label or an invented label followed by *. Labels are easier to remember and identify than the decimal or hexadecimal address. A label-to-address cross reference index is included in the back of this book.

(Notes). (*User Storage*) indicates locations that can be used for user storage without drastic complications. Be sure to read the complete description of the location for possible dependencies by BASIC or the Kernal.

(*Possible User Storage*) flags locations that are available for your use when the listed functions are not being performed by the Kernal or BASIC.

(*Handy Location*) flags locations that are helpful to the BASIC or ML programmer in obtaining information or causing actions that are unavailable through BASIC keywords. Every memory location may be useful to someone, for use in some particular situation. This flag is placed on those locations that have the greatest potential for the average user. Many of these Handy Locations include short sample programming routines to demonstrate their use.

Values. The contents of the location on an *unexpanded* VIC-20 in decimal and hexadecimal. Included if the contents are predictable. As in the rest of the book, the hexadecimal location is separated from the decimal by parentheses.

Explanation of Location or Routine Usage. Here you'll find a description of the purpose(s) of the location or routine, along with any considerations, relationship with other locations or routines, tips and hints, further information references, and sample routines to explore this location more fully.

Chapter 1

Memory Page 0

Memory Page 0

VIC-20 memory between location 0 and location 255 is referred to as *page 0* or *zero page*. A *page* on the VIC-20 is a chunk of memory 256 bytes long. Page 1 is the chunk between 256 and 511, and correspondingly page 4 refers to locations 1024 through 1279. Since the VIC-20 has a memory range of 0 through 65535 (a total of 65536 bytes), there are 256 pages of memory. The term *page* is used as a shorthand method of referring to a range of memory, and you'll primarily see it when page 0 or page 1 memory locations are discussed. For more information about the concept of pages, see page 114 of *The VIC-20 Programmer's Reference Guide*.

Page 0 is an important section of memory, as many VIC-20 machine language (ML) programmers will confirm. Page 0 locations are required by some of the handiest ML instructions, and instructions can be written with shorter and faster operands when using page 0. When bypassing the facilities of BASIC or using only a portion of it, many page 0 locations can be freed to be used as the ML programmer desires.

To the BASIC programmer, page 0 is also important since both BASIC and the Kernal store some of the most intriguing and useful pieces of information in these memory locations. Knowing what has been placed here and how to use this information can open up a whole new dimension of possibilities for customization and control of the VIC-20.

Let's begin with a look at the portion of page 0 that is free of the Kernal and of which BASIC takes charge.

Location Range: 0–143 ($0–$8F)
BASIC Working Storage

Page 0 working storage for BASIC.

INITMEM* routine initializes this area to zeros at *power-on* or *reset*. (Some memory expansion socket boards have a reset switch. See location 45 for a description of how to make your own.) A follow-on routine (INITBA, a BASIC cold start routine) then initializes any locations with values that BASIC needs. See the comments in each location for initialization values.

0	$0	USRPOK	76 ($4C)
		(possible user storage)	

The INITBA routine initializes this location during power-on/reset.

Used with USR vector at location 1-2 ($1-2). BASIC interpretation of the keyword USR causes a branch to this location (via the FUNDISP table). This JMP opcode then causes the vector in 1-2 ($1-2) to be branched to, ending in the programmer's USR routine. You may use this byte for your own purposes, but POKE 0,76 before attempting USR.

The Commodore 64 uses location 784 ($310) for a JMP opcode, and 785-786 ($311-312) for the USR vector. Locations 0-1 are wired into the chip for memory control on the 64, while location 2 is unused.

1-2 $1-2   ADDPRC   72/210 ($48/D2)
(possible user storage)

The USR jump vector in LSB/MSB (displacement/page) form.

Before using the USR keyword, you POKE the LSB/MSB address of the desired ML routine in these locations. For example, if an ML routine at 828 is the target ML routine (the tape buffer which begins at location 828 is traditionally a favorite location) POKE 2,INT(828/256):POKE 1,828−(PEEK(2)*256):X=USR(expression) sets location 1-2 to 60/3. 60 is the Least Significant Byte, 3 the Most Significant Byte. In hex, that's $3C/03 (normally expressed as $033C), which is 828 decimal.

If the preceding seems mysterious to you, you may want to review Appendix A which discusses number systems and hex/binary/decimal conversion methods, charts, and programs.

The evaluated expression result is placed into the floating point accumulator at location 97-102 ($61-66) in floating point format by BASIC. When your ML routine is executed, it converts the floating point accumulator back to a two-byte integer with a JSR to $DC9B. Locations $61-62 then contain the two-byte integer (MSB/LSB).

When the ML routine is ready to return to BASIC, it loads the .Y register with the LSB of the return number, and .A with the MSB. JSR $D391 then converts these in the floating point accumulator found at 97-102 ($61-66). When the ML routine issues the RTS instruction, BASIC assigns the floating point contents of the floating point accumulator to the variable assigned by the USR statement (X in our example). Once you have become familiar with the use of locations 780-783 ($30C-$30F), which are the register SAVE area for SYS, this process of number conversion can be done in BASIC.

At power-on/reset, locations 1-2 are loaded with a vector that points to the ERROR routine, the BASIC error message handler. This will cause an ILLEGAL QUANTITY error message if USR is issued before 1-2 have been altered by POKEs to the desired ML routine. Once you have changed the default vector, the ILLEGAL QUANTITY message will not appear. If the vector does not point to valid

6502 instructions, the VIC-20 will probably *hang* until you reset it or turn it off and on.

You may use locations 1–2 for your own purposes if you are not using the USR command.

On the Commodore 64, this function has been moved to locations 785–786 ($311–312). Keep this in mind if you are writing programs that are to run on both machines.

For an in-depth discussion of the potentials of USR, see "How to Use SYS and USR," by J.C. Johnson in the November and December 1982 issues of *COMPUTE!*, or "PEEK and POKE: A USR Instruction Sheet," by George Gaukel in the May 1983 issue of *Commander*, for a clever technique of extending language functions with the USR command.

3–4 $3–4   ADRAY1   170/209 ($AA/D1)
(user storage)
Vector to floating point/integer conversion routines.

Initialized during power-on/reset to routine INTIDX*, which is at location 53674 ($D1AA). It appears that BASIC (and the Kernal) never use this vector. Nonetheless, you may want to take advantage of this vector and use it as a level of protection against future ROM changes.

5–6 $5–6   ADRAY2   145/211 ($91/D3)
(user storage)
Vector to the integer to floating point conversion routines, starting at MAKFP.

The power-on/reset routines initialize this location. No reference to this location is found in BASIC or the Kernal. You can also use this vector to protect against future ROM changes.

7 $7   CHARAC
Search-character for scanning BASIC statements.

This is a busy and critical location, used in conjunction with location 8 ($8) by BASIC routines that scan the BASIC input buffer at 512 ($200), looking for line ends, colons, quotes, commas, and other special characters.

Typical values here are the ASCII codes for colons commas, quotes, or zeros. String handling routines may use this location during the scan of BASIC statements in locations other than the BASIC buffer. A pointer to that area will be set in 111 ($6F), which is used as a temporary string pointer holder.

Still other routines (AND, OR, DECBIN) use these locations (7–8) for work areas for their processes, since they do not interfere with scanning.

8 **$8** **ENDCHR**

Scan-quotes flag for scanning BASIC statements.

 See the description of location 7 ($7).

 This is also used during the tokenization of a BASIC statement.

9 **$9** **TRMPOS**

            (possible user storage)

Column that the cursor was on just before last TAB or SPC.

 This is a logical, not physical, column and can range between 0 and 87 since there are 88 columns on a logical line.

 TAB or SPC obtains the cursor column from the Kernal, which uses locations 211 ($D3), cursor displacement within the screen RAM line, and 213 ($D5), logical line length. TAB or SPC stores the cursor column at this location and uses it to calculate the target cursor location.

 This location may be used as desired between SPC and TAB functions.

10 **$A** **VERCHK**

            (possible user storage)

Tape: 0=LOAD, 1=VERIFY

LOAD sets this byte to 0 and VERIFY sets it to 1.

 LOAD and VERIFY are similar processes, handled within the same routines, and they use this location to determine which process is occurring.

 This value is passed to the Kernal routine LOAD, which saves it in 147 ($93), a LOAD/VERIFY switch byte, for its own use.

11 **$B** **COUNT**

Buffer index/array dimensions.

 This location is used as an index into the BASIC input text buffer at 512 ($200) for tokenization and store-the-line tasks. When the line has been fully tokenized, this location holds the length of the tokenized line, including four bytes for a line number and link address. See the BASIC routine NEWLIN at 50332 ($C49C).

 Array processing, such as building an array or locating an array item, uses this location to calculate the size of an array descriptor. It also calculates the number of subscripts specified when referencing an item, and the number of dimensions in an array definition DIM. For example, DIM X(5,10,15) would set this location to 3.

 Both AND and OR use this location. OR sets this location to 255 ($FF), while AND sets it to zero.

12 $C   DIMFLG
Flags for locate-or-build-array routines.

Various temporary flags are set and used by the Kernal to signal to itself whether the variable is an array, whether it's numeric or string, whether the array has already been DIMed, and if a new array is to be the default size.

The coding of routine ARY5 allows redefinition of an array, such as DIM X:X(5)=2:DIM X, without getting the REDIM'D ARRAY error message or affecting the contents of X(5) as long as you don't specify a different size for the array.

13 $D   VALTYP
(handy location)
Type of variable: 255 ($FF)=string, 00=numeric

BASIC routines set this location to indicate the type of variable being processed. FRMEVL (expression evaluation) routines set this for every variable located or created.

Additional locations that are helpful when evaluating variables are 14 ($E), 69 ($45), and 71 ($47).

14 $E   INTFLG
(handy location)
Numeric variable type: 128 ($80)=integer, 00=floating point

This is not meaningful unless location 13 ($D) is set to zero, indicating a numeric variable.

See Appendix B for details of the format and range of both types of numeric variables. FRMEVL (expression evaluation) and associated routines also maintain this location.

15 $F   GARBFL
Flag byte: LIST quote/collect done/tokenize character.

LIST uses this byte as a flag to prevent detokenization of a character string within quotes.

This location indicates that garbage collection has already been tried while adding a new string, resulting in an OUT OF MEMORY error message.

It is also used as a work byte while tokenizing a line in the BASIC text buffer at 512 ($200) to prevent DATA items from being tokenized.

16 $10   SUBFLG
Subscript or FN X flag byte.

The BASIC routine EVLVAR (locate or create a variable) uses this byte as a flag to remind itself that parentheses ( ) are allowed in

the current BASIC statement. Parentheses are allowed if a subscripted variable or an FN command is in the statement.

When FN is defined, this location is set to 128 ($80) so that EVLVAR will know to create or replace a function definition.

A function can be replaced by redefining it. For example:

DEF FNZZ(Q)=2*Q:X=FNZZ(66):DEF FNZZ(S)=22/X

replaces the function named ZZ.

FN redefinition never frees any memory since FN commands are stored in the BASIC statement which includes the DEF FN, as well as seven bytes for the descriptor pointing to them.

The name given to a function can be the same name as an existing or future floating point variable. For example:

DEF FNB(A)=A/22:B=FNB(B)

is completely valid.

17 $11 INPFLG
Indicates which of READ, INPUT, or GET is active.

0=INPUT; 64 ($40)=GET; 152 ($98)=READ

This location is used by the READ routine, which includes common instructions for all three keywords, to determine which sections of code to execute. Some of the differences in these routines are: show prompt and ? for INPUT; only obtain one character for GET; echo back INPUT characters; and allow colon and comma as valid data for GET but treat them as delimiters for READ and INPUT.

This location is also used to determine the appropriate message in case of an error. For example, it will show the line number of a DATA statement if READ data is bad; REDO FROM START or FILE DATA ERROR will display if INPUT data is bad; and EXTRA IGNORED or no message will show depending on the active channel number specified in location 19 ($13).

18 $12 TANSGN
TAN/ SIN sign/comparison results.

SIN and TAN use this byte to determine the resultant sign.

The formula evaluation routine stores at this location the returned results from math operator routines it calls.

String comparison routines also save their results here.

During comparison of variable A to variable B, the value saved here is 1 if A>B, 2 if A=B, or 4 if A<B. Combinations of these values may also be here, if the routine that evaluated A and B performs more than one comparison. For example, if 5 was stored here, it would indicate A<>B. The AND and OR routine does not set this location.

19   **$13**   **CHANNL**
(handy location)

Current channel number for BASIC input/output routines.

This is a *significant* and active location for BASIC. Whenever BASIC needs to talk to a device, it looks here to determine the correct channel number.

See location 153 ($99), current input device number, for an explanation of the process BASIC uses to tell the Kernal what the input channel number should be.

Altering this location can make BASIC think that tape or disk is the keyboard, for example.

VIC-20 devices are:
0 keyboard
1 tape
2 RS-232/user port
3 screen
4–5 printer
8–11 disk
Device numbers 4–31 could be any type of serial device.

Let's see how this location is used by BASIC, so that you can use it to your advantage. If an error message is being displayed, the input device number is restored to zero, indicating the keyboard, so that error messages will appear on the screen, rather than the CMD device, with the necessary carriage return and linefeed.

The PRINT routine uses this location to test if carriage return and linefeed are needed.

CMD saves the new output file number here, then calls the Kernal to open that device for output, leaving it in a listen mode. PRINT# uses the same routine, but then closes the channel, taking the device out of listen mode, and resets this location to zero. DEVICE NOT PRESENT or FILE NOT OPEN may be displayed on the screen, but READY would appear on the new output device, as would any INPUT prompts. Hitting RUN/STOP–RESTORE while CMD is active to the VIC 1525 printer can hang the computer.

PRINT tests this location and normally uses cursor right characters for screen tabbing rather than spaces. You could alter this by changing this location.

If INPUT data is bad, FILE DATA ERROR is shown if this location is *not* zero. GET, GET#, and PRINT# will restore this location to zero when done, effectively cancelling any *active* CMD. INPUT# does the same thing. INPUT will also cancel if this location is not zero. Thus PRINT is the only I/O keyword that can be used without cancelling the CMD device. LIST also leaves this location as it was.

INPUT also accepts a carriage return from the keyboard and

shows it as a null string, but only if this location is zero. For any-
thing else, it gets the next piece of data and no prompt is printed.
READ, used for INPUT also, will print EXTRA IGNORED as an
error only if this is zero.

Power-on/reset routines set this location to 0, as does RUN/
STOP–RESTORE. Here are some suggestions for using this location:

● Tape, printer, or disk can be used as the CMD target device.
The file number must be between 1 and 255.

● Remember that zero is not a valid CMD file number, and that
CMD may have a string just as PRINT# can, but needs a comma
before it. For example, OPEN4,4:CMD4,"PRINT THESE WORDS".
That's because PRINT# and CMD use the same routines.

● Change this location to a number greater than zero to sup-
press the EXTRA IGNORED error message when inputting from the
keyboard. This POKE will also cause INPUT to not show a ? and
will ignore a carriage-return-only entry.

● See location 153 ($99), input device number, for instructions
for reading tape as though it were the keyboard.

● Reissue CMD after GET, INPUT#, PRINT#, LOAD or RUN.
When writing your program, you could also avoid CMD-cancelling
keywords and obtain keyboard input from the keyboard buffer (see
location 631, $277) by adding a front-end to the IRQ keyboard scan-
ner in location 788 ($314), or by examining location 197 ($C5) or
203 ($CB), matrix of last/current key pressed, and 653 ($28D), the
current shift pattern.

● See location 154 ($9A) for details of input/output diversion to
other devices using SYS and locations 780–783 ($30C–30F), the SYS
register SAVE area, or by using ML.

The following routine demonstrates a number of uses of this
location. This short program will suppress the question mark nor-
mally displayed in an INPUT command, leaving instead the cursor
on your chosen default character. It will also ignore a null entry,
made when the user presses RETURN without entering any charac-
ters, and will allow a null entry to represent an entry of the number
zero. Notice that ?FILE DATA ERROR will be displayed if string
data is entered for a numeric variable, rather than the usual message
?REDO FROM START.

Program I—I. Prompt Suppression

```
5 REM LOCATION 19 $13 %CHANNL
10 POKE 19,88:PRINT"?A=32{2 LEFT}"; :REM ";" KEEPS
     CURSOR ON DEFAULT, IGNORES NULL ENTRY
15 REMOVING ";" ALLOWS A NULL ENTRY TO BE 0
20 INPUTA
30 POKE19,0 :REM CLEANUP SO CR/LF PRECEEDS PRINT
```

```
35 REM PRINTCHR$(13)A : REM WITHOUT 13 WILL PRINT
   {SPACE}ON SAME LINE-2 SPACES AWAY FROM END
```

20-21 $14-15   LINNUM
(handy location)

Line number integer in two-byte LSB/MSB format.

Subject line number for GOTO, LIST, ON, and GOSUB is stored here, as well as the line number of a BASIC line to be replaced, added, or deleted by the BASIC line editor NEWLIN at 50332 ($C49C). The DECBIN routine puts the number here for all users.

By placing a BASIC line number here (LSB/MSB), SYSing to 50707, and examining the .P register in 783 ($30F), you can obtain the address of the line's link field in 95–96 ($5F–60) if the carry flag is set (bit 0 of .P is on). If the carry flag is not set, the line number doesn't exist in the program. This capability would interest advanced BASIC or ML programmers.

Here's a short program which does this:

Program 1-2. Line's Link Field Address

```
300 POKE 15,INT(LN/256):REM LINE NUM MSB
310 POKE 14,LN(PEEK(15)*256):REM LINE NUM LSB
320 SYS 50707: GO SEARCH
330 IF (PEEK(783)AND 1)<>1 THEN 500: REM NO LINE F
    OUND
340 LA=PEEK(96*256)+PEEK(95):REM POINT TO LINE
```

The LIST routine saves the highest line number to be listed, or 65535 ($FFFF) for all, in this location.

GOTO and GOSUB determine which way to scan to find the subject line by testing if the subject line number MSB is greater than the current line number MSB. If so, it scans forward; otherwise, it scans from the first line of the program. Try to minimize the search time in your own programs by being aware of the scan direction that will be used. A GOTO to the line that the GOTO is on would result in the maximum search time if it were the last line of the program.

Take a look at Figure 1-1 for some examples of search patterns.

Figure 1—1. GOTO/GOSUB Search Patterns

GOTO/GOSUB SEARCHING

```
100                   100                  100
200        GOTO800 200                     200
300                   300                  300
400                   400                  400
500                   500               ↓  500
600                   600         GOTO500 600
700                   700                  700
800                ↓  800                  800
900                   900                  900 ──────→ GOTO900
```
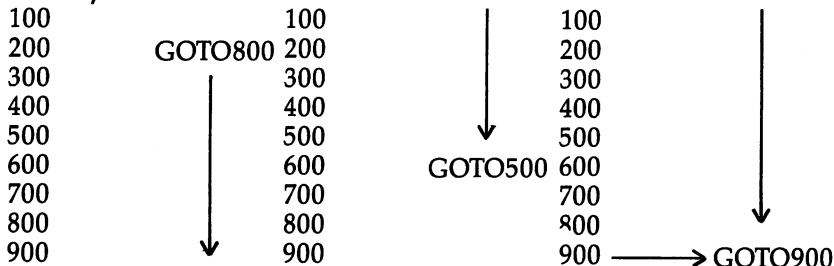
The effect of the MSB-only check is sometimes this*:

```
                              100
                              256
                              512
                              768
                              1024
                              1280
        Current line number → 1536           GOTO1791
                              1791        ↓
                              1792
                              2048
                              2304
```
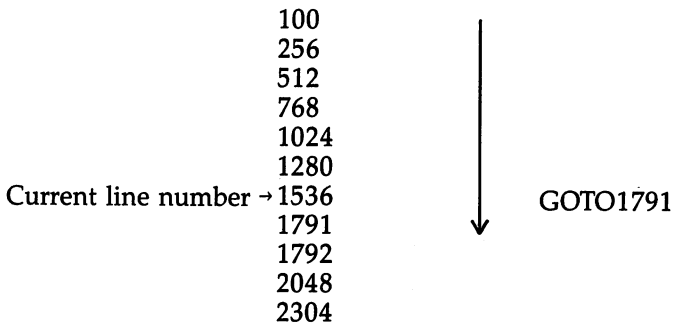
*Notice that the search starts at the beginning of the program even though the subject line number (1791) is greater than the current line number (1536). This is because the MSB of 1791 is not greater than the MSB of 1536.

Floating point to integer conversion MADADR, using FPINT to do the actual conversion, puts the output integer here.

PEEK saves the contents of this, uses it as a pointer to the subject byte, then restores the previous contents.

POKE, WAIT, and SYS use this as a pointer to the subject memory location.

22 $16   TEMPPT   25 ($19)

Pointer to available slot in temporary string stack.

The temporary string descriptor stack is at 25–33 ($19–21) and has room for three string descriptors of three bytes each. Possible values then are: 25 ($19) if empty and 28 ($1C), 31 ($1F), or 34 ($22) if full. An example of temporary strings:

PRINT MID$("abcdefgh",3,4)+" is cdef."

In this example, *cdef* and *cdef is cdef* are both temporary strings.

See 23 ($17), the last string descriptor used, for a related pointer.

The CLR routine resets this location back to 25 ($19), as does power-on/reset.

When trying to save a string descriptor in the temporary stack, if the value is 34 ($22), then the FORMULA TOO COMPLEX error message will be displayed. For other values, the descriptor is saved and this location's value is incremented by three.

The garbage collection routine checks this pointer to determine if there are temporary slots that should be cleaned up.

23-24 $17-18   LASTPT   22/0 ($16/0)
Pointer to the last string descriptor used in the stack.

Possible contents are 25 ($19) and 0 if only one string descriptor is used, 28 ($1C) and 0 if two are being used, or 31 ($1F) and 0 if all three have been used. If none of the slots is used, location 22 ($16) contains a value of 25 ($19).

25-33 $19-21   TEMPST
Descriptor stack for three temporary strings.

Each descriptor contains the length of the string, as well as the address, in LSB/MSB format of the string's beginning in the BASIC program line or string storage pool.

34-37 $22-25   INDEX
Miscellaneous temporary pointers and SAVE areas.

This area is used throughout BASIC to hold temporary pointers and calculation results. It's also used as a jump vector for the math operation routine address when formula evaluation has determined the correct routine to be used.

38-42 $26-2A   RESHO
BASIC multiplication work area.

These locations are used by BASIC multiplication and division routines, and by array creation routines for array size computations.

43-44 $2B-2C   TXTTAB   1/16 ($1/10)
Pointer to the start of the tokenized BASIC program.

If a byte containing zero does *not* precede the location pointed to, a SYNTAX ERROR message will be given when RUN is typed in. Power-on/reset routines place the zero here for you.

BASIC statements are stored internally as:

LL HL LN HN/tokens-and-characters/0
where LL HL = a pointer to the next line

LL= LSB of the address of the next line
HL= MSB of the address of the next line
If HL is zero, then this is the end of the program, since BASIC state-
ments would not be in zero page.
LN= LSB of the line number
HN= MSB of the line number
tokens-and-characters= the text of the BASIC line after it has been
tokenized. BASIC keywords, functions, and operators become one
byte each, ranging between 128 and 255. Characters, such as vari-
able names and strings, are not tokenized.
0=end of the BASIC line

You can examine the address contained here by entering PRINT
PEEK(43)+PEEK(44)*256

This pointer can be raised to reserve space for ML or for other
purposes, for instance, relocating screen or character memory. See
location 55-56 ($37-$38) for a program to adjust this pointer to
reserve memory for ML routines.

A program which will calculate the number of blocks required to
store a BASIC program on disk, as well as calculate the time it will
take to save it on tape, takes only a few lines.

Program 1-3. Disk Space and Tape Time

```
10 REM 43
20 REM THE NUMBER OF BLOCKS REQUIRED TO STORE YOUR
   BASIC PROGRAM{2 SPACES}ON DISK CAN BE
30 REM CALCULATED BY:
40 SZ=(PEEK(45)+PEEK(46)*256)-(PEEK(43)+PEEK(44)*2
   56)
50 BLKS=INT((SZ/254)+.5)
60 REM EACH BLK BEGINSWITH A 2-BYTE POINTER TO THE
   NEXT BLOCK
70 REM TO CALCULATE THE APPROXIMATE SECONDS REQUIR
   ED TO SAVE THE PROGRAM TO TAPE:
80 SECS=SZ*2/1024*21.5+17
```

Be sure to (PEEK(43)+PEEK(44)*256)−1,0 and issue NEW if
you change this pointer.

NEW places two bytes of zeros at the location this points to,
which indicates that no more BASIC lines follow.

GOTO and GOSUB will start looking for the subject line from
where this points, unless the subject line is greater than the current
line number MSB stored in 58 ($3A). See the search direction dis-
cussion at location 20-21 ($14-15) LINNUM. This is why it's a good
idea to put your most-used program lines at the top of the program,
with a GOSUB before them to perform any initialization. If the

most-used lines were after several lines of initialization, DATA, and REM statements, any backward reference to a line number would have to scan them every time.

SAVE always saves from where this points to, so you can save from any location you want by altering this.

LOAD, if used without ,1 after the device number, causes the load to start at the address that this location points to.

Other BASIC routines also use this pointer to scan the BASIC statements to complete their tasks.

Note: As expansion memory is added to the VIC-20, the power-on/reset routines change this pointer to reflect different start-of-BASIC locations. See Appendix E for details of the effect that adding expansion memory has on this and related pointers, and how to adjust for this in a program. Programs *can and should* be written to run in any standard memory expansion configuration.

Location 641–642 ($281–282) has a routine that you can use to set a VIC-20 back to its unexpanded RAM size, without removing any memory boards from the socket.

See related locations 45 ($2D) through 55 ($37), 641 ($281) and 643 ($283) for other useful techniques and programs. Appendix B includes a diagram of the relationship between these pointers and a BASIC program.

LOAD, without the ,1 after the device number, causes the load to start at the address that this location points to. You can use this to make a simple append program. It's easy to do but requires that the program to be appended has higher line numbers than the program to be appended *to*. Otherwise, you'll have problems changing or deleting the appended lines. The program to be appended *to* is loaded first. Then in direct mode enter:

AT=(PEEK(45)+PEEK(46)*256)−2

Since 45 points one byte past the zero link bytes, you can use it to find the end of the program. Now enter:

POKE 251,PEEK(43):POKE 252,PEEK(44)

in order to save the BASIC start address. Another line:

POKE 44,INT(AT/256):POKE 43,AT−(PEEK(44)*256)

will set the start address to the end of the program. (Another way to do this would be to enter POKE 43,AT AND 256:POKE 44, AT/ 256.) Then LOAD the program to be appended. To clean up, enter:

POKE 43,PEEK(251):POKE 44,PEEK(252)

which restores the pointer to the start of the first program. Now the two programs have been appended. A more sophisticated technique that allows intermixing lines based on line number is discussed at location 153 ($99).

It is possible to recover a BASIC program from an inadvertent
NEW. An automated technique is discussed in "UnNEW for the VIC
and 64," by Jim Wilcox, in the June 1983 issue of *COMPUTE!*. A
similar program which you can also use with a disk drive was pub-
lished in the November 1983 issue of *COMPUTE!'s Gazette* in the
article "VIC/64 Program Lifesaver," by Vern Buis.

45-46 $2D-2E   VARTAB

(handy location)

Pointer to the end of BASIC program, start of variables.

Scalar (nonarray) variables are built in this area, from where this
pointer is set, to increasing addresses. String variables are not stored
in this area, but their descriptors are. See location 51 ($33) for
information regarding string storage and 47 ($2F) for arrays.

See 187 ($BB) for a method of pointing a string descriptor to any
characters in memory, rather than BASIC's string pool. The format of
variables is discussed in Appendix B.

This pointer is also useful for finding the end of a BASIC pro-
gram since it points to the byte past the 0,0 end-of-program link.

After an array has been defined, it is stored above the area used
for scalar variables. When creating variables in this environment, all
arrays must be moved up seven bytes, the length of all scalar vari-
ables or descriptors, to accommodate the new variable. For this rea-
son, it's often suggested that variables be defined before arrays, and
that it's better to define variables that never get used than to move
up large arrays. If you're concerned about conserving memory, you
may want to simply predefine only as many *used* variables as pos-
sible. Remember that once a variable is defined, it takes up memory
space until you use a CLR command.

Defining variables in the order of their frequency of use is also
often suggested, since the variable locator and builder routine
GETPTR starts at the bottom of variable storage and works upward.
If the variable F is used the most, yet defined last, all other scalar
nonstring variables and functions have to be scanned every time F is
referenced. You may find it convenient to define variables in alpha-
betical order, instead. This makes it easy to tell if a variable name
has already been used, even if you look at the program much later.

See location 47–48 ($2F-30), the array pointer, for more
explanation of this.

Memory expansion indirectly affects this pointer, since the start
of the BASIC program moves. Appendix E has details.

CLR, NEW, RUN, LOAD, or the addition or modification of a
BASIC statement resets this pointer to one byte past the end of the
BASIC program, effectively making any variables inaccessible. See
the description of the workings of CLR at location 50782 ($C65E).
Any existing higher numbered statements have to be moved up in

the program storage area when a statement is added or lengthened. The BASIC MAKSPC routine opens a space for the new BASIC statement by moving up higher numbered lines with the MOVEBL routine at 50111 ($C3BF).

SAVE uses this pointer as the last byte+1 to save to the output device.

LOAD resets this pointer to 1 past the end of the loaded program, unless issued within the program. This allows the variables to be shared by the shorter LOADed program.

See "The Enhanced VIC-20: Adding a Reset Switch," by Joel Swank, in the February 1983 issue of *BYTE* for a description of an ML routine to restore this pointer after a RESET has been triggered.

By ending any larger program with a LOAD command for the following routine, you'll see a map of the order in which the nonarray variables were defined, as well as their addresses. You could use this map to find the variable names and their addresses within your own programs.

Program 1-4. Nonarray Variable Names and Addresses

```
0  P0=5:P%=5:P$="W":DEFFNP(C)=234
1  PRINT"VARNAME  ADDR"
2  PRINT"-------  ----"
5  QQ=0 : REM END MARKER
10 VS=PEEK(45)+PEEK(46)*256:VE=(PEEK(47)+PEEK(48)*
   256)-22
15 FORX=VSTOVESTEP7
20 X%=PEEK(X)AND128:Y%=PEEK(X+1)AND128:X1$=CHR$(PE
   EK(X)AND127)
25 Y1$=CHR$(PEEK(X+1)AND 127)
30 PRINTX1$;Y1$;
40 IFX%ANDY%THENPRINT"%{4 SPACES}";:GOTO80
50 IF(NOTX%)ANDY%THENPRINT"${4 SPACES}";:GOTO80
60 IFX%AND(NOTY%)THENPRINT"(FN) ";:GOTO80
70 IF(NOTX%)AND(NOTY%)THENPRINT"{5 SPACES}";
80 IFASC(Y1$)=0THENPRINT" ";
90 PRINTX:NEXT:PRINT"{RVS}{2 SPACES}USED="VE-VS+1:
   END
```

47-48 $2F-30   ARYTAB
(handy location)

Pointer to the end of BASIC variables, start of arrays.

Arrays are built from low to high memory, starting from where this pointer indicates. See the description of the move-up problem at location 45 ($2D).

Just as in locations 45–46, CLR, NEW, RUN, LOAD, or the addition or modification of a BASIC statement resets this pointer to

17

one byte past the end of the BASIC program, making the variables inaccessible. See the description of the workings of CLR at location 50782 ($C65E).

Again, Appendix B describes in detail the format of the array variables, and Appendix E shows the indirect effect on this pointer by memory expansion.

String arrays consist of a three-byte descriptor for each element in the array. The string itself is stored in the area pointed to by 51 ($33).

Remember that the zero element of an array exists, takes up space, and may be used.

To see a map of the order in which the variables were defined, and their addresses, LOAD the following routine at the end of a larger program. As with Program 1-4, you can use this to locate the array names and sizes within your own programs.

Program 1-5. Array Names, Addresses, and Sizes

```
0 DIMI6(5,4,3),I%(8,9),P$(2):E(8)=9
1 AE=0:AS=0:X=0:Y=0:Z=0:X1$="":Y1$="":AL=0:X%=0:Y%
  =0
2 PRINT" ---- -----"
3 PRINT" ADDR ARRAY"
4 PRINT" ---- -----"
10 AS=PEEK(47)+PEEK(48)*256:AE=(PEEK(49)+PEEK(50)*
   256)-1:X=AS
18 X=X+AL:IFX+1>PEEK(49)+PEEK(50)*256THENPRINT"
   {RVS}{2 SPACES}USED="AE-AS+1:END
19 PRINTX;
20 X%=PEEK(X)AND128:Y%=PEEK(X+1)AND128:X1$=CHR$(PE
   EK(X)AND127)
25 Y1$=CHR$(PEEK(X+1)AND 127)
30 PRINTX1$;Y1$;
40 IFX%ANDY%THENPRINT"%";:GOTO60
50 IF(NOTX%)ANDY%THENPRINT"$";
60 PRINT"(";:FORY=PEEK(X+4)*2TO1STEP-2:Z=(PEEK(X+4
   +Y)+PEEK(X+3+Y)*256)-1
62 PRINTMID$(STR$(Z),2,3)",";
70 NEXT:PRINT"{LEFT})"
80 AL=PEEK(X+2)+PEEK(X+3)*256
90 GOTO18
```

49-50   **$31-32**   **STREND**

(handy location)

Pointer to the end of BASIC arrays, start of free area.

Array or scalar nonstring variable definitions, as well as additional BASIC program lines, move this pointer upwards.

When storage for a string is allocated, starting at 51 ($33) and pointing down in memory, garbage collection is performed if the string begins before the location of the pointer.

FRE performs garbage collection and returns the value difference between this pointer and 51 ($33).

The BASIC routine MAKSPC at 50104 ($C3B8) pushes memory contents upward to create room for additional or lengthened BASIC lines or scalar variables.

The commands CLR, NEW, RUN, LOAD, or the addition or modification of a BASIC statement resets this pointer to one byte past the end of the program.

To calculate the free memory remaining without causing a garbage collection, you could use the following statement. If the amount of memory available is insufficient, you can use the FRE command to discard unneeded strings.

```
FREE=(PEEK(51)+PEEK(52)*256)-(PEEK(49)+PEEK(50)*25
   6)
```

51-52 $33-34   FRETOP
(handy location)

Pointer to the bottom of BASIC active strings.

This pointer marks the bottom of the active strings and the top of available free space. Strings are built from where 51 ($33) points, downward in memory addresses.

When finding space for a new string, the string routines begin looking at the location this pointer indicates and look downward. Then this pointer changes to indicate the beginning of the added string.

FRE and garbage collection routines readjust this pointer upwards. See the explanation of garbage collection at location 54566 ($D526).

Power-on/reset routines set this to the top of available RAM, but CLR copies the pointer at location 55 ($37).

53-54 $35-36   FRESPC
Pointer to the most current string added or moved.

Used as a temporary pointer by string building or moving routines.

55-56 $37-38   MEMSIZ   0/30 ($0/1E)
(handy location)

Pointer to the end of BASIC memory.

Power-on/reset routines set this pointer to the top of available RAM.

This pointer is decreased by 512 bytes when an RS-232 channel is opened to create two 256-byte buffers for input/output. A CLR is then issued. The pointer at location 643 ($283) which points to the top of user RAM is also lowered by 512 bytes (*Note: This may destroy any high RAM resident ML code*), making it necessary to open the RS-232 port (device 2) before defining any variables.

You can lower this pointer to reserve space for ML or any other purpose, such as relocating screen or character memory. After changing this pointer, enter a CLR which copies this pointer into location 51 ($33), preventing BASIC from using RAM above the pointer's indication.

Here's a program that reserves space at the bottom and/or top of BASIC. It first displays the range of BASIC and the amount of space included. Prompts for the amount of space to be reserved should be answered with 0 or the desired amount. The new range for BASIC and the size will then be displayed. A NEW is issued for you, and 0 placed in the byte before the start of BASIC.

Program 1-6. Reserving Space

```
100 BWAS = PEEK (44) *256 + PEEK(43)
110 EWAS = PEEK (56) *256 + PEEK(55)
120 PRINT "{CLR}BASIC ="BWAS"TO"EWAS"="EWAS-BWAS
130 PRINT "RESERVE AT BOT" : INPUT BRSRV
140 PRINT "RESERVE AT TOP" : INPUT ERSRV
150 BAFTER = BWAS + BRSRV
160 EAFTER = EWAS - ERSRV
170 BMSB = INT( BAFTER / 256 ) : BLSB = BAFTER - B
    MSB * 256
180 EMSB = INT( EAFTER / 256 ) : ELSB = EAFTER - E
    MSB * 256
190 PRINT "NEW ="BAFTER"TO"EAFTER"="EAFTER-BAFTER
200 POKE 198,4 :POKE 631,78 : POKE632,69 :POKE633,
    87 : POKE 634,13
210 POKE 44,BMSB :POKE 43,BLSB :POKE 56,EMSB :POKE
    55,ELSB :POKEBAFTER-1,0 :END
```

If an RS-232 device is to be opened in the BASIC program or in ML placed high in RAM, see the note at location 643 ($283).

Location 641-642 ($281-282) has a routine that you can use to set a VIC-20 back to its unexpanded RAM memory, without removing any memory boards from the socket.

See Appendix E for details of the effect that adding expansion memory has on this pointer, related pointers, and how to use these techniques in your own programs. It's a good idea to write your programs so that they will run in any standard memory expansion configuration; refer to Appendix E for an explanation of this.

57-58   **$39-3A**   **CURLIN**

(handy location)

Line number of the BASIC statement being executed, in LSB/ MSB format.

A 255 ($FF) in location 58 ($39), which would set the line number above the 63999 limit, indicates a direct mode statement or that editing of program lines is taking place. The routine MAIN sets location 58 ($3A) to 255 ($FF) when input from the keyboard is received.

The main BASIC execution routine NEWSTT updates this location as a new BASIC line is taken for execution.

Illegal BASIC keywords in direct mode check location 58 ($3A) to see if the direct mode is active.

BREAK and any error messages show the contents of this location as the number of the line being executed, unless it's an OUT OF DATA message, in which case the contents of location 63-64 ($3F-40), the current DATA line, are displayed.

The values in these locations are copied to address 59 ($3B) by the STOP and END commands, as well as when the RUN/STOP key is pressed. The command CONT moves 59 ($3B), the previous line number, back to here if 62 ($3E) is not zero. This indicates it is *not* a syntax error. GOTO uses this to determine how to search for its target line.

FOR and GOSUB stack this for NEXT and RETURN.

You can trace program execution, displaying this line number, by diverting the 776 ($308) vector.

See 61 ($3D) for the address of the current line.

The routine below shows how a trace can be used to display the area of the program being executed, with a switch to quickly disable the function.

Program 1-7. Trace

```
10 REM 57
20 REM A TRACE ROUTINE CAN DISPLAY THIS LOCATION.
30 REM BY TESTING A VARIABLE IN THE ROUTINE (LINE
   {SPACE}90) THE TRACE CAN BE DISABLED.
35 REM THIS IS SUPERIOR TO PRINTING A CONSTANT TIT
   LE AT THE START OF A ROUTINE.
40 TRACE=1:Z=57:DEFFNL(T)=PEEK(T)+PEEK(T+1)*256
50 T=FNL(Z):T$="INITIALIZING":GOSUB90
55 REM STATEMENTS IN THE ROUTINE
60 T=FNL(Z):T$="MENU SETUP":GOSUB90
65 REM STATEMENTS IN THE ROUTINE
70 T=FNL(Z):T$="ROUTER":GOSUB90
75 REM STATEMENTS IN THE ROUTINE
80 T=FNL(Z):T$="READDATA":GOSUB90
85 REM STATEMENTS IN THE ROUTINE
```

21

```
89 END
90 IF TRACE THEN PRINT"{RVS}@LINE"T;T$
95 RETURN
```

59-60 $3B-3C   OLDLIN

(possible user storage)

Previous BASIC line number executed, in LSB/MSB form.

You may use this location, but keep in mind that END, STOP, and the RUN/STOP key all copy the value in location 57 ($39) to these addresses to save the current line number for a possible CONT. CONT returns the value to location 57 ($39).

61-62 $3D-3E   OLDTXT

(handy location)

Saves TXTPTR pointer of statement being executed, for CONT.

Not to be confused with the BASIC line number, this is the *address* of the end byte of the BASIC statement that was just executed. NEWSTT saves TXTPTR here when executing a new BASIC statement.

TXTPTR is the pointer to the character being scanned. See locations 115–138 ($73–8A), which are the CHRGET routine.

END saves TXTPTR here and CONT restores from here. However, CONT won't continue if 62 ($3E) is set to zero by a clear routine, the LOAD command, program modifications, or error routines.

An ML tracer could print each BASIC statement before it executes by intercepting the indirect vector at 776 ($308)—(a colon in .A means new statement; otherwise, new line)—detokenizing with QPLOP, and continuing with the original value of 776 ($308).

See location 57 ($39) for the actual BASIC line number.

63-64 $3F-40   DATLIN

Current DATA line number in LSB/MSB form.

This location is *not* used by the read routines to select the line to read data from. It is informational only.

READ routines keep this pointer current. If a problem with a DATA statement occurs, this line number is included in the error message by moving it to location 57 ($39). Remember that this location is the standard line number for the error routines.

You can use the following technique to display the line number of the DATA statement currently being processed by a READ command. This can be of help to you when you're debugging a program, for you can insure that the proper DATA values are loaded and used.

Program 1-8. DATA-READ Review

```
10 REM 63
20 REM DISPLAYS THE LINE NUMBER OF DATA BEING READ
30 READ X$:PRINT"{RVS}DATA AT"PEEK(63)+PEEK(64)*25
   6
40 IFX$<>"Z"THEN30
45 END
50 DATA{3 SPACES}A,B,C,D
60 DATA{3 SPACES}E,F,G,H
70 DATA{3 SPACES}I,J,K,L
80 DATA{3 SPACES}M,N,O,P
90 DATA{3 SPACES}Q,R,S,T
100 DATA{3 SPACES}U,V,W,X
110 DATA{3 SPACES}Y,Z
```

65-66   **$41-42**   **DATPTR**

Pointer to the current BASIC data item.

The READ routine uses this location and location 67 ($43) to track where it is currently reading data. Also see 75 ($4B).

RESTORE or CLR resets this pointer to the location the pointer at 43 ($2B) indicates, the beginning of the BASIC program. See the description of the workings of CLR at location 50782 ($C65E).

67-68   **$43-44**   **INPPTR**

Pointer to source of INPUT, GET, and READ information.

A common pointer for all sources of incoming information, whether from the BASIC input text buffer at 512 ($200) or DATA statements. Also see location 75 ($4B).

69-70   **$45-46**   **VARNAM**
                                      (handy location)

Current BASIC variable name with type flags.

Two characters are stored here, the second a 0 if the variable has only a one-character name. Dollar and percent signs are not saved as part of the name.

The high order bit of each character indicates the type of variable. They are:

Floating point: no high bits on
Integer: both characters have high bit on (128 is added)
String: second character has high bit on (128 is added)
Function: first character has high bit on (128 is added)

Also see locations 13 ($D) and 14 ($E).

Appendix B has a description of the variable formats in storage.

71-72 $47-48   VARPNT

(handy location)

Pointer to the descriptor of the current BASIC variable.

This points to the byte just after the two-character name in the variable descriptor.

The FN routines alter this pointer so that the dependent variable is not changed while the function is performed. For example, when DEF FNA(B)=B/60:X=FNA(Y) is entered, the dependent variable *B* must be protected from change. This pointer is changed by the FN processing to reference an area in the FN descriptor rather than the variable *B*. The area is initialized to the value of variable *Y*, divided by 60, because of the given FN equation, and then assigned to the variable *X*. *B* was never changed.

See Appendix B for the format of variable descriptors.

See 95 ($5F), the pointer to the start of the variable descriptor used after locating or creating the variable.

The following routine shows a method of determining the address of any BASIC variable descriptor. Once the address of AV$ has been found, the routine modifies the descriptor to point to an error message string within BASIC ROM. You can use this same technique in modifying descriptors for your own purposes. BV is also modified in this routine to contain the number of the program line it searched for.

Program 1-9. Variable Descriptor

```
100 REM 71
110 REM ROUTINE TO FIND THE ADDRESS OF THE CURRENT
    VARIABLE DESCRIPTOR
120 REM REFERNCE THE VARIABLE BY SETTING{2 SPACES}
    TO ITSELF, SUBROUTINE RETURNS THE VARIABLE
130 REM ADDRESS IN VA.
140 AV$="TEST STRING"
150 AV$=AV$:GOSUB220:PRINT"AV$@"VA"{RVS}NOW=";:POK
    EVA+2,14:POKEVA+3,158:POKEVA+4,193
160 PRINT"'"AV$"' ERR MSG."
170 BV%=BV%:GOSUB220:PRINT"BV%@"VA:POKEVA+3,PEEK(5
    7):POKEVA+2,PEEK(58)
180 PRINT"{RVS}NOW= MY LINE NUMBER OF"BV%
190 CV=CV:GOSUB220:PRINT"CV@"VA
200 CV(4)=CV(4):GOSUB220:PRINT"DIM CV BEGINS @"VA
210 END
220 POKE251,PEEK(71):POKE252,PEEK(72):VA=PEEK(251)
    +PEEK(252)*256-2:RETURN
```

73-74 $49-4A   FORPNT
Pointer to BASIC variable used in FOR loop.

These locations are also used by many routines for other purposes.

FOR saves the address of the dependent variable here, then pushes the following onto the stack in this order:

- Address of the return line statement
- The return line number, TO value
- STEP sign, STEP value, this pointer, and a constant value 129 ($81).

See the description of the stack area at location 256 ($100).

FOR also uses locations 73–74 to save the result of each STEP increment/decrement stored in the dependent variable area. For example, FOR C=1 TO 10 STEP 2.

This location is later used by the BASIC routine SCNSTK to find the proper FOR loop values when NEXT is encountered, since NEXT can include variable names. If 74 ($4A) is set to 0 ($0), the first FOR information found on the stack will be used. This occurs when NEXT doesn't specify a variable name. For example, NEXT C has to locate the proper stack items for the FOR C= loop, ignoring any others.

Other routine usage of this location:

- READ/INPUT/GET use this location to point to the variable to assign the value to. It is also used as a temporary area for the evaluated value of the subject variable during assignment (for example, S=16*(17/3)), or the string pointer for something like S$=A$+"ABC".
- LIST uses 73 ($49) as a temporary save area.
- WAIT uses 73 ($49) to save its second parameter and 74 ($4A) for the third, or zero, default.
- CLOSE uses 73 ($49) to save the file number.
- LOAD and SAVE use this location to save the device number.
- RETURN uses 74 ($4A) as a flag, set to 255 ($FF), to pull the associated GOSUB entry off the stack.

Other routines can use this area without affecting FOR because all needed information is now stored on the stack.

75-76 $4B-4C   OPPTR
Math operator displacement/INPUT TXTPTR.

These locations serve as the displacement of the current math operator in a table during formula evaluation. The math operator table is at location 49280 ($C080).

This location is yet another save area for TXTPTR by the READ, INPUT, and GET commands; these are the original contents before they are altered by READ, INPUT, or GET.

77 $4D   OPMASK
Comparison desired mask.

This location's value is created by the expression evaluator routine FRMVL. A value of 1 indicates a greater-than check, 2 signifies an equals check, and 4 flags a less-than check. They may be used in combination by adding the values.
See also location 18 ($12).

78-79 $4E-4F   DEFPNT
Pointer to current FN descriptor in variable storage.

DEF FN uses this location as a pointer to the descriptor created. During FN, this is a pointer to the FN descriptor used to save the evaluation results. This is also a work pointer for garbage collection.

80-81 $50-51   DSCPTN
Pointer to the current string descriptor.

This location is used and set by the string assignment and handling routines. Location 82 ($52) is related to this location.

82 $52   SIZE
Length of the current BASIC string.

See location 80 ($50).

83 $53   FOUR6
Constant, set at either 3 or 7, for garbage collection.

Used to instruct garbage collection routines whether a three- or seven-byte string descriptor is being collected.

84-86 $54-56   JMPER
Jump opcode and vector to function routine.

The 6502 ML jump operation code is 76 ($4C), followed by the address of the required function from the function vector table at 49234 ($C052). This is determined by the expression evaluation routines.
85 ($55) is also used as a one-byte work area for garbage collection and string substringing, such as with the LEFT$, RIGHT$, and MID$ commands. Location 86 ($56) is used as a one-byte work area for addition and exponentiation rounding.

87-96 $57-60   TEMPF3

BASIC numeric work area.

Another busy work area for BASIC. Because so many BASIC routines use and overlay this area, none of it can be assumed to contain any specific data at any one time.

● 95 ($5F) is a pointer to the variable's seven-byte descriptor, after location or creation of the variable.

● 95-96 ($5F-60) is used by LIST as a pointer through the BASIC program as it lists it.

● 95 ($5F) is used as a flag to indicate that a decimal point has already been found when converting a string to a floating point number. For example, VAL(+123.456).

The garbage collection routines at 54566 ($D526) GRBCOL use this area for temporary string pointers and length counters.

Also see location 71 ($47).

97-102 $61-66   FAC
                                                    (handy location)

BASIC floating point accumulator one.

Appendix B has a full description of each of the floating point accumulators, as well as an explanation of floating point numbers and conversion from/to integer format.

● 97 $61 FACEXP
Exponent of the value + 128.
● 98-101 $62-65 FACHO
Normalized mantissa of the value.
● 102 $66 FACSGN
Sign: 0=positive, 128-255 ($80-FF)=negative
These locations are used by BASIC routines to perform any mathematical processes called for by the user or by BASIC itself. Integer numbers are converted to floating point before any computations are made, then converted back to integer if needed. Strings may be converted to floating point and vice versa.

Conversion to/from floating point is a rather involved subject to master. Fortunately, the VIC-20 includes many efficient routines for this process. See FAC and FAC2 in the label cross-reference index in the appendices.

Other uses of this area include:

● For two-byte integer to floating conversion, locations 98 ($62) and 99 ($63) are filled with the integer number, the exponent is set to 136 ($88) or 144 ($90), part of the SGN routine initializes the rest of FAC, and the ADD routine normalizes and converts. String processing routines use this location to process the string descriptor, with 100-101 ($64-65) being the descriptor address.

● The formula/expression evaluation routine at location 52638 ($CD9E) stores its results in this location in the form of a floating point number or a pointer to a string.

  ● LIST uses this location to convert integer line numbers to floating point.

  ● A variety of other routines use this location as a work area.

The floating point accumulators and the BASIC routines that use them for mathematical operations can be used in your own ML routines to perform the BASIC operations. You can do this by storing the operands in the appropriate accumulators, either directly or as the result of calling integer to floating point conversion routines. You would then branch to the desired BASIC ROM routine. The format of the floating point numbers is discussed in more detail in Appendix B.

The following assembler instructions can be entered to use the BASIC routines in your own ML routines:

● To convert an integer to floating point in FAC, you would load the Least Significant Byte (LSB) in the Y register, load the Most Significant Byte (MSB) in the Accumulator, and jump to location $D391.

● To load FAC from memory location 828 (the start of the tape buffer), you need to load the Y register with the number value of location $3, load the Accumulator with the number value found in location $3C, and jump to location $DBA2.

● Loading FAC2 from location 828 is identical to the process used to load FAC, except that the jump should be to location $DA8C.

● To move FAC2 to FAC, you only need to jump to location $DBF6.

● To move FAC to FAC2, jump to location $DC0C.

● Storing FAC at any memory location can be done by loading the X register with the MSB, the Y register with the LSB, and then jumping to location $DBD7.

● To store FAC2 at any location is a bit more involved. You need to LOAD the Accumulator with the MSB of the address, store the Accumulator at location $49, LOAD the Accumulator with LSB of the address, store the Accumulator at location $74, and finally jump to location $DBC7.

● To use addition, you simply jump to location $D86A. The first value is placed in FAC, the second in FAC2, and the final result is stored in FAC.

● Jumping to location $D853 will give you subtraction. The second value, stored in FAC2, is subtracted from the first value, found in FAC. The resulting value is stored in FAC.

● To use multiplication, jump to location $DA28. The values in FAC and FAC2 are multiplied and the final result is stored in FAC.

• To use division, jump to $DB12. The value stored in FAC2 is divided by the value found in FAC. The result is stored in FAC.

• Exponentiation can be used by jumping to location $DF7B. The value in FAC2 is raised by the value in FAC, and the result is found in FAC.

• Various trigonometry functions can be used as well. However, these operations use FAC only. SIN can be used by jumping to location $E268; TAN can be used by jumping to location $E2B1; and COS can be used by jumping to location $E261.

• Converting FAC to an ASCII string at location $100 (until a $00 value is found) is done by jumping to location $DDDD.

• Converting FAC to an integer at locations $64–65 is accomplished by jumping to location $D1AA.

103 $67   SGNFLG
BASIC series evaluation number of items.

This location is used by the mathematical formula evaluation routine to indicate the number of evaluations to be done. A complex formula may need several levels of evaluation of terms before the final result can be determined. This location contains the number of terms to be resolved.

Occasionally, this location serves as temporary storage for the sign of FAC.

104 $68   BITS
High order FAC propagation word. Overflow.

Overflow work area byte resulting from normalization of FAC when a floating point number is being constructed.

105–110 $69–6E   FAC2
(handy location)

BASIC floating point accumulator two.

Appendix B has a full description of each of the floating point accumulators, as well as an explanation of floating point numbers and conversion.

• 105 $69 ARGEXP
Exponent of the value + 128
• 106–109 $6A–6D ARGHO
Normalized mantissa of the value
• 110 $6E ARGSGN
Sign: 0=positive, 128–255 ($80–FF)=negative

These locations are used by BASIC routines to perform any mathematical processes that involve more than one value, such as add, subtract, multiply, divide, and so on. Typical is the divide routine which divides FAC by FAC2 and leaves the result in FAC. See

29

the description at the end of the explanation for locations 97–102 ($61–66).

These are also used in the normalization process of FAC, comparing numerics, and formula evaluation.

III $6F   ARISGN
FAC to FAC2 sign comparison.

This is used to indicate the difference or likeness of signs. 0 in this address means FAC and FAC2 have same sign, while a value of 225 ($FF) means their signs are different.

Along with locations 112–114 ($70–72), this address is used as a work area for string handling routines. Locations 111–112 ($6F–$70) are also used as a pointer to a string.

112 $70   FACOV
Low order of FAC mantissa for rounding.

With location 111 ($6F), this location is used by string handlers as a pointer to string.

113–114 $71–72   FBUFPT
Series evaluation pointer.

This is a pointer to the table of constants for the trigonometric function being evaluated by the formula evaluation routines. For this, the location will point somewhere within the tables starting at 58171 ($E33B), 55745 ($D9C1), 57284 ($DFC4), or 58092 ($E2EC). See the tables at these locations for further information.

The pointer is also used for saved TXTPTR for READ, GET, INPUT, and VAL; index to the end of the BASIC line in the BASIC text buffer at 512 ($200) during tokenization by CRNCH; string setup pointer; TI$ assignment work area; and the work area for the building of an array descriptor.

It's also described in various notes as the tape buffer pointer, but I've found no reference to it for that specific purpose. See 178 ($B2) for the tape buffer pointer.

115–138 $73–8A   CHRGET
(handy location)

Get-BASIC-character routine.

This routine is used to scan BASIC lines or any other area that a calling routine desires by setting TXTPTR. Either the next character or the current character can be retrieved, depending on the entry point used. This routine, when entered from CHRGET, increments TXTPTR to point to the next possible location of a character. Notice that this is accomplished by modifying the operand of its own LDA (LoaD the Accumulator) instruction. Upon entering at CHRGOT or

falling through from the previous instruction, the routine skips spaces, sets flags to indicate the type of character TXTPTR is pointing at, and returns to the calling routine with the retrieved character in .A, the accumulator.

This sequence of instructions is copied from CGIMAG at location 58247 ($E387) to page 0 at power-on/reset, or at BASIC cold start, so that the routine runs faster, can modify its TXTPTR, and to allow the wedging-in of user routines. The copying of this ROM to page 0 is done by the routine INITBA at location 58276 ($E3A4).

When a new BASIC line is retrieved from BASIC program storage for execution, NEWSTT processes the line number and link addresses before using CHRGET to find tokens and ASCII characters.

The .A register holds the character at exit.
Processor status flags at exit are:

Carry Clear if digit
Carry Set if not digit
Zero Set if $00 (end-of-line) or $3A (colon)
Zero Clear if any other character
Negative Clear if the value is between $00 and $B9
Negative Set if the value is between $BA and $FF
Overflow Set if colon and Overflow was previously set
Overflow Clear otherwise

Since this is such an important routine, a disassembly listing is provided below to give you a better understanding of how it works.

115	$73	CHRGET	INC	$7A	;increment TXTPTR LSB
117	$75		BNE	$79	;if LSB not 0, then skip ;next
119	$77		INC	$7B	;increment TXTPTR MSB ;byte
121	$79	CHRGOT	LDA	TXTPTR	;load byte from where ;TXTPTR points ;if $7B = 02 then direct ;mode statement
124	$7C	CHRTST	CMP	#$3A	;if > 57 $39 (char 9) then ;carry set
126	$7E		BCS	$8A	;and exit
128	$80		CMP	#$20	;if space, ignore it and
130	$82		BEQ	$73	;get the next character
132	$84		SEC		
133	$85		SBC	#$30	;if 48–57 $30–39 then digit, ;carry clear
135	$87		SEC		
136	$88		SBC	#$D0	;if < 48 $30 (char 0) then ;carry set
138	$8A		RTS		;carry clear if 0-9, else ;carry set

31

A wedge can be inserted in CHRGET/CHRGOT to intercept the interpretation of BASIC keywords or ASCII characters. Since the locations 122–123 ($7A–7B) are referenced by ROM BASIC routines, you should not change TXTPTR to another location. One wedge insertion technique is to place a JMP opcode in 115 ($73) and the address of your wedge in 116–117 ($74–75). Your wedge should immediately JSR to 118 ($76), which you'd have changed to a JSR to location 58247 ($E387), the ROM master copy of this routine. The ROM copy will increment $7A and $7B for you and test the charac-ter in location $EA60. That address is frozen in the ROM version of the $79 LDA instruction. The flags can be ignored. The RTS in the ROM copy will return to the RAM CHRGET, which then tests the chosen character TXTPTR is pointing to. Your wedge is reentered via the RTS in the RAM CHRGET routine. Your wedge can process the character before BASIC sees it. You can go back to BASIC with JMP $79 to reset the status flags using CHRGOT, or get the next charac-ter with JSR $76.

For example:

```
CHRGET $73 JMP  $4000    ;branch to my wedge
       $76 JSR  $E387    ;perform the ROM copy of routine
```

The rest of the routine is left as is. *The first instruction in my wedge should be JSR $76.*

Another technique for wedging into CHRGET is to place the JMP to your routine at $73–75, then in your wedge routine increment TXTPTR. A JSR $79 lets the CHRGET routine do its nor-mal testing. When CHRGET issues RTS, your routine is reentered, and can decide to process and then jump (JMP) to $73 or let BASIC have it by using a JMP $79.

A wedge can be placed in the CHRGOT routine by replacing the CMP at location $7C with a JMP to your wedge. This can then jump to the ROM copy of the $7C CMP at location 58256 ($E390) after your wedge determines if it wants the character. The ROM routine RTS will return to the original routine. If your wedge processes the character, you would JMP to $73 or $79.

CLR, NEW, and power-on/reset set TXTPTR back to the start of the BASIC program.

Here's an example of a wedge program for you to examine, type in, and use. It features single keystroke entry of BASIC keywords and automatic detection of the quote mode so that strings within quotes are left alone. It immediately displays the entire word on the screen. The program uses the Commodore key to indicate that the key pressed along with it is a shorthand entry.

You can change the second 2 in line 130 to 4 if you want to use the CTRL key instead of the Commodore key for this shorthand entry.

The program works with any memory configuration. The loader relocates the ML routine at the top of memory and protects it from BASIC. When the program requests the memory address, simply press the RETURN key. To toggle it off/on, you can SYS PEEK(253)*256. The function keys have been defined as direct mode keywords, the space bar as LIST, and other keys can be pressed to see which keywords each represents.

Program 1-10. Wedge

```
3 REM USE C= KEY AND OTHER KEYS FOR BASIC KEYWORDS
5 REM ** WARNING, DO NOT RENUMBER LINES **
9 PRINT"{CLR}"
10 PRINT"{RVS}BUILD MLC WHERE? Ø=TOP{OFF}":OPEN1,Ø
   :INPUT#1,S$:S=VAL(S$):CLOSE1
20 L=256:LN=100
30 E=PEEK(56)*256+PEEK(55)
33 IFS=ØTHENS=E-L
35 P=INT(S/256):S=P*256
37 IFS<4096ANDE>7680THEN50
40 IFS<ETHENPOKE56,P:POKE55,Ø
50 PRINT"BUILDING...":FORI=STOS+9999:READA:IFA=888
   THEN900
55 IFA=999THEN90
60 CK=CK+A:IFA=300THENA=P
70 IFA<ØTHENA=ABS(A)
80 POKEI,A
85 NEXT
90 RESTORE:READA:IFPEEK(S)<>ATHENPRINT"{RVS}POKED
   {SPACE}TO ROM, RELOAD.{OFF}":FORX=1TO300:NEXTX:
   RUN1Ø
95 PRINT"SYS"S"TO TOGGLE.":POKE253,P:NEW
100 DATA120,173,20,3,72,173,21,3,72,173,-39,300,20
    8,2,169,-41,888,1429
110 DATA141,20,3,173,-40,300,208,2,169,300,141,21,
    3,104,141,-40,888,1646
120 DATA300,104,141,-39,300,88,96,0,0,72,138,72,15
    2,72,234,234,888,1964
130 DATA234,234,165,212,208,17,173,141,2,201,2,208
    ,10,165,203,201,888,2376
140 DATA64,176,4,76,-156,300,234,165,203,72,76,-14
    1,300,0,234,76,888,1683
150 DATA-147,300,138,24,105,158,133,34,144,8,160,1
    93,132,35,144,6,888,1567
160 DATA176,4,160,192,132,35,160,0,162,0,132,198,1
    77,34,48,12,888,1622
170 DATA200,230,198,166,198,157,119,2,176,242,144,
    240,230,198,166,198,888,2864
180 DATA41,127,157,119,2,169,20,141,119,2,230,198,
    234,104,141,-155,888,1649
```

33

```
190 DATA300,234,234,104,168,104,170,104,76,191,234
    ,15,234,234,234,234,888,2870
200 DATA234,170,41,128,205,-155,300,240,166,168,18
    9,-185,300,201,255,240,888,2497
210 DATA150,170,152,72,76,-82,300,0,0,175,181,199,
    205,211,89,187,888,2085
220 DATA255,196,10,243,44,20,92,62,255,255,231,25,
    133,35,238,69,888,2163
230 DATA255,255,255,28,228,6,42,102,255,111,121,23
    4,158,249,151,255,888,2705
240 DATA75,255,224,3,65,51,143,107,83,217,0,139,18
    4,124,56,178,888,1904
250 DATA79,255,193,202,208,214,145,255,39,0,888,15
    90
260 DATA 999
900 READ A:IF A<>CK THEN PRINT"*** DATA ENTRY ERRO
    R ON LINE"LN"***" :END
910 CK=0 :LN=LN+10 :I=I-1 :GOTO85
```

139-143 $8B-8F   RNDX
BASIC RND work area, last random number, or initial seed.

This routine is initialized at power-on/reset, along with the CHRGET routine, from a master copy in ROM. The copying of this area to zero page is done by the routine INITBA at location 58276 ($E3A4). The initial value of this location is .811635157, or $80,4F,C7,52,58 in five-byte floating point format. RND (Random) returns a number ranging between 0 and 1.

The sign of the argument affects the resulting random number generated by RND. The number or variable in parentheses after RND is called the argument. For example, in:

RND(9)

9 is the argument.

The RND routine creates the random number by exchanging the first and fourth byte of the mantissa of the argument. The exponent is then overlaid to insure a value ranging between zero and one.

A positive argument is ignored by RND and the resulting random number is based on the last seed stored in this location. Because of this, using only positive arguments for RND will cause the *same sequence* of numbers to be returned from the original ROM-copied seed. That's not to say that running the program twice will give the same random numbers. The ROM seed is only copied here at power-on/reset. It *does* mean that the *n*th RND used since power-on/reset will return the same number. In other words, the third random number will always be the same, as long as the computer has not been turned off, then back on. This can be helpful when testing a program.

Location 57482 ($E08A) shows the constants used to derive this version of RND. Once the constants have been applied to the previous seed value, RND treats the resulting value the same as it would a negative argument.

A negative argument will return the same number for a given argument. This happens because a negative argument replaces the seed number in this work area. Normally, the result of the previous random number replaces the seed, and is used in determining the next random number. The negative argument sets the seed at that value and so determines the future random numbers from that point on. A particular seed value will always cause the next random number to be a particular number.

RND(0) causes the values in the 6522 VIA#1 chip timer 1 at 37140–37141 ($9114–9115) and in timer 2 at 37144–37145 ($9118–9119) to be used to derive the returned random number. The RND(0) feature is not the best recommendation for repetitive random numbers since clock cycles may remain constant during program loops.

R=RND(−TI) is recommended as an initial seed (ignoring the returned result in variable R), with the remainder of the program using R=RND(1) whenever a random number is needed. The desired range of integers returned (range=A to Z) may be insured by using R=INT((Z−A+1)*RND(1)+A).

Location Range: 144–255 ($90–$FF)
Kernal Working Storage

Page 0 storage for the Kernal
The INITMEM routine initializes this area to zeros at power-on or reset.

INITSK, a Kernal power-on/reset routine, then initializes any locations with needed values. See the comments in each location for these initialization values.

BASIC lets the Kernal manage the second half of page 0, but it can and does examine parts of it, as well as alter other parts. You can change any location with POKE or with ML instructions.

144 $90   STATUS
(handy location)

ST status of I/O completion.

Kernal routines that open channels or perform input/output functions check and set this location. When BASIC examines this status or sets the ST variable for the programmer to examine, a jump is made to the Kernal vector CRDST. This, in turn, goes to the read-status routine at 65111 ($FE57). The BASIC syntax checker does not allow ST=expression.

Machine language routines need to load and examine this byte, rather than using the BASIC variable ST.

See Appendix D for a device, secondary address, and status codes table.

145   **$91**   **STKEY**   **255 ($FF)**

(handy location)

Keyswitch PIA: bottom keyboard row scan.

Each time the jiffy clock TIME is updated by the Kernal, the contents of VIA2PA2 (VIA 2-Port A) are copied to this location.

Every other key on the bottom row of the keyboard may be tested for in this location, without using a GET command in BASIC.

Here's an explanation of the different values you'll find in this location:

255 ($FF) = no key pressed

254 ($FE) = STOP key pressed; STOP routine will find and act on

253 ($FD) = left SHIFT key pressed; this may be the most useful returned value. It allows the program to distinguish between left/ right SHIFT by checking location 653, then location 145, bit 1 for left/notleft.

251 ($FB) = X key pressed

247 ($F7) = V key pressed

239 ($EF) = N key pressed

223 ($DF) = comma key pressed

191 ($BF) = slash key pressed

127 ($7F) = cursor-down/up key pressed

You can examine the values in this location when the bottom row keys are pressed by entering and running this short program.

Program 1–11. Key Values, Bottom Row

```
10 REM 145
20 X=PEEK(145):IFX<>255THENPRINTX;
30 POKE198,0:GOTO20
```

Also see location 197 ($C5), matrix coordinate of key, and location 37153 ($9121), VIA2PA1.

See location 631 ($277) for a summation of keyboard-related RAM locations.

146   **$92**   **SVXT**

(possible user storage)

Tape: 0/1 bit timebase fluctuation during read operations.

This location stores the difference between the actual time for

the dipole just read and the adjustable timebase. This determines whether a dipole is considered to be 0 or 1.

The positive and negative voltages recorded onto a tape result in two different poles, just like a magnet with a north and south pole. Recording on tape can then be represented as a square wave, as shown in Figure 1-2. A *dipole* is one square wave cycle. A dipole time is the time taken to go through two poles, or one square wave cycle, such as the one labeled A and B in Figure 1-2.

Figure 1-2. Square Wave
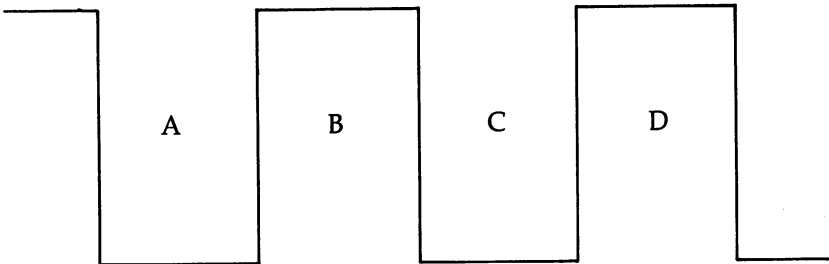
After a bit is read, the value in this location is used to adjust the value in location 176 ($B0), which slides the timebase value for the boundary between a zero and one bit dipole. By comparing the actual time it takes to read a bit to the time the tape routines expect it would take, the tape routines are able to make adjustments. They adjust the value limitations to define noise, 0, 1, or a word marker for the next bit. Through this method, tape units can read tapes recorded at slightly different speeds than the speed at which the unit itself is reading. This location is reset to 0 after each bit has been read.

147   **$93**   **VERCK**
                                                 (possible user
                                                 storage)

Tape: 0=LOAD, 1=VERIFY

The Kernal performs LOAD and VERIFY in the same routines, with this location used to determine which is being performed. The Kernal LOAD routine saves the 1 or 0 value here.

BASIC, on the other hand, uses location 10 ($A) for its LOAD or SAVE determination.

148   **$94**   **C3PO**
(possible user storage)

Serial: output deferred character flag.

This location is used by the Kernal serial output routines to determine when to send the buffered output serial character stored in location 149 ($95).

149   **$95**   **BSOUR**
(possible user storage)

Serial: output buffered character.

A value of 255 ($FF) in this location indicates that no character is waiting for serial output.

150   **$96**   **SYNO**
(possible user storage)

Tape: block found flag, tape leader length bit count.

This location's values indicate during tape LOAD that:

● 0: either no block is recognized yet or a block is recognized and data is being read from that block.

● 16–126: has read at least 16 leader bits during read of the tape leader either before the first block or between blocks 1 and 2, and is now waiting for the word marker at the end of the leader. Leader bits are composed of dipoles containing zeros, differing from data dipoles which have reverse dipoles of zero then one, or vice versa.

151   **$97**   **XSAV**
.X register SAVE area for get and put ASCII characters routines.

158 ($9E) is also used by output routines.

The INITEM routine uses this location to test for the start of RAM and to insure that locations 0–1024 are accessible.

152   **$98**   **LDTND**
(handy location)

Number of currently open files, not to exceed ten.

This is used as an end index for the last-used entry in the file,

device number, and secondary address data tables, found at the following locations:

601 ($259) LAT File number table
611 ($263) FAT Device number table
621 ($26D) SAT Secondary address table

Entry n in any table corresponds to entry n in the other two tables.

CLOSE decrements this number and shifts up the table entries to close any empty gap.

OPEN increments this number and adds the appropriate information to the bottom of the tables.

Routine CLALL, close-all-files, sets this number to zero, emptying the tables.

You can cause BASIC to *forget* all open files by POKEing a 0 into this location. This does not CLOSE any currently open files. To insure that no more than ten files are currently open, you could use the following lines to begin your program:

Program 1-12. Number of Open Files

```
10 REM 152
20 REM YOU CAN CAUSE BASIC TO 'FORGET' ALL OPEN FI
   LES BY POKING Ø INTO THIS LOCATION.
30 REM *NOTE* THAT THIS DOES *NOT* CLOSE ANY CURRE
   NTLY OPEN FILES.
40 REM THE FOLLOWING CAN BE USED TO INSURE THAT NO
    MORE THAN 1Ø FILES ARE OPEN:
50 IF PEEK(152)<1ØTHEN7Ø
60 PRINT"MAX FILES ALREADY OPEN, SPECIFY FILE NUMB
   ER TO BE CLOSED?":INPUT F% :CLOSE F%
```

See locations 183–187 ($B7–BB) for the current file parameters.

153 $99   DFLTN
(handy location)

Device number of the current input device.

Used by the Kernal to determine the routines called for processing the received data.

VIC-20 devices are:

0 keyboard
1 tape
2 RS-232/user port
3 screen
4–5 printer
8–11 disk
Device numbers 4–31 could be any serial device.

BASIC passes the file number to the Kernal when INPUT# or GET# causes an indirect jump via the vector at 798 ($31E) to 62151 ($F2C7). This is the open-input-channel routine, CHKIN. The vector at 798 ($31E) may be changed to your own front-end routine. CHKIN stores the current input device number here. See CHKIN on page 186 of the *VIC-20 Programmer's Reference Guide*. BASIC calls CLRCHN after every input to close the channel. See CLRCHN on page 191 of the reference guide. BASIC uses location 19 ($13) CHANNL to save the active input device number for prompting and screen control purposes.

SETIODEF sets the default to 0 during power-on/reset.

See Appendix D for a device, secondary address, and status codes table.

Jim Butterfield gave directions for reading the tape as though it were the keyboard, in the article "BASIC Program Merges: PET and VIC," in the June 1982 issue of *COMPUTE!*.

1. LOAD the program or routines that you wish to later merge into other routines or programs.

2. OPEN 1,1,1,"filename": CMD 1: LIST will create a tape in LIST format, with keywords detokenized and readable line numbers. The LIST command could be qualified by giving a line number range, if desired.

3. PRINT#1 : CLOSE 1 will properly finish the tape. The output may be directed to disk instead, but the following steps work only with tape.

4. Rewind the tape. LOAD the other program or routine that you wish to merge to.

5. POKE 19,1 : OPEN 1 to set the current input channel number and to bypass the tape header.

6. Clear the screen and press the cursor-down key three times. It's very important that this be exactly three cursor-downs, preceded by a screen clear.

7. PRINT "home" : POKE198,1 : POKE631,13 : POKE153,1. This puts a carriage return in the keyboard buffer, indicates that one character is in the buffer, and changes the current input device number to the tape.

8. Ignore any SYNTAX ERROR or OUT OF DATA messages.

9. CLOSE 1 to finish.

10. The program lines on the tape have now been merged.

Steps 1–3 may be used to obtain a detokenized LIST form of the program in machine-readable form on either tape or disk.

Step 7 doesn't seem to work for disk. You can save the program on tape and use the tape technique.

Jim Butterfield also explored the subject of merging programs from disk files. His article "Merging BASIC Programs from Commodore Disk " appeared in the October 1983 issue of *COMPUTE!*. A step-by-step explanation of the process is included in this article, along with a program which merges programs. The process involves writing a BASIC program that reads lines from two program files on disk and writing a third program with the lines from both files in the correct order. This is done by comparing the line numbers from each program, writing the line with the lowest number, and getting the next line from the file. This continues until both program files are emptied. A few details are worth mentioning about this procedure.

- Open the input program files with OPEN x,8,x,"filename,P,R"
- Open the output file with OPEN y,8,y,"0:filename,P,W"
- DIM A$(2),B$(2),C$(2),N(2) can be used to contain each line number's LSB in A$, its MSB in B$, its text line in C$, and the full line number in N. By varying the subscript used to reference the arrays, you can easily select the line from file one or file two. If 1 and 2 are used as file numbers in the OPEN statement, GET# can use the subscript value of the file number.
- The first two bytes of the input disk files (the saved-from address) can be discarded with GET#x,A$,A$ since the corresponding output file's address can be set to allow a PET as well as a VIC and Commodore 64 to load the program.
PRINT#y,CHR$(1);CHR$(4) does this.
- When reading the existing program files, the two-byte link filed on the start of every line should be tested for an end-of-program condition:

GET#x,A$,B$:IF A$="" AND B$="" THEN...

The routine branched to should insure that only the other file is read from, unless *that* file has also ended, meaning the output file can be marked with the end-of-program link field (PRINT#y,CHR$(0);CHR$(0);) and all files should be closed and the program ended.

- Since the decision of which line to output, and the file to get that line from, is based on the line number, N(x) is obtained by:

```
GET#x,A$(x),B$(x)
IF A$(x)="" THEN A$(x)=CHR$(0)
IF B$(x)="" THEN B$(x)=CHR$(0)
N(x)=ASC(A$(x))+ASC(B$(x))*256
```

- The actual text of the program line can be obtained with:

```
500 C$(x)=""
510 GET#x,A$:IF A$="" THEN 999
520 C$(x)=C$(x)+A$:GOTO 510
```

The routine at 999 is branched to when end-of-program line is encountered.
   • To write the output file, use:

PRINT#y,CHR$(1);CHR$(1);A$(x);B$(x);C$(x);CHR$(0)

where the leading ones are a dummy link field that is recalculated by LOAD, and the ending zero signifies end of line.
   • Be sure to examine the disk return codes when opening the input and output files:

INPUT#15,E,E$,E1,E2
IF E THEN PRINT E$:CLOSE 15:END

This assumes you OPENed #15 at the start of your program.
   See location 154 ($9A), output device number, for details of input/output diversion to other devices using SYS and locations 780–783 ($30C–30F). See location 19 ($13) for a description of BASIC device diversion.
   See location 186 ($BA) for the current device number.

154 $9A   DFLTO
(handy location)

Device number of output device.

   BASIC passes the file number to the Kernal when PRINT# or CMD causes an indirect jump via the vector at 800 ($320) to 62217 ($F309), open-channel-for-output. The vector at 800 ($320) may be changed to your own front-end routine. CHKOUT stores the current output device number here. See the reference to CHKOUT on page 186 of the *VIC-20 Programmer's Reference Guide*. BASIC calls CLRCHN after every output to close the channel. Refer to page 191 in the reference guide for details of CLRCHN.
   SETIODEF sets default to 3 at power-on/reset. This location is also used by Kernal output routines to determine the routines for sending the data.
   By using ML or SYS and the SAVE area for registers at location 780 ($30C), you can divert input/output from/to any file number. First OPEN a file to the device. See page 196 of *VIC-20 Programmer's Reference Guide* for details on the OPEN routine.
   To divert input, POKE 781 or LOAD .X with the file number and SYS or use a JSR to 65478 ($FFC6). Location 65478 ($FFC6) is simply a vector which calls the CHKIN routine. A SYS or JSR to location 65508 ($FFE4), the vector pointing to the GETIN routine, will retrieve a byte from the device and place it into .A. Note that the routine CHRIN, location 61966 ($F20E), will do the same for a serial device. When finished, you can use another SYS or JSR to 65484 ($FFCC) to restore the keyboard and screen as the default devices.

To divert output, POKE 781 or LOAD .X with the file number and then SYS or use a JSR to 65481 ($FFC9) (the vector to the CHKOUT routine) to open the output channel. Then SYS or use a JSR to 65490 ($FFD2) to output a byte that is passed in 780 (.A). Again, a SYS or JSR to 65484 ($FFCC) CLRCHN (close-input-and-output) is used to restore the default devices.

See location 186 ($BA) for the current device number.

155	**$9B**	**PRTY**
		(possible user storage)

Tape: character parity.

This location is used to help detect missing dropped bits in tape data. It's also a parity work byte during tape load and save, with bit 0 used to calculate parity. Odd parity is used when the parity bits' total number of 1's, for all 8 data bits, is an odd number.

Parity is simply a way of checking data transmissions, making sure that the data is received correctly.

156	**$9C**	**DPSW**
		(possible user storage)

Tape: dipole switch/byte-received flag.

During tape load, the following values in this location mean that:
1: a byte has been completely received.
0: the computer is waiting for the next byte or is still receiving a byte.

157	**$9D**	**MSGFLG**
		(handy location)

Kernal message control flag.

The following values in this location signify:
128 ($80) = Kernal control messages wanted
64 ($40) = Kernal error messages wanted
192 ($C0) = Kernal control *and* error messages wanted

If bit 7 is off, no Kernal control messages, such as SAVING, FOUND, PRESS PLAY, and so on, will be shown on the screen.

If bit 6 is off, no Kernal I/O ERROR number messages will be displayed.

BASIC calls SETMSG to set this location to 128 ($80) when it is in the READY mode, and to 0 ($0) for RUN mode. BASIC thus preempts the Kernal error messages altogether. BASIC has its own error messages, and prefers them over the Kernal message of I/O ERROR followed by an error number. These error numbers correspond to

those returned by an error during a Kernal operation in the .A register. See the list of BASIC messages at location 49566 ($C19E) and the list of Kernal messages at location 61812 ($F174).

You can also refer to page 203 of the *VIC-20 Programmer's Reference Guide*, but notice that the significance of bits 6 and 7 is reversed, in error.

You could set this location yourself with a POKE statement.

158	**$9E**	**PTR1**
		(possible user storage)

Tape: error log index/filename index/header I.D./out byte.

This location is used by various tape routines for several purposes:

Tape SAVE. Temporary storage for tape I.D. header. The header values are:

1 = relocatable
2 = user data record
3 = nonrelocatable
4 = user data header
5 = end of tape

This I.D. is the first byte of the tape header, except for 2 which is in the first byte of every record and accounts for the fact that the 192-byte tape buffer at 828 ($33C) can contain only 191 bytes of data.

Tape LOAD. Pass 1 error index value. If not zero, it equals two times the number of errors. It indexes into the stack area where error addresses are stored. The maximum value is 61 ($3D), resulting in locations 256–317 ($0100–$013D) possibly used for error pointers, with a maximum of 31 possible errors.

Tape header LOAD. Index into tape buffer during comparison of filename from tape header to filename specified in LOAD. Location 187 ($BB) points to the desired filename.

Tape write from BASIC. Holds character for output during the CHROUT ($F27A) routine's processing to tape.

159	**$9F**	**PTR2**
		(possible user storage)

Tape: pass 2 error pointer/tape buffer filename index.

This location is also used for several tape routine operations, such as:

Tape LOAD. Pass 2 error correction index, and indexes through stack error location address. This is limited to a value no greater than the pass 1 error correction index. See 158 ($9E).

Tape header LOAD. Index into filename in the tape buffer during comparison of filename from tape header to filename specified in LOAD. Location 178 ($B2) points to the tape buffer.

160-162 $A0-A2   TIME
(handy location)

Jiffy clock, realtime clock.

These locations keep a count of the jiffies, one-sixtieths of a second, since power-on. They're reset to zero after 24 hours.

Tape I/O interferes with both accurate clocking and testing of the STOP key; but serial I/O interference, from the disk drive or a printer, for example, is negligible.

The individual locations in this routine have these functions:

160 ($A0) is incremented every 18.2044 minutes
161 ($A1) every 4.26667 seconds
162 ($A2) every .01667 second (one jiffy)

TI$ and TI are not actually variables in BASIC, since they're not stored in the RAM variable pool. Setting TI$ in BASIC (TI$=HOUR$+MIN$+SEC$) calls SETTIM. Assigning TI$ in BASIC (XY$=TI$) calls RDTIM. See pages 198 and 204 of the *VIC-20 Programmer's Reference Guide.*

An ML subroutine of BASIC can examine and change these locations. Any changes will be reflected in the BASIC variable ST.

You can create a digital clock on the screen using this short program:

Program 1-13. Digital Clock

```
10 REM 160
20 INPUT"HRS,MIN,SEC";H$,M$,S$
25 T$=RIGHT$("0"+H$,2)
26 T$=T$+RIGHT$("0"+M$,2)
27 T$=T$+RIGHT$("0"+S$,2)
28 TI$=T$:T0=TI
30 IFTI<T0+60THEN30
35 T0=TI:PRINT"{CLR}{11 DOWN}"SPC(7);MID$(TI$,1,2)
   ":"MID$(TI$,3,2)":"MID$(TI$,5,2)
40 GOTO30
```

The TI numeric variable may not be set, only assigned. A good reference for this is "Timekeeping," by Keith Schleiffer. This article appeared in the February 1982 issue of *COMPUTE!.*

163 $A3   PCNTR

(possible user storage)

Serial: input bit count/Tape: input/output bit count.

Serial. If the high order flag bit is on, this indicates that the last byte has been sent to the serial device.

Tape. Count of bits remaining to be written for a byte during tape write, or bits remaining to be read for a byte during tape read. This location is initialized to 8 before each byte and decremented after each bit is written or read. During tape write, it is time to set up the parity bit to be written when this location is decremented to 0. When this location's value reaches −1, it's time to prepare the next byte to be written. That's when this location is reset to 8. During tape read, when this is decremented to −1, the parity bit has just been read and it's time to check for a parity error. Then it's ready to read the next byte, so the location is reset to 8.

164 $A4   FIRT

(possible user storage)

Serial: input byte/cycle counter/Tape: dipole number.

Serial. The input byte read in during a serial LOAD or VERIFY.

Tape read/write. Flag to indicate which dipole has been processed. Set to 1 if you just processed the first half of the dipole or to 0 if you just processed the second half.

165 $A5   CNTDN

(possible user storage)

Tape: block sync countdown/Serial: countdown.

Tape. Countdown for block synchronization. During tape SAVE, is a counter for block countdown characters written to tape before each block's data begins. The location is initialized to 9 for each block, so each block contains 9 countdown characters. The countdown characters are: 9, 8, 7, 6, 5, 4, 3, 2, and 1. For the first block, the countdown characters have their high order bit on. The high order bit is off for the second block. Later, during tape load operations, the block countdown characters can be used to determine whether block 1 or block 2 is being read.

Serial. Countdown from 8 to 0 of bits left in byte to be sent.

166 **$A6** **BUFPNT**

(handy location)

Tape: count of characters in the tape buffer.

This location is used to count bytes when writing the tape header, and when accumulating BASIC program output for the tape buffer.

You can POKE 166,191 to force a 192-byte buffer to tape, regardless of the actual amount of data in it. Refer to the tape buffer at location 828 ($33C) for an explanation of the disparity between the 192-byte buffer and the 191 bytes of data it contains.

You can create a tape of 100 191-byte blank records, which can be updated later, by entering this two-line routine:

Program 1–14. Blank Record Files

```
10 REM 166
20 REM CREATE A TAPE FILE OF 100 191-BYTE BLANK RE
   CORDS FOR LATER OVERLAYING.
30 OPEN1,1,1,"100RECS"
40 FORX=1TO100:POKE166,191:PRINT#1:PRINTX;:NEXT:CL
   OSE1:END
```

See 178 ($B2) for the tape buffer pointer and restrictions.

167 **$A7** **INBIT**

(possible user storage)

Tape: write leader count/read block reverse counter.
RS-232: bit 0 is the temporary storage for input bit.

Tape. Write leader length counter.

When writing leader dipoles to tape, this location is used as a counter for an inner loop that counts down to 0 each time the loop is performed *before* the outer loop decrements its counter. This results in a total number of leader dipoles written equal to the value of INBIT multiplied by the value in RIPRTY, plus one. See 171 ($AB) RIPRTY for the second value.

This byte is set to 0 before writing the leader for the header or a program. It's set to 128 ($80) before writing the leader between blocks. See location 146 ($92) for a short description of a dipole. Leader dipoles contain zeros, but data dipoles are reverse dipoles, which contain a zero, then one, or vice versa.

During tape load operations, this location indicates which block is currently being loaded. If its value is 2, then the first block is loading; if its value is 1, it's loading the second block; and if its value is 0, all blocks are loaded.

168 $A8   BITCI
(possible user storage)

Tape: error flags, 0 = no errors/long word marker switch.
RS-232: input byte bit count and output new byte.

Tape. During SAVE, this is a switch for word marker write. If the value is 0, it's writing the long time for a word marker dipole. If 1 is the value, the long time for a word marker dipole has already been written.

During LOAD, if the value is not zero, the byte just read is considered in error. An example would be if a parity error has occurred.

RS-232. The input byte bit count is derived from location 664 ($298), BITNUM.

169 $A9   RINONE
(possible user storage)

Tape: dipole balance counter/medium word marker switch.
RS-232: input flag for checking for a start bit.

Tape. During SAVE, this byte acts as a switch for word marker write. If its value is 0, then write the 1 time, medium time, for a word marker dipole. If its value is 1, the 1 time for a word marker dipole has already been written.

During LOAD, a 0/1 means a balanced counter. Each time a 0 dipole is read, this value is incremented, and each time a 1 dipole is read, this value is decremented. When reading the all-0 leader dipoles, location 150 ($96) is set when this value reaches 16. The maximum value this location can contain is 126. When actually reading data bytes, this location is initialized to 0 before each byte. Since each data bit contains one dipole that is a 0 and another that is 1, this counter should be zero after each bit is read. If not, the byte error flag in 182 ($B6) is set.

See location 146 ($92) for a brief description of a dipole.

RS-232. A value of 144 ($90) in this byte indicates no start bit received, while 0 means a start bit was received.

170 $A4   RIDATA
(possible user storage)

Tape: input status flags, sync countdown.
RS-232: byte assembly.

If the SAVE starting address is greater than the ending address, this location is set to 128 ($80), indicating invalid parameters.

Tape. During tape LOAD, the action taken for the byte just read is:
- If 0, then it's waiting for the first block countdown character to arrive. Note: This location is initialized to 0 before reading of the first header block and before reading of the first program block.
- If 1–15, block countdown characters are being read.
- If 64 ($40), valid block countdown characters have arrived, and the byte received is treated as a valid data byte.
- If 128 ($80), the first block has been loaded and a search is proceeding for the second block.

RS-232. This is used as a byte assembly area to be stored where ($F7) points, as well as for detecting framing error and BREAK.

See location 247 ($F7), receive buffer pointer, for additional RS-232 related locations.

171 $AB   RIPRTY
(possible user storage)

Tape: write leader counter/read checksum comparison.
RS-232: input parity/checksum bit storage.

Tape. When writing leader dipoles to tape, this is used as a counter for an outer loop that must count down to −1 before the routine will stop writing leader dipoles. This results in a total number of leader dipoles written equal to the value of INBIT multiplied by the value in RIPRTY, plus one. See location 167 ($A7), INBIT for its value.

This location is set to 105 before writing the header leader, to 20 before writing a leader for the first block, and to 0 for a leader between blocks.

During tape LOAD, once both copies of a program are read and the load is considered complete, this byte then computes the parity over all bytes loaded. This parity or *checksum* should be the same as that just read as the last byte of the program on tape, which was similarly computed over all bytes saved to tape. If it's not the same, then it's set to checksum error status. This checksum computation is done both for the header and the program. The checksum that was recorded during the save at the end of the first copy does not appear to be used during the tape load. Only the second copy has a checksum comparison performed.

172–173 $AC–AD   SAL
(possible user storage)

Tape/Serial: start address for LOAD/SAVE/VERIFY.

Copied here from 193 ($C1), the pointer to start of I/O area, this location is then used as a pointer through the data, and

incremented as the data is sent or loaded. Location 174 ($AE) contains the end+1 address to be attained in this pointer. These locations may be pointed to any area when calling Kernal LOAD and SAVE routines. Before each byte is written to tape, right after the word marker long dipole has been written to tape, a check is made to see if 173 ($AD) is above 127 ($7F). There are two ways that location 173 ($AD) can be set above 127 ($7F).

   Once the memory area and the checksum have been written for a block, BLKEND at location 64518 ($FC06) is executed, which sets the high order bit on to write an interblock leader. The second way 172 can be set above 128 is if you specify to save from an area over 32767 ($7FFF). When 173 ($AD) is found with its high order bit on, WRTN1 ($FC95) executes to write the interblock leader to tape if the first block has just finished, or turns off the tape motor if the second copy is complete. Thus, a save from 32768 ($8000) causes, after a valid header has been written to tape, interblock leader dipoles to be written, and the tape motor to be turned off. Through this dual use of 173 ($AD), both as block end indicator and as the high order byte of the save area, it's impossible to save areas from 32768 ($8000) on up.

   The BASIC tape buffer at 828 ($33C) is used for BASIC LOAD, SAVE, and VERIFY tape headers. This location is restored from the value saved in location 193 ($C1) at the end of the operation.

   Screen management routines save this pointer, use it as a work pointer, and then restore it.

   See Appendix F for a technique that you can use with BASIC to save/load whole storage blocks. This technique does not require that an ML routine be preloaded.

   A type 3 tape header will always load where the tape's start address specifies, no matter what a secondary address indicates. This header I.D. is created when the secondary address was 1 or 3 when creating the tape. If the secondary address was 0 or 2, an I.D. of 1 is used. This is relocatable. See the explanation at locations 829–830 ($33D–33E) for how to read and modify the tape header before it's acted upon by the LOAD routine. Also see locations 193 ($C1) and 195 ($C3).

174-175 $AE-AF   EAL
                                                    (possible user
                                                    storage)

Tape: ending address for LOAD, SAVE, and VERIFY.
Serial: loading address for LOAD/VERIFY, end address plus 1 for SAVE.

   This location is initially set by the Kernal SAVE routine from parameters passed to it.

RAM saved to disk has a pointer that indicates where it was saved from. This pointer is after the next sector pointer and is written to disk from the start-saving pointer located at 172 ($AC). During a disk LOAD, the pointer at 172 ($AC) is read from the device. If the LOAD specifies a secondary address of 1, the pointer at location 174 is used as the starting pointer for the loaded data. Otherwise, the data is loaded wherever the parameters passed to the LOAD routine are pointing. If a BASIC program is being loaded, this information is obtained from the contents of location 43–44 ($2B–2C).

This pointer is incremented as data loads, and at the load's completion, it points to the end of the loaded RAM.

The ending address, plus one, of a tape save is stored at this location, after the length of the loaded data is added to the value in 195 ($C3). Once the tape buffer is full, BASIC PRINT# causes the Kernal to set this pointer to the end of the tape buffer 828 ($33C) and write the block to tape.

Screen management routines save this pointer, use it as a work pointer, and then restore it.

176 $B0   CMP0
(possible user storage)

Tape: dipole timing adjustment values.

Timer 1 of VIA 2 at location 37156 ($9124) is adjusted with this value during the reading of tape.

During tape LOAD, this location is used as a factor in computing the values to set the adjustable timebase for the next dipole read. See location 146 ($92). If this location's value is greater than 0, that amount of time is added to the timebase. If it's less than 0, that amount of time is subtracted.

177 $B1   TEMP1
(possible user storage)

Tape: dipole timing timer 2 difference.

Timer 1 of VIA 2 at location 37156 ($9124) is adjusted with this value during the reading of tape.

During tape LOAD this location's value is equivalent to the tape dipole time *minus* the time between reading timer 2 and resetting timer 2.

This is also a one-byte field that the two-byte timer 2 has been compressed into. Bits 9–2 of the timer 2 value since it was last set are stored in bits 7–0 of this location.

178-179 $B2-B3 TAPE1 60/3 ($3C/03)
(handy location)
Tape: pointer to tape buffer.

Initialized by power-on/reset, this location normally points to 828 ($33C). BASIC uses the tape buffer for all program tape data I/O. LOAD and SAVE only use the tape buffer for tape header records. Serial devices use the buffer at 512 ($200), but filename information is sent from the location the pointer at 187 ($BB) indicates, the length specified in 183 ($B7). RS-232 allocates its 512 bytes of buffers (256 in, 256 out) from the top of available RAM.

This pointer must contain an address greater than or equal to 512 ($200) or an ILLEGAL DEVICE NUMBER error will occur. This error is checked in the TPBUFA routine, location 63565 ($F84D).

The TAPEH routine at 63463 ($F7E7) clears the 192 bytes of the buffer to spaces before building the tape header, so a filename shorter than 187 bytes is padded with blanks. A filename may contain an ML program after or instead of the name. See location 187 ($BB).

The pointer to the tape buffer can be used as the operand of an ML indirect JMP off of the zero page location to any ML routine that you've stored in the buffer.

180 $B4 BITTS
(possible user storage)

Tape: miscellaneous flags/RS-232: various uses.

Tape. Nonzero means tape routines are ready to receive data byte. This is reset to zero between blocks.

RS-232. Transmit bit count out, timer enable flag, parity, and stop bit manipulation.

Output bit count is derived from location 664 ($298), BITNUM.

181 B5 NXTBIT
(possible user storage)

Tape: flag for currently reading data or leader.
RS-232: next bit to be sent or EOT.

During tape LOAD, this address indicates where the tape LOAD routine is currently reading data. If its value is not zero, the tape is before a block of data, waiting for a word marker at the end of the leader bits. If its value is zero, then bytes of data are being read from the block. This location is set to zero once a word marker has been received after location 150 ($96) has been set.

This location is also used by RS-232 NMI routines to hold the next value for VIA 1 control line options at location 37148 ($911C).

182 $B6   RODATA
(possible user location)

Tape: accumulator for number of read errors.
RS-232: byte disassembly area for buffer pointed to by 249 ($F9).

This holds the byte currently being sent from the RS-232 buffer.

    See location 249 ($F9) for additional related locations.

    Tape. During tape LOAD, this flag is set to nonzero if the byte just read was considered in error. This error can be a parity error, a dipole mismatch, or a verify error.

183 $B7   FNLEN
(handy location)

Number of characters in filename.

    Only the first 16 characters of a filename will show in a FOUND message, so the message is never longer than a 22-column line.

    Disk directories allow a 16-or-less character filename. When merging disk files, 16 can be too long since the command syntax allows only 40 characters. Three names with delimiters must be included in these 40 characters. Because of this, it's a good idea to use shorter names or rename a file before merging.

    If you have tape files with ML programs saved within them, there is no corresponding method for saving ML code beyond 16 bytes in disk program headers. Disk filenames may contain a SHIFTed space which ends the name as though a quote were placed there. The remainder of the filename is used as comments. For example, entering SAVE"PROGRAM.3K",8 (where . is a SHIFTed space) causes the filename to appear as "PROGRAM"3K in the directory. You may reference the file as PROGRAM, or as its full name by once again including the SHIFTed space in the name. A second file named "PROGRAM"8K+ would not be found unless the shifted space and 8K+ were included in the name.

    If OPEN, LOAD, or VERIFY tape doesn't specify a filename, this location will contain a zero after opening, making the filename pointer at 187 ($BB) irrelevant. Disk always requires a full or generic name, and this location will always be greater than zero. If a completely generic name is used for disk, for example *LOAD"*",8*, the length at this location will be 1. The * is only one character.

    When RS-232 is opened, there can be up to four characters in the filename which are copied to location 659 ($293) through 662 ($296). These correspond to the control register, command register, and nonstandard bit-timing values.

    A value is stored here by SETNAM from .A when called.

    See related locations 187 ($BB) and 828 ($33C).

This location and 187 ($BB) can be used to retrieve the filename for use in a program. See the description of the technique at location 187 ($BB).

(By some clever manipulations, a tape filename may theoretically be up to 250 bytes long. I haven't seen this done, but routine TAPEH is the place to start investigating if you'd like to pursue the idea.)

184	**$B8**	**LA**
		(handy location)

Current logical file number being used.

A maximum of ten files can be open at any one time. The file number can range between 1 and 255.

If the file number is greater than 127, a linefeed character is sent to the file following any carriage returns.

BASIC's OPEN and the OPEN routine at location 62474 ($F40A) use this number to build the file number table at location 601 ($259).

SETLFS is used to set this location, as well as locations 185 ($B9), and 186 ($BA) from the contents of .A, .X, and .Y.

Whenever I/O needs to change the current channel for input or output, the file number is passed to CHKIN or CHKOUT, unless the device is the keyboard or screen with no other channel currently open. This location, 185 ($B9), and 186 ($BA) are then set by pulling the corresponding entries from:

601 ($259) LAT File number table
611 ($263) FAT Device number table
621 ($26D) SAT Secondary address table

This location serves primarily as an index into these tables so that the secondary address and device number for a file can be remembered by the Kernal. Devices that support multiple open files at the same time—for example, a disk—differentiate internally between files by the secondary address, not the file number. So, OPEN 4,8,15 is perfectly acceptable to communicate with the disk DOS. The file number need not be 15, except for convenience sake.

See locations 152 ($98), 153 ($99), and 187 ($BB). Also see Appendix D for a device, secondary address, and status code table.

185	**$B9**	**SA**
		(handy location)

Current secondary address being used.

The valid range of this number is 0–31 for serial devices and 0–127 for nonserial devices. 0, 1, and 15–31 have special meanings for DOS. Use 2–14 for disk data files.

The keyboard and screen ignore this secondary address.

A secondary address for tape signifies read (0) or write (1). EOT (2) can be added to either. This number is *not* the tape header I.D. that is stored on the tape. See location 828 ($33C) for details of the tape header I.D. The *VIC-20 Programmer's Reference Guide* has an error regarding tape secondary addresses. An odd secondary address results in a nonrelocatable program; an even secondary address results in a tape I.D. header of 1, indicating a relocatable program.

By adding two to the secondary address, an end-of-tape (EOT) header is written at the end of the file.

For serial devices, the Kernal ORs the secondary address with 96 ($60), giving a high order nybble of 0110. When listen-with-attention is sent to these devices, the secondary address is ORed with 32 ($20), resulting in a high order of 0010 in binary.

The disk tells which files are open by the secondary address, not the file number. When loading a program from disk, a secondary address of 0 is used by the Kernal, and a 1 when saving. A secondary address of 15 for disk is the DOS communication channel.

The printer determines the character set to be used by the specified secondary addresses.

See 621 ($26D) SAT secondary address table.

See 184 ($B8) for list of related fields.

186 $BA   FA
(handy location)

Current device number being used.

This location is also called the *primary address* in some documentation.

VIC-20 devices are:

0	keyboard
1	tape
2	RS-232/user port
3	screen
4–5	printer
8–11	disk
4–31	could also be any serial device.
	See location 611 ($263) FAT device number table.
	See locations 184 ($B8), 185 ($B9), 153 ($99), and 154 ($9A).

187—188 $BB—BC   FNADR
(handy location)

Pointer to the current filename.

If an OPEN, LOAD, SAVE, or VERIFY for tape doesn't specify a filename, then location 183 ($B7), the length of the filename, contains zero, and this pointer is unpredictable. However, location 833

55

($341), which is part of the tape buffer, contains the filename after an OPEN for input of an unspecified tape file, but not after any subsequent tape I/O.

Disk always requires a full or generic name, so the filename length in location 183 ($B7) will always be greater than zero.

When RS-232 is opened, there may be up to four characters in the filename which are copied to locations 659 ($293) through 662 ($296) and correspond to the control register, command register, and nonstandard bit timing values.

The TAPEH routine, which builds the tape header, clears the 192 bytes of the tape buffer to spaces before creating the tape header. Any filename shorter than 187 bytes is padded with blanks. A filename may contain an ML program after or instead of the name. The UNNEW technique mentioned in location 43–44 uses this method. Another reference is "Saving Machine Language Programs on PET Tape Headers," by Louis Sander, which appeared in the July 1981 issue of *COMPUTE!*. There is no corresponding method for saving ML code beyond 16 bytes in disk program headers.

If a tape being read is opened without a filename specified, the tape filename will be in the tape header at location 833 ($341) after OPEN, but not after any subsequent tape I/O.

See related locations 183 ($B7), 178 ($B2), and 828 ($33C).

189	**$BD**	**ROPRTY**
		(possible user storage)

RS-232: send parity calculation work byte.
Tape: byte just read or shifting byte currently being written.

During SAVE, this byte is saved to tape. After one bit is written to tape, the byte is shifted right one bit, and the procedure repeats until all eight bits have been written.

During LOAD, this location holds the byte that has been read after being built in 191 ($BF).

190	**$BE**	**FSBLK**
		(possible user storage)

Tape: which copy of block remaining to read/write.

• A 2 in this location means both copies of block remain to save/load.
• 1 in this location means that the last copy of block remains to save/load.
• A 0 means that both copies of block are done.

191 $BF   MYCH

(possible user storage)

Tape: input byte currently being constructed.

During tape read, the bits read from tape are rotated into the high order to low order bits to build a byte. Once eight bits have been received, the byte is considered complete.

192 $C0   CAS1

(handy location)

Tape: motor interlock switch.

A nonzero value here, which is possible only if some tape buttons are pressed down, prevents any change of tape motor switch. The IRQ interrupt handler at location 60095 ($EABF) normally sets this location to 0 if no tape buttons are pressed down.

During tape read or write, this is set to nonzero once a tape button has been pressed and will be reset to zero once the tape action is completed. A zero at this location, which is possible with either some or no buttons down, allows the tape motor to be turned on. This is done within the normal IRQ interrupt routine if location 37148 ($911C) has bits 2 and 3 on.

The effect of placing values in location 37148 ($911C) using POKE 37148,(PEEK(37148) AND 241) OR n, where n is:

- 0,2,4,6 Stops the motor
- 8 No change
- 10 Stops the motor
- 12,14 Starts the motor

This location has no control over tape motor settings outside of the default IRQ interrupt handler.

193-194 $C1-C2   STAL

(handy location)

Tape/Serial: pointer to the start of the I/O area.

These locations are initially set in SAVE/LOAD from parameters passed to them.

This points to the area being loaded or saved, such as the tape buffer or RAM address.

See location 174 ($AE) for disk LOAD. A SAVE to disk writes this initial value as the address the RAM was saved from. No ending address is sent.

Locations 195–196 ($C3–C4) are copied to this location after the LOAD is completed, or before the LOAD is actually begun if a nonrelocatable, I.D. type 3, tape header is found. Finally, this location is copied to 172–173 ($AC–AD).

This pointer is used by INITMEM to find the lowest RAM. The screen memory location is determined and set up from this test.

It's also used by TESTMEM as a pointer while performing a nondestructive test of every RAM bit's quality.

195-196   **$C3-C4**   **MEMUSS**
(handy location)

Pointer to the RAM area being LOADed.

The start-of-LOAD address is saved at these locations by the LOAD routine from parameters passed to it.

This is overwritten with the starting address of the saved data from the tape header if a tape header I.D. 3 is found. It is also overwritten if the secondary address specified for LOAD was not zero. See location 829–830 ($33D–33E) for a method of circumventing the type 3 header I.D.

Power-on/reset or the RUN/STOP–RESTORE key causes the 16 default vectors in ROM at 64877 ($FD6D) to be copied into locations from 788 ($314) to 818 ($332). This pointer location is also used as a base address during this process.

The VECTOR routine at 64855 ($FD57) can be used to read or change these default vectors.

197   **$C5**   **LSTX**
(handy location)

Matrix coordinate of last key pressed. 64 if none pressed.

Used for stabilizing the current key in location 203 ($CB).

This value is set with every IRQ interrupt. The advantage of using this location, or 203 ($CB), is that near-instantaneous recognition of the key and that near-simultaneous pressing of keys can be prioritized. In an IRQ wedge preamble, this value will need to be used since ASCII conversion of the key is done later in IRQ code.

• The SCNKEY routine translates this key value into ASCII by picking up the nth value in the table, where n is the contents of this location with SHIFT, Commodore, and CTRL keys determining the table used.

The following statement will allow an in-program pause when displaying more than a screenful of information to the screen.

WAIT 197,64

Pressing any key other than SHIFT, RESTORE, CTRL, or the Commodore key will halt the program until the key is released.

See also 653 ($28D) for SHIFT/CTRL/Commodore key flags, and a routine for a locking pause key. Refer to location 37153 ($9121) for data register contents.

The values returned for each key pressed are below:

Table 1-1. Keycode Values

Key pressed	Code	Key pressed	Code	Key pressed	Code	Key pressed	Code
←	8	A	17	Q	48	Z	33
1	0	S	41	W	9	X	26
2	56	D	18	E	49	C	34
3	1	F	42	R	10	V	27
4	57	G	19	T	50	B	35
5	2	H	43	Y	11	N	28
6	58	J	20	U	51	M	36
7	3	K	44	I	12	,	29
8	59	L	21	O	52	.	37
9	4	:	45	P	13	/	30
0	60	;	22	@	53	CRSRDN	31
f1	39	+	5	*	14	CRSRRT	23
f3	47	-	61	↑	54	DEL	7
f5	55	£	6	=	46	HOME	62
f7	63	None	64	RETURN	15	Space	32

Values not placed in location 197 ($C5), but reserved:

Commodore 40   CTRL   16   rtSHIFT   38   ltSHIFT   25

The STOP key is represented by a code of 24 in this location. You would not be able to detect this in a program without disabling the STOP key first. See the vector description at 808 ($328) for a description on how to disable the STOP key.

The following program can be used to examine the various representations of a character entered on the keyboard:

Program 1-15. Keyboard Character Values

```
10 PRINT"{CLR}{RVS}PRESS ANY KEY"
20 GETK$:IFK$=""THEN20
30 PRINT"KEY="K$" MATRIX="PEEK(197)"ASC="ASC(K$)"S
   HIFT="PEEK(653):GOTO20
```

See location 631 ($277) for a summation of keyboard-related RAM locations.

198   **$C6**   **NDX**

(handy location)

Number of characters (0-10) in the keyboard buffer at 631 ($277).

The INPUT and GET statements pull characters that haven't yet been used from the keyboard buffer. By putting a zero in this

location, you can cause any current contents of the keyboard buffer to be ignored.

To wait for the user to press a key to signal that the program can continue, you could enter:

POKE 198,0:WAIT 198,1:POKE 198,0.

The zero POKEs insure that previous keystrokes are not currently in the buffer, and clear out the key that was pressed to continue the program.

By storing a number in this location and putting the desired characters in the keyboard buffer, you can simulate keyboard entry. Be sure to include the ending carriage return in this count and the buffer.

By putting multiple carriage returns in the keyboard buffer, clearing the screen, carefully placing messages on the screen, homing the cursor, setting this location to the number of carriage returns, and ending the program, the lines on the screen will be read just as though they were entered at the keyboard. Careful placement of the lines on the screen is crucial in using this feature. You must avoid BASIC messages. See the sample dynamic keyboard routine at location 277 ($115)

You can increase this number to as high as 15 since locations 641–645 ($281–285), the end-of-RAM-pointer, start-of-RAM-pointer, and serial timeout flag, are not normally used after BASIC is cold started. The maximum length of the keyboard buffer specified in 649 ($289) would then be set to a value of 15.

The RUN/STOP key zeros this number, clearing the keyboard buffer. RUN/STOP–RESTORE resets 649 ($289) back to ten.

GET reads from the keyboard buffer using the GETIN routine. OPEN 3,3:INPUT#3 will read from the current screen line up to the end of the line or to a carriage return.

The Kernal routine LP2 at location 58831 ($E5CF) is used to get characters from the keyboard buffer and maintain the count of characters in the buffer.

199   **$C7**   **RVS**
(handy location)

Flag for reversed screen characters.

If this location is set to 18 ($12), characters will appear reversed on the screen. This is done by ORing the character with 128 ($80), which causes the reverse character set to be used. See location 32768 ($8000).

This flag is set on entry of a RVSON (reverse on) key, and cleared when a carriage return or RVSOFF (reverse off) is entered.

This location may be POKEd directly, but it may need to be POKEd again to compensate for the factors that return it to zero.

200 $C8   INDX
(handy location)

Pointer to the end of line for input.

This location indicates how many columns in a logical line are nonblank.

Its value originates from the current screen line logical length in location 213 ($D5) and is decremented to the last nonblank position on the screen line.

See location 201 ($C9) for line/column/cursor pointer summation.

201–202 $C9–CA   LXSP
(handy location)

Current cursor INPUT logical X-Y position (line, column).

This location is used by GET and INPUT, the GETIN and CHRIN routines, when reading the screen.

The logical line number range is from 0 to 22, while the column number can be from 0 to 87. The logical line could contain up to four physical lines.

The screen line link table at 217 ($D9) flags physical lines not continued.

Here's a summation of the page 0/1 locations used by the Kernal screen editor and other routines:

    200 ($C8) End of the text on current line
    201 ($C9) Logical line number
    202 ($CA) Column of cursor
    204 ($CC) Cursor blink switch
    206 ($CE) Character under cursor
    207 ($CF) Character blink status
    208 ($D0) Screen length or keyboard input
    209 ($D1) Pointer to start of line in screen RAM
    211 ($D3) Cursor displacement with screen RAM line
    213 ($D5) Logical line length
    214 ($D6) Physical line number
    243 ($F3) Pointer to start of line in color RAM
    647 ($287) Original color under cursor
    658 ($292) Screen scroll enable flag
    The screen line link table at locations 217–241 ($D9–F1) includes examples of using some of these locations.

203 $CB   SFDX
(handy location)

Matrix coordinate of current key pressed. 64 if none.

See Table 1-1 at location 197 ($C5).

See also 653 ($28D) for SHIFT/CTRL/Commodore key flags.
See also 37153 ($9121) for the actual data register contents.

204 $CC   BLNSW
Cursor blink switch. 0=flash, nonzero=quiet.

This location insures that the cursor won't flash when characters
are in the keyboard buffer.
See location 631 ($277) for a summation of keyboard-related
RAM locations.

205 $CD   BLNCT
Cursor countdown before blink.

Normally set for 20 jiffies between blinks of the cursor, as
counted by the interrupt routine IRQ, this location makes the cursor
blink three times per second. Every time the cursor blinks, locations
206 ($CE) and 647 ($287) are updated. Turning the reverse on and
off for the character under the cursor causes the blinking image.

206 $CE   GDBLN
(handy location)
Character under cursor in screen POKE code.

Screen POKE codes (*P*) can be converted to ASCII (*A*) by the
following routine, with *R* being set to 1 if the screen POKE code was
for a reverse character. To add this routine to a program of your
own, use a GOSUB command to line 50.

Program 1–16. Converting POKE Codes to ASCII Code Values
```
50 P = PEEK(206) : R= 0
60 IF P > 127 THEN R = 1 : P = P AND 127
70 IF P < 32 OR P > 95 THEN A = P + 64 : GOTO 100
80 IF P > 31 AND P < 64 THEN A = P : GOTO 100
90 A = P + 32
100 RETURN
```

Remember that the values returned will be for the character
beneath the cursor. Many times this character will be the space
(ASCII value of 32), so to receive a different value the cursor will
have to be placed atop another character.
See Appendix C for a character code chart. Every time the
cursor is blinked, determined with location 205 ($CD), the cursor
countdown, this location and 647 ($287) are updated.

207 $CF   BLNON
Cursor blink status. 1=reversed character, 0=nonreversed.

This location indicates whether the current cursor blink has reversed or unreversed the character under the cursor.

The use of some of the cursor control fields is shown in the following routine. The cursor is turned on during a GET operation, with a different blink rate than normal; a default multicharacter entry is placed under the cursor; and your entry is collected until a RETURN is pressed. The POKEs that cause the cursor to blink must be left on the same program line as the GET statement. The value of BS (blink speed) can be changed to obtain the correct value for your own program.

Program 1–17. Cursor Control Fields

```
10 REM LOC 207 (FOR 205,204,207,211)
20 REM INPUT-LIKE GET, (EXCEPT DEL & INST) UP TO 2
   55 CHARS, COMMAS OR COLONS OK
30 BS=3 :REM BLINK SPEED, ADJUST TO YOUR PROGRAM N
   EEDS
40 D$="DFLT":PRINTD$;:POKE211,0: REM DEFAULT
50 POKE207,0:POKE204,0:POKE205,BS:GETK$:REM BLINK
   {SPACE}FAST CURSOR
60 IFK$=CHR$(13)THEN140: REM ALL DONE WITH ENTRY
70 IFK$<>CHR$(133)THEN110:REM INPUT ESCAPE IS F1 K
   EY
80 POKE204,0:POKE207,0
90 X=PEEK(209)+PEEK(210)*256+PEEK(211): POKEX,PEEK
   (X)AND127: REM TURN OFF ANY REVERSE
100 END
110 PRINTK$;:REM SHOW KEY
120 S$=S$+K$: REM BUILD THE ENTRY STRING
130 GOTO50: REM GET THE NEXT KEY
140 IFLEN(S$)=0THENS$=D$:GOTO160: REM USE DEFAULT
    {SPACE}IF NO ENTRY
150 X=PEEK(209)+PEEK(210)*256+PEEK(211): POKEX,PEE
    K(X)AND127:PRINT:REM OFF ANY REVERSE
160 PRINT"{RVS}"S$;LEN(S$):S$="":GOTO40: REM SHOW
    {SPACE}THE STRING AND GET NEXT
```

208　　　　　　　　$D0　　　　　　　　CRSW

Flag indicating if input from screen or keyboard.

Zero in this location indicates input from the keyboard. Other values indicate input from the screen. Possible values are 0, 21, 43, 65, or 87.

This address is also used to save the current line length from location 213 ($D5) after a carriage return while getting it from the keyboard buffer.

See location 631 ($277) for a summation of keyboard-related RAM locations.

(handy location)

Pointer to the start of the logical line that the cursor is on.

    This pointer indicates the line in screen map RAM. When the line is a continuation of a previous line, this pointer contains the location of the start of the continued line. The MSB of this pointer is also the screen page that the logical line is in.

    The line pointed to and the values in this location are:

Logical line	Pointer DEC	209 ($D1) LSB value HEX
01	00	$00
02	22	$16
03	44	$2C
04	66	$42
05	88	$58
06	110	$6E
07	132	$84
08	154	$9A
09	176	$B0
10	198	$C6
11	220	$DC
12	242	$F2

(Screen line 12 crosses a page boundary, so one is added to the MSB in location 210, $D2.)

13	08	$08
14	30	$1E
15	52	$34
16	74	$4A
17	96	$60
18	118	$76
19	140	$8C
20	162	$A2
21	184	$B8
22	206	$CE
23	228	$E4

    See Appendix E for details of the relocatable screen map.

    Location 243–244 ($F3–F4) is set to the address of the corresponding line in the color map.

    Location 211 ($D3) contains the number of the column that the cursor is on at the time.

    See location 201 ($C9) for a line/column/cursor pointer summation.

211   **$D3**   **PNTR**
(handy location)

Cursor position within the logical screen line.

The range of this location's value is 0–87 ($0–57).

This is typically used as an index added to location 209–210 ($D1–D2), pointer to the start of the logical line that cursor is on. It's also used for screen editing, input, and output routines. You can use this location to position the cursor wherever you wish. For example:

POKE 211,18

will position the cursor on column 19. Refer to location 207 ($BF) for a method of turning on the cursor during a GET statement.

The PLOT routine at 58634 ($E50A) will set or read this value from/to .Y and set or read the physical line number in location 214 ($D6) from/to .X. Refer to location 201 ($C9) for a line/column/cursor pointer summation.

212   **$D4**   **QTSW**
(handy location)

Flag to indicate if within quote marks.

A 0 value in this location indicates not within quotes, a 1 indicates within quotes.

The QUOTECK* routine sets/unsets this flag.

A carriage return turns off the quote mode flag.

When printing to the screen, if a control character such as a cursor down, color command, or a clear command is within quotes, the control character is printed and not acted upon. When the string is sent to the screen but not enclosed in quotes, the control codes *are* acted upon.

A special case is the INST/DEL key: An insert stores the INST control in the string, but hitting the RETURN key or SHIFT and then RETURN, followed by positioning the cursor over the original string and using an insert, allows insertion within the string. Pressing the RETURN or SHIFT and RETURN keys, followed by positioning the cursor over the original string and using an insert, then a delete, causes the DEL control to be stored in the string. Normally, DEL deletes a character in the string and doesn't store a control for itself.

You can also insert control codes by entering a blank for them in the string, pressing RETURN or SHIFT and RETURN, pressing CTRL and RVSON together, then going back to the string and typing the letter or SHIFTed letter.

This flag is maintained by the routine at 59064 ($E6B8) when reading the keyboard buffer or writing to the screen.

See location 788 ($314) for an example of modifying this location to escape from the quote mode.

A related location is 216 ($D8).

See location 631 ($277) for a summation of keyboard-related RAM locations.

213 $D5   LNMX
(handy location)

Current screen line logical length.

This location determines when to start a new screen line or expand the current logical line with another physical line. Possible lengths are 21, 43, 65, or 87.

Screen scrolling uses this location to scroll the entire logical line.

Some routines use this to determine the end-of-line position for backward scanning of the line.

Location 200 ($C8) is derived from this location's value.

See location 201 ($C9) for a line/column/cursor pointer summation.

214 $D6   TBLX
(handy location)

Cursor: current physical screen line cursor is on.

The possible positions for the cursor range from line 0 to line 22. You can place the cursor on any line, simply by entering a POKE statement. For example:

POKE 214,7

places the cursor on line 8. See location 207 ($BF) for a way to turn on the cursor during a GET statement.

The PLOT routine at location 65520 ($FFF0) may be used to set/read this and location 211 ($D3).

See location 201 ($C9) for a line/column/cursor pointer summation.

215 $D7   ASCII*
(handy location)

ASCII value of last key pressed.

Temporarily used by SCRNOUT* to hold the character going to the screen.

Tape. Checksum of bytes in the current block being written and the value of the first dipole being read. The value can be either 0 or 1.

216 $D8   INSRT
(handy location)

Number of outstanding inserts remaining.

As the INST/DEL key is pressed to indicate an insert, the

Kernal screen editor shifts the line to the right, allocates an additional line if necessary and possible, updates the screen line length in location 213 ($D5), and adjusts the screen line link table in 217 ($D9).

You can POKE 216,0 within a program to turn off the insert mode. Disabling the quote mode like this can also be done by entering a carriage return or SHIFT and RETURN.

Characters that fill an inserted space are treated as though they were enclosed in quotes.

Refer to location 788 ($314) for an example of how to modify this location to escape from the insert mode.

See location 631 ($277) for a summation of keyboard-related RAM locations.

217-241 $D9-F1   LDTB1
Screen line link table.

This table contains one byte per screen line used to indicate the page of memory in RAM that any particular screen line is in. It also contains a flag bit that is set *on* when the physical line is the first or only physical line in an up-to-four-line logical line. The byte for the first physical line of a four-physical-line logical line would have the high-order bit set, and the three following bytes would not be set, to indicate continued lines.

The page number of the screen line is used in conjunction with the displacement table of screen lines (LDTB2) to obtain the location of any byte within the screen map in RAM. Be sure to mask out the high order bit with PN=PEEK(X)AND127. LDTB2 is at 60925 ($EDFD) and contains a byte for each screen line that is the offset within the screen page the line starts on. Using this displacement, the screen page, and locations 214 ($D6) and 211 ($D3), any specific byte in the screen map RAM can be addressed.

Location 209-210 ($D1-D2), the pointer to the start of the physical line that the cursor is on, is kept current using these tables.

You can use these tables to quickly determine the correct address of a new line, last character on a line, or any particular screen position. Besides using these tables and pointers directly, a general-purpose routine can be used. For example, if you wanted to place a character on the seventh physical line, in the fourth column you could enter:

POKE 214,7-1 : REM physical line number relative to zero
POKE 211,4-1 : REM physical column number relative to zero
SYS 58759 : REM call for set of screen line pointers

The Kernal determines that line 7 is the second line of a continued line, sets location 211 ($D3) to 25 to indicate the position

within the logical line that starts in line 6 (21+4), and sets 209–210 ($D1–D2) to point to the start of the logical line. Adding the contents of 211 ($D3) to 209–210 ($D1–D2) gets you to the byte you wanted. Then the Kernal stores a value of 65 in location 213 ($D5) so that you know the logical line is 66 bytes long and includes three lines, 7 through 9. This is the BASIC equivalent of using the PLOT routine, without having to load any registers for PLOT. To explore how to load registers before SYS and examine them afterward, see location 780 ($30C), the SYS register SAVE area.

SYS 60045 will clear 22 bytes of the screen to white spaces from the location where 209–210 ($D1–D2) point.

The corresponding color map location for the byte in screen RAM that you've located is available by SYSing 60082. This updates the pointer at 243–244 ($F3–F4). Add the contents of it and 211 ($D3) and insert the desired color code.

SYSing 58719 clears the screen, homes the cursor, and resets all the links in this table to indicate noncontinued lines. This is the equivalent of the CLR key.

The twenty-fourth byte of this table is used in the scrolling of the screen, while the twenty-fifth byte marks the end of the table.

If the screen is at location 7680, the first 11 link bytes are set to 158 ($9E) and the remainder set to 159 ($9F). Continuations have the high order bit turned off.

With the screen at location 4096, you would find the first 11 link bytes set to 144 ($90) and the rest set to 145 ($91). Continuations would have the high order bit off, their bytes set to 16 ($10) or 17 ($11).

If you don't know where the screen is, using this program line will tell you:

SP = PEEK(217) AND 127 : SCRN = SP * 256

The first 11 links will be the SP value, with 128 added if they are not continuations of the previous lines; the remaining 12 links will be SP plus 1. Let's approach it from the other end: If your screen is at SCRN, then SP = INT(SCRN / 256).

One way to find the color map is:

CM = 37888 : X = SCRN / 1024 : IF INT(X) <> X THEN CM = CM+512

This works because the color map is located at 37888 ($9400), unless the screen is on a half 1024-byte boundary, rather than on a 1K boundary such as 1024, 2048, 3072, 4096, and so on. If it is not on a 1K boundary, the color map moves to location 38400 ($9600).

See locations 648 ($288) and 201 ($C9). Also see Appendix E for details on screen relocation.

242 $F2

SAVE byte for screen line link table byte.

This byte is used by screen management routines to save a link byte while a related routine updates the screen line link bytes.

See location 217 ($D9).

243-244 $F3-F4   USER

(handy location)

Pointer to the current physical screen lines color map area.

This location is synchronized with location 209-210 ($D1-D2) by the COLORSYN* routine and serves as a base pointer to the appropriate color nybble for routines that store characters on the screen. These routines use location 209-210 ($D1-D2) as a screen map pointer.

You can use location 244 ($F4) as the beginning address of the color map by turning off the low-order bit of the page number:

CM = (PEEK(244) AND 254) * 256

See locations 209-210 ($D1-D2), 217 ($D9), and 201 ($C9) for a line/column/cursor pointer summation. Also see Appendix E for an explanation of screen relocation.

245-246 $F5-F6   KEYTAB

Pointer to keyboard table being used.

The keyboard tables are used to point to the proper character in ROM or RAM that a key corresponds to. Location 203 ($CB) is used as the input key number, and location 653 ($28D) is used for SHIFT patterns. The resulting ASCII character is placed in the keyboard buffer at 631 ($277) by the SCNKEY routine.

See location 631 ($277) for a summation of keyboard-related RAM locations. Refer to location 653 ($28D) for a table detailing the values of that address.

247-248 $F7-F8   RIBUF

(possible user storage)

RS-232: pointer to start of receiving buffer.

The 256-byte buffer this location points to is created when device 2 is OPENed, starting at the address in 643 ($283), minus 256. 643 ($283) is reset to that address, minus one, to protect the buffer. The address of the receiving buffer is also stored here.

CLOSEing device 2 frees this buffer and resets 643 ($283) to reflect the free memory.

A BASIC program should always open device 2 before assigning

any variables, because of the CLR that BASIC issues after device 2 is opened.

ML programs can allocate as many buffers as needed by adjusting this location, locations 643 ($283), 667 ($29B), and 668 ($29C) properly. If the MSB byte of this pointer and/or the pointer at location 249–250 ($F9–FA) is not zero, the OPEN routine assumes that the corresponding buffer(s) has (have) been already allocated. A zero in the MSB at the time CLOSE is issued causes the routine to skip the deallocation of the buffer(s).

Also see locations 249 ($F9) and 668 ($29C) and page 259 of the *VIC-20 Programmer's Reference Guide*.

249–250 $F9–FA   ROBUF
(possible user storage)

RS-232: pointer to the start of the transmitting buffer.

This address points to an additional 256-byte buffer for RS-232 data.

See the description of location 247 ($F7) and related locations 182 ($B6), 669 ($29D), and 670 ($29E).

251–254 $FB–FE   FREKZP
(user storage)

Four bytes of unused page 0 space.

255 $FF   BASZPT

BASIC temporary area for floating point to ASCII conversion.

See location 256 ($100).

Chapter 2

Memory Page 1

Memory Page 1

The 6502 processor chip in the VIC-20, as in other personal computers, reserves 256 bytes beginning at location 256 ($100) for use as a *stack* and provides instructions for saving and retrieving data from this area.

Location Range: 256–511 ($100–$1FF)
Stack Area

256–511 $100–1FF   STACK

BASIC, the Kernal, and the 6502 processor itself use this Last In First Out (LIFO) stack for storing and retrieving temporary information. Although this is only temporary information, important data is kept here.

The stack is frequently compared to a stack of plates. The number (or plate) placed on the stack goes to the bottom, and each succeeding number is put on top of it. Pulling a number off the stack gives you the last one placed on. This is called Last In First Out (LIFO). This analogy is accurate to a point, but the stack is actually upside down, built from 511 ($1FF) downward. The first item placed on the stack is put at location 511, and subsequent entries stored at consecutively lower addresses. A register in the 6502 called the *stack pointer* keeps track of the next available location in the stack and, by implication, the item next pulled from the stack.

At power-on/reset, the stack pointer is set to 255 ($FF), which is effectively 511 ($1FF) because the 6502 chip adds 256 ($100). However, NEW and CLR set the pointer to 250 ($1FA) because the JSR return address is still on the stack. This is also done after an error message is displayed by BASIC.

BASIC uses the stack for several purposes. It uses it to save register information temporarily while it calls another routine that uses the register and for GOSUB return point information. The BASIC GOSUB command uses five bytes of the stack at a time. The FOR command uses 18 bytes of the stack for every FOR outstanding, and complex expressions cause intermediate results to be saved on the stack by the formula evaluation routine at location 52638 ($CD9E).

Whenever a JSR (jump to subroutine, the ML equivalent of a BASIC GOSUB) is issued, the address of the instruction to return to, minus one, is pushed onto the stack. Since an address takes two bytes, this limits the nesting of JSRs to 127 levels if no other data was in the stack. However, there *will* be other data placed here by IRQ routines.

The area that BASIC doesn't use for stacking, the last 64 bytes at locations 256 ($100) to 319 ($13F), is used for conversion of floating point values to ASCII for printing, and by tape error recovery routines.

Here are the subareas used by BASIC as temporary work areas in the VIC-20.

256-266 $100-10A
Temporary floating point to ASCII work area for printing numbers.

This area is used by the routine at location 56797 ($DDDD), which converts the floating point accumulator to TI$ or an ASCII string.

It is also used for string-scanning purposes.

256-318 $100-13E   BAD
62 bytes of tape error log, indexes of bad data.

These indexes of erroneous data are examined during the LOAD of the second copy of the data from tape, and the data is corrected if possible.

320-511 $140-1FF   BASTACK*
Stack area used by BASIC; OUT OF MEMORY message if exceeded.

Some BASIC commands, such as FOR-NEXT loops and GOSUB, require many stack entries at a time. Thus, BASIC checks the stack size, before pushing more than a few bytes onto it, and returns the OUT OF MEMORY error message if there are less than 62 bytes left available on the stack.

Each GOSUB command causes five bytes to be pushed onto the stack, in the following order:
- Value of 141 ($8D); one byte
- Two-byte return line number
- Two-byte pointer to the return address for a total of five bytes
  Each FOR statement causes 18 bytes to be pushed onto the stack:
- Value of 129 ($81); one byte
- Pointer to the variable; two bytes
- Five-byte STEP value
- Sign; one byte
- Five-byte TO value
- Two-byte return line number
- Pointer to the loop return point; two bytes for a total of 18 bytes

See the article "FOR/NEXT/GOSUB/RETURN and the Stack," by Jim Butterfield, in the November 1981 issue of *COMPUTE!*.

Chapter 3

Memory
Pages 2 and 3

Memory Pages 2 and 3

The second and third pages of VIC-20 memory are the home of information shared between the Kernal and BASIC. The two major buffers used by BASIC are in this area, as well as the vector table that can be changed to point to the user's routines rather than default BASIC/Kernal routines. The user routine can front-end these routines to perform any extra or alternative functions. The INITMEM routine initializes this area to zeros during a power-on/reset. The routines of INITVCTRS and VECTOR then initialize the vectors stored in this page.

Location Range: 512–767 ($200–$2FF)
BASIC/Kernal Working Storage

512–600 $200–258   BUF
89-byte BASIC input buffer.

The screen editor allows a maximum of 88 characters in the input line, with the eighty-ninth byte of this buffer used to contain a 0 as an end-of-line indicator. To make your programs compatible with the Commodore 64, keep the program lines under 80 characters.

The routine FIND2 scans this buffer to find quotes, colons, and end-of-line delimiters in BASIC statements. The CRNCH routine tokenizes the BASIC statement from left to right, packing it in after the two-byte integer line number and link address. The end-of-line zero is placed after the last character of the tokenized line and the STORLN* routine stores the new line in the BASIC program area. If this line were entered in direct mode, then the MAIN routine would immediately execute the statement.

This area is also used for INPUT and GET incoming data. This is why INPUT and GET are illegal in direct mode—they require the same buffer as the statement itself. It also explains the 88-character limit on INPUT. GET places one byte in the first location, and a zero in the second to indicate the end of line/data.

See related locations 7 ($7), 8 ($8), 11 ($B), 15 ($F), and 122 ($7A).

Location Range: 601–630 ($259–276)
File Number, Device Number, and Secondary Address Tables

The three tables in this area can store up to ten one-byte entries, each representing an active Input/Output file. When an I/O file is

opened, its logical file number is placed in the table at location 601 ($259), its device number in the table at location 611 ($263), and its secondary address into the table at location 621 ($26D). The entry for each file occupies the same position in each of the three tables. If logical file number 8 is the second entry in the file number table, its device number and secondary address will also be the second entries in those tables. When a device is OPENed, its information is added as the last entry in the table. The value of location 152 ($98) is increased by one to indicate that there is one more active I/O file. CLOSEing a device decreases location 152 ($98) by one, moving higher entries down one position, thus eliminating that device's entry.

601-610 $259-262   LAT
Open logical file number table. Ten one-byte entries.

The OPEN, CLOSE, FNDFLNO, and SETFLCH routines set up, clean up, and locate information in this table.

Note that the CLALL routine simply zeros location 152 ($98) to empty these tables.

This location corresponds entry-for-entry to the tables at 611 ($263) and 621 ($26D), with location 152 ($98) usually serving as the index.

If the file number is greater than 127, a linefeed character is sent to the file following any carriage return.

See also locations 19 ($13), 152 ($98), and 184 ($B8).

See Appendix D for a device, secondary address, and status codes table.

611-620 $263-26C   FAT
Open device number table. Ten one-byte entries.

This table's entries correspond to those at locations 601 ($259) and 621 ($26D). Location 152 ($98) usually is the index.

Related locations are 73 ($49), 153 ($99), 154 ($9A), 184 ($B8), and 186 ($BA).

621-630 $26D-276   SAT
Open secondary address table. Ten one-byte entries.

This table corresponds entry for entry to the tables at 601 ($259) and 611 ($263), with 152 ($98) usually being the index.

Locations 184 ($B8) and 185 ($B9) are also related.

631-640 $277-280   KEYD
(handy location)

Ten-byte keyboard buffer.

The IRQ driven routine causes SCNKEY to fill this buffer as

it detects keystrokes with VIA 2. The Kernal routine LP2 at location
58831 ($E5CF) empties characters from the keyboard buffer and
maintains the count of characters in the buffer. Location 198 ($C6) is
updated to contain the number of characters in the keyboard buffer
to a limit of ten. Keys pressed after the limit has been reached are
ignored, unlike PETs that started over from the beginning. GET and
INPUT retrieve data from this buffer, decrementing the count of
characters and shifting the remaining characters down. This buffer is
First-In-First-Out (FIFO).

If an INPUT or GET command is issued while characters are
already in this buffer, the characters will become part of the data
stream retrieved. You can prevent this by POKEing a zero into loca-
tion 198 ($C6).

By storing the appropriate number in 198 ($C6) and placing the
desired characters in the keyboard buffer, you can program simu-
lated keyboard entry. Be sure to include the ending carriage return in
the count and buffer. See the note at 649 ($289) about exceeding ten
characters.

Unfortunately, there is no pointer to the keyboard buffer, so its
location is not changeable.

See location 153 ($99), input device number, for instructions for
reading tape as though it were the keyboard. A disk doesn't work in
the method described for tape. When disk needs to be read as the
keyboard, you can either create a tape of the information or use the
dynamic keyboard method of displaying the lines on the screen and
entering a carriage return over them.

By putting multiple carriage returns in the keyboard buffer,
clearing the screen, carefully placing lines on the screen, homing the
cursor, setting 198 ($C6) to the number of carriage returns, and end-
ing the program, the lines on the screen can be read just as though
they were entered at the keyboard. *Careful* placement of the lines on
the screen is crucial in using this feature. You must avoid BASIC
messages.

The following program can be used to create DATA statements
from the contents of memory, or modified to perform other dynamic
keyboard program functions, such as reading from disk and building
program lines to be entered. The routine is short and may be easily
appended to another program. It's handy for converting ML routines
or custom character set pixel maps into DATA statements, as well as
serving as an excellent example of the concept and a model for fur-
ther expansion. Dr. Harald Linder authored a PET version of this
routine in "Basically Useful BASIC: Automatic DATA Statements for
CBM and Atari" in *COMPUTE!*'s October 1981 issue. I've modified
his routine for the VIC-20. Because of its obscurities, I've included an
explanation of each statement following the routine.

Program 3-1. DATA Statements from Memory

```
1 INPUT "START ADDRESS";A : INPUT "END ADDRESS";E
  {SPACE}: Z = 2000
2 PRINT"{CLR}{2 DOWN}"Z"DATA"; : IF A > E THEN END
3 FOR A = A TO A+15 + (E<A+15) * (A+15-E)
4 PRINT MID$(STR$(PEEK(A)),2)","; : NEXT
5 PRINT"{LEFT} " : PRINT "A="A":E="E":Z="Z+10":GOT
  O2{HOME}";
6 POKE631,13 : POKE 632,13 : POKE 198,2 : END
```

Here's how this routine works:

Line	Function
1	Obtains the range of memory to be stored into DATA statements and sets the beginning line number to be generated at 2000. It may be tailored to your own needs.
2	The screen is cleared and the cursor positioned on the third line. The line number and the word DATA are printed there. If the starting address exceeds the ending address, the program ENDs.
3	Insures that no more than 16 numbers appear on any one DATA statement, taking into account the fact that the ending address may shorten that number. This line may be replaced with your own statement to insure that the logical line length of 88 is not exceeded.
4	PRINTs the number PEEKed from memory.
5	When the FOR loop ends, delete the last comma, and format the next direct statement that has all the necessary variables specified for reentering the program at line 2. The lines appear such as: 2000 DATA 17,5,28,27,198,182,102,55,72,91,244,7,67,212,1,187 and A=844:E= 857 :Z= 2010 :GOTO"home". The *home* is needed to safely position the READY message BASIC issues when the program ends.
6	Two carriage returns are entered into the keyboard buffer to simulate the keyboard entry of the two direct lines just PRINTed to the screen. The count of those (2) is put into the buffer counter. The program ends with the cursor in the home position. BASIC displays READY on the next line and puts the cursor on the line of our DATA statement. The first carriage return in the keyboard buffer is seen and line 2000 is stored in the program area. The cursor is placed on the next line and the second carriage return is seen and *that* line is executed, causing the program to be reentered at

line 2 with the needed variables set to the correct values. The process continues until the end address is reached.

Here's a summation of the RAM locations related to keyboard processing:

145	($91)	Keyswitch PIA: bottom keyboard row scan
197	($C5)	Matrix coordinate of last key pressed
198	($C6)	Number of characters in keyboard buffer
203	($CB)	Matrix coordinate of current key pressed
204	($CC)	Cursor blink switch
208	($D0)	Flag indicating if input from screen or keyboard
212	($D4)	Flag to indicate if within quote marks
216	($D8)	Number of outstanding inserts
245	($F5)	Pointer to keyboard table being used
→ 649	($289)	Size of keyboard buffer
650	($28A)	Keyboard repeater flags
651	($28B)	Delay before other than first repeat of key
652	($28C)	Delay before first repeat of key
653	($28D)	Current SHIFT keys pattern
654	($28E)	Previous SHIFT keys pattern
→ 655	($28F)	Pointer to the keyboard table setup routine
657	($291)	Flag to enable/disable SHIFT/Commodore switch

641-642 $281-282 MEMSTR   0/16 ($0/10)
(handy location)
Pointer to the start of user RAM memory.

At power-on/reset, the Kernal INITMEM* routine finds the first RAM location above address 1023 ($3FF) and saves that address here. After BASIC has been started, by the COLDST* routine, this location has no further use, as BASIC uses location 43 ($2B) for its start-of-memory pointer.

For an unexpanded VIC-20, the value here is 4096 ($1000). With a 3K expansion, the value is 1024 ($400), and without a 3K expansion but with 8K or more of expansion added, this location contains 4608 ($1200).

Here's a routine that you can use in direct mode or in a program by itself to cause the VIC-20 to think that it has no expansion memory available. It *de-expands* the VIC-20.

POKE 44,16:POKE 56,30:POKE 642,16:POKE 644,30:POKE
    648,30:POKE 36866,150:POKE 36869,240:POKE 4096,0:SYS 58232

You can then use any expansion RAM for your own purposes.

When you wish to *re-expand* the memory, use: SYS 64802.

See Appendix E for details of the effect that adding expansion memory has on this pointer, related pointers, and how to adjust for this in a program.

The routine MEMBOT may be used to read/set this pointer. Reference to the MEMBOT routine is on page 195 of the *VIC-20 Programmer's Reference Guide.*

643-644　　$283-284　　MEMHIGH*　　0/30 ($0/1E)
(handy location)
Pointer to the end of user RAM memory plus one.

At power-on/reset, the Kernal INITMEM* routine finds the last RAM location above 1024 ($400) and saves that address here. After BASIC has been started, by the COLDST* routine, this location is altered only by an OPEN or CLOSE of an RS-232 device. When that happens, this pointer is lowered 512 bytes to create two 256-byte buffers for Input/Output. A CLR is also issued by BASIC. *Note:* This may destroy any high-RAM-resident ML code. BASIC uses location 55 ($37) or 56 ($38) for its end-of-memory pointer. See those locations for a sample program to reserve space at the beginning and/or end of BASIC storage for ML or other uses.

The normal value for this pointer for an unexpanded VIC-20 is 7680 ($1E00).

See Appendix E for details of the effect that adding expansion memory has on this pointer, related pointers, and how to adjust for this in a program.

The routine MEMTOP may be used to read/set this pointer. Refer to page 196 of the *VIC-20 Programmer's Reference Guide.*

645　　　　　　$285　　　　　　TIMOUT
(user storage)
· **Serial: timeout enable/disable flag.**

Regardless of the description on page 205 of the *VIC-20 Programmer's Reference Guide* and the fact that SETTMO sets this location from the .A passed to it, *no reference* has been found to this location. Serial timeout is determined with the serial-clock-in line (VIA1 PA0) in the VIC-20 and this location does not disable or enable that.

The *Commodore 64 User's Guide* notes that this location is used only with the IEEE-488 expansion card.

646　　　　　　$286　　　　　　COLOR
(handy location)
Current foreground color selected by color keys.

This address is initialized to color 6 (blue) by INITSK* routine at

power-on/reset or RUN/STOP–RESTORE. When the Kernal is
about to put a character to the screen, this location's content is
stored in the corresponding color map location.

   The routine COLORSET* uses the table COLORTBL* to find the
appropriate color code to store in this location when the CTRL and
color keys are pressed.

   You can POKE this location with values from 0 to 7 to change
the color of all subsequent printed characters. Valid color codes on
the VIC are:

Black 0
White 1
Red 2
Cyan 3
Purple 4
Green 5
Blue 6
Yellow 7

   The multicolor bit (bit 3; add 8 ($8) to the color) may be set,
causing an interesting effect—useful when you've defined your own
custom character sets. The color is selected in multicolor mode by bit
pair values of 00, 01, 10, or 11.

   See locations 36879 ($900F) and 36878 ($900E) for additional
color setting locations.

   See locations 243 ($F3), 647 ($287), 217 ($D9), and 201 ($C9)
for a line/column/cursor pointer summation. Also see Appendix E
for details on screen and character pixel map relocation.

647 $287   GDCOL
Cursor: original color at this screen location.

   Every time the cursor is blinked by the IRQ routine, location
206 ($CE) is updated and the color map code for the current screen
location is stored at this address.

   See locations 243 ($F3), 646 ($286), 217 ($D9), as well as 201
($C9), for a line/column/cursor pointer summation. Also see Appen-
dix E for an explanation of screen and color map relocation. See
location 36879 ($900F) for a color code chart.

648 $288   HIBASE
                                    (handy location)
Screen map RAM page number.

   This byte is set by the power-on/reset routines to the page
number of the beginning of screen RAM. You can multiply the con-
tents of this location by 256 to find the current location of the screen
map RAM. On an unexpanded VIC-20, the screen is at 7680, and

this location contains 30 ($1E) (30*256=7680).

This location is used as a basis for the screen line link table at location 193 ($C1) and helps to derive the screen line pointer at 209–210 ($D1–D2).

See locations 243 ($F3) and 217–241 ($D9–F1). You can also refer to Appendices E and G for details on screen relocation.

649   **$289**   **XMAX**

(handy location)

Maximum number of characters in the keyboard buffer.

The normal value in this location is ten, the length of the keyboard buffer at 631 ($277). This location is compared with location 198 ($C6), which holds the number of characters in the keyboard buffer, in order to ignore key presses after the buffer is filled.

You can increase this number to 15 since locations 641–645 ($281–285) are not normally used after BASIC is cold started. It may be worth a try if you are programming the keyboard buffer.

Using this line, location 198 ($C6) could be set to 15:

POKE 649,15 : REM 15 character keyboard buffer

If you lower this value to zero, the keyboard buffer is always empty since no characters can be stored in zero bytes. This has the effect of disabling the keyboard until the value is raised above zero.

RUN/STOP–RESTORE resets this address to ten.

See location 631 ($277) for a summation of the keyboard-related RAM locations.

650   **$28A**   **RPTFLG**

(handy location)

Keyboard repeater flags.

This location is initialized to zero, which causes only the cursor, space bar, and INST/DEL keys to repeat. You can set this byte to:

POKE 650,128 : REM ALL KEYS TO REPEAT
POKE 650,64 : REM NO KEYS TO REPEAT
POKE 650,0 : REM DEFAULT KEYS TO REPEAT

See locations 651 ($28B) and 652 ($28C) for repeat timing values.

See location 631 ($277) for a summation of the keyboard-related RAM locations.

651   **$28B**   **KOUNT**

Delay before other than first repeat of key.

This is initialized to 6 once 652 ($28C), the first repeat delay counter, has been decremented to zero. Location 652 ($28C) is

reduced to zero when the same key is held down. Location 651 ($28B) is then decremented once each jiffy until it reaches zero.

The key is then placed in the keyboard buffer, location 652 ($28C) is allowed to remain zero, and this location is reinitialized to 4, allowing faster subsequent repeats. Therefore, the first repeat of a key will occur in about one-third of a second, with additional repeats occurring 15 times per second.

652 $28C   DELAY
Delay before first repeat of key.

This address's initial value of 16 is counted down every sixtieth of a second by each IRQ interrupt, as long as the same key is pressed. When zero is reached, the value in 651 ($28B) is decremented to zero on every jiffy that the key is still held down, then the key is duplicated in the keyboard buffer by the SCNKEY routine. The value 4 is then stored in location 651 ($28B), and this location is left with its value as zero so that following repeats occur rapidly.

When a different key is pressed, this location is reset to 16 ($10) by SCNKEY and the whole repeat process begins again.

653 $28D   SHFLAG
(handy location)

Current SHIFT keys pattern.

This location is used to determine which keyboard table is used for converting the key pressed into an ASCII character. Different SHIFT patterns cause the selection of the appropriate character table. Location 245 ($F5) is then set as a pointer to the current table. The values and meanings in this location are:

Table 3–1. Values in Location 653 ($28D)

Dec	Binary	Keys being pressed	Contents of this location	
0	00000000	none	60510	$EC5E
1	00000001	SHIFT	60575	$EC9F
2	00000010	Commodore key	60640	$ECE0
3	00000011	SHIFT + Commodore	60510	$EC5E
			or 60575	$EC9F
			(until pressed again)	
4	00000100	CTRL	60835	$EDA3
5	00000101	SHIFT + CTRL	60835	$EDA3
6	00000110	Commodore + CTRL	60835	$EDA3
7	00000111	SHIFT + CTRL + Comm.	60835	$EDA3

The Commodore/SHIFT key combination changes location 36869 ($9005) to cause the next character set in ROM or RAM to be used, but keyboard decoding of the characters is done using the same keyboard table.

The left and right SHIFT keys are not uniquely flagged. This value will be saved in 654 ($28E) for stabilization and to prevent the SHIFT/Commodore key combination from flipping back and forth between character sets without an additional pressing of those keys.

The following statement uses the SHIFT key, which may be locked down, as a pause key. This is handy when many screens of information are to be displayed, or you want to answer the phone during an exciting game. Simply add:

WAIT 653,1,1

to your program in the main loop. To resume the program, you only need to release the SHIFT key.

The following sample program can be an aid to understanding the CTRL codes and cursor movement key values that are placed within quotes in a program. If you create a program file within the parameters of line 110 below, control key codes will be printed within brackets on the screen. You need to load this program, then your own, to see this effect.

Program 3–2. Control Codes Displayed

```
100 REM PRINT CONTROL KEYS IN [ ] (I.E., [CLR]) FR
    OM PROGRAM FILE CREATED WITH
110 REM 'OPEN 1,1,1: CMD 1: LIST: CLOSE 1'
    {2 SPACES}OR 'OPEN 8,8,8,"NAME,S,W": CMD 8: CL
    OSE 8'
120 DIMK$(255)
130 READ K: IF K=0THEN150
140 READ K$(K):GOTO130
150 PRINT"{CLR}{2 DOWN}FILE NAME":INPUTN$
160 PRINT"{2 DOWN}{RVS}T{OFF}APE OR {RVS}D{OFF}ISK
     IN?"
165 GETD$:IFD$<>"D"ANDD$<>"T"THEN165
170 PRINT"{2 DOWN}{RVS}P{OFF}RTR OR {RVS}D{OFF}ISK
     OUT?"
172 GETO$:IFO$<>"P"ANDO$<>"D"THEN172
175 IFO$="D"THENOPEN4,8,4,N$+".LXS,S,W"
176 IFO$="P"THENOPEN4,4
180 IFD$="T"THENOPEN5,1,0,N$
190 IFD$="D" THEN OPEN15,8,15: OPEN5,8,5,N$+",S,R"
200 IF D$="D"THEN INPUT#15,E,E1$,E2,E3: IF E<>0THE
    N PRINT"{RVS}"E;E1$: CLOSE15: CLOSE4: END
210 GET#5,X$:ST%=ST:IFX$=CHR$(34)THENQ%=ABS(Q%-1)
220 IFX$=CHR$(13)THENQ%=0
```

```
230 IFQ%THEN IFLEN(K$(ASC(X$)))<>0THENX$=K$(ASC(X$
    ))
240 PRINT#4,X$;:IFST%=64THEN CLOSE 4: CLOSE 5: END
250 GOTO210
260 DATA5,"[WHT]",8,"[DISABLE2]",9,"[ENABLE2]",10,
    "[LF]",14,"[LOWERCASE]",17,"[CRSRDN]"
270 DATA 18,"[RVSON]",19,"[HOME]",20,"[DEL]",28,"[
    RED]",29,"[CRSRRT]",30,"[GRN]"
280 DATA 31,"[BLU]",131,"[LOAD/RUN]",142,"[UPPERCA
    SE]",144,"[BLK]",145,"[CRSRUP]"
290 DATA 146,"[RVSOFF]",147,"[CLR]",148,"[INST]",1
    56,"[PUR]"
300 DATA 157,"[CRSRLFT]",158,"[YEL]",159,"[CYN]",0
```

See location 36869 ($9005) for programmed SHIFT of character sets or use the following PRINT statements:

PRINT CHR$(14) : REM lowercase/uppercase (text set)
PRINT CHR$(142) : REM uppercase/graphics (graphics set)

Also see location 657 ($291) for a flag to enable or disable combined SHIFT and Commodore keys.

654 $28E   LSTSHF
Previous SHIFT key pattern.

This location is used in combination with location 653 ($28D) to *debounce* the special SHIFT keys. This will keep the SHIFT/Commodore key combination from changing character sets back and forth during a single pressing of both keys. The values in this location are saved from location 653 ($28D). See location 653 ($28D) for the meaning of values in this location. You can also refer to location 631 ($277) for a summation of keyboard-related RAM locations.

655-656 $28F-290   KEYLOG
Pointer to the default keyboard table setup routine.

This location is used as an indirect jump pointer by the SCNKEY routine to the routine that determines which keyboard decoding table is used, based on the SHIFT key pattern in 653 ($28D). By changing this pointer after power-on/reset, you can intercept the provided routine or replace it altogether. The default routine is at location 60380 ($EBDC), which uses keyboard table vectors at 60486 ($EC46) to access the tables starting at location 60510 ($EC5E) NORMKEYS*. You will probably want to model your routine after the default routine, or front-end it.

The default routine in the Kernal stores the address of the table currently in use at location 245 ($F5).

See location 631 ($277) for a summation of keyboard-related RAM locations.

657 $291   MODE
(handy location)
Flag to enable or disable combined SHIFT and Commodore keys.

Values in this location are 0 for enabled, 128 for disabled.

Although this location's value is initially set to 0, a value of 128 ($80) here will disable the SHIFT and Commodore key combination from switching to the alternate character set. The SHIFT, Commodore, and CTRL keys will work normally when pressed for any other purpose.

You can use these program lines to disable or enable character set switching. Both a POKE statement and a PRINT statement have been included:

```
POKE 657,128 : REM disable character set switching
PRINT CHR$(8) : REM disable character set switching
POKE 657,0 : REM enable character set switching
PRINT CHR$(9) : REM enable character set switching
```

See location 36869 ($9005) for programmed SHIFT of character sets or location 653 ($28D), the current SHIFT pattern, for PRINT statements.

658 $292   AUTODN
Screen scroll-down enabled flag.

This byte's value is set to 0 to enable the scroll-down function when the computer is turned on. Any other value in this location disables the scroll down.

This location flags whether the bottom logical line is dropped off the screen to make room for another physical line added to the current logical line.

This function is temporarily disabled while characters are in the keyboard queue.

See location 201 ($C9) for a summation of keyboard RAM locations.

659 $293   M51CTR
(possible user storage)
RS-232: pseudo-6551 control register.

This location specifies the RS-232 baud rate (the transmit/ receive speed), the word length in bits, and the number of stop bits. The VIC-20 emulates a 6551 UART chip with software.

When device number two is opened, you can specify up to four characters in the filename which are copied to locations 659 ($293) through 662 ($296) and correspond to the RS-232 control register, command register, and nonstandard bit-timing values. Only the first

of the four characters is required, and typically only the first two are specified.

If the command 6551 command register is not specified, the defaults are: disabled parity, full duplex, and three-line handshaking.

To set this location:

OPEN n,2,0,CHR\$(w)+CHR\$(x)

where w is the value for this location, x is the value for location 660 (\$294), and n is the file number. Remember that a file number greater than 127 causes a linefeed to follow any carriage returns sent.

The meaning of the bits in this location is:

Table 3-2. Location 659 Bit Values

Bit	Use	Meaning
7	stop bits	0 = 1 stop bit, 1 = 2 stop bits
6-5	word length	00 = 8, 01 = 7, 10 = 6, 11 = 5
4	unused	
3-0	baud rate	

0000 = user*	0001 = 50	0010 = 75
0011 = 110	0100 = 134.5	0101 = 150
0110 = 300	0111 = 600	1000 = 1200
1001 = 1800	1010 = 2400	1011 = 3600*
1100 = 4800*	1101 = 7200*	1110 = 9600*
1111 = 19,200*		

* indicates NI (not implemented). If the baud rate bits are 0000, the user timing at location 661 (\$295) was designed to be used, but this is not implemented.

Here are the positions and values for each of the eight bits that make up any byte.

Table 3-3. Bit Positions and Values

Table of bit positions / decimal values / hex values

BIT	7	6	5	4	3	2	1	0
DEC	128	64	32	16	8	4	2	1
HEX	\$80	\$40	\$20	\$10	\$08	\$04	\$02	\$01

For example, the statement OPEN 2,2,0,CHR\$(6+32)+CHR\$(32+16) sets this location to 38 (6+32). In binary, this looks like 0010 0110. Referring to Table 3-2, you can see that this sets the baud to 300 (6) and the word length to seven bits (32), with one stop bit.

When a word length less than eight is chosen, the remaining bits of the byte are set to zeros. ML should be used for rates over 300 baud.

See page 251 of the *VIC-20 Programmer's Reference Guide*, locations 663 ($297), 660 ($294), and 37136 ($9110).

660	**$294**	**M5ICDR**
		(possible user storage)

RS-232: pseudo-6551 command register.

This location specifies the RS-232 parity, for data checking; duplex mode, to determine if talking and listening occur simultaneously; and the handshaking protocol used, to insure that the same protocol is used on the other end of the transmission. (If it's not the same protocol, it's like one person extending his left hand and the other person extending his right to shake hands.)

The VIC-20 emulates a 6551 UART chip with software.

See location 659 ($293) for a description of how *this* location is set with the OPEN statement. Setting with the OPEN statement is optional.

The bits in this location represent:

Table 3-4. Location 660 Bit Values

Bit	Use	Meaning		
7–5	parity	xx0 = disabled	001 = odd	011 = even
		101 = mark	111 = space	
4	duplex	0 = full	1 = half	
3–1	unused			
0	handshaking	0 = 3 line	1 = x line	

There seems to be a problem with x line handshaking in the VIC-20. It appears that the code at 62738 ($F512) and 61428 ($EFF4) is checking the wrong VIA port. If not otherwise set, the zero in this location will default RS-232 to no parity, full duplex, and three-line handshaking.

Using the example OPEN statement found in location 659 (OPEN 2,2,0,CHR$(6+32)+CHR$(32+16)), you can see how to set location 660. Note that in our example, location 660 is being set to 48 (32+16), which is expressed as 0011 0000 in binary. This sets this register to odd parity (32) and half duplex (16). Refer to Table 3-3 for bit positions and values to see how the bit values translate into these register settings.

See page 251 of the *VIC-20 Programmer's Reference Guide*, as well as locations 663 ($297) and 37136 ($9110).

661–662   **$295–296**   **M51AJB**

(user storage)

RS-232: nonstandard bit timing specification.

This location is used to store the user-desired baud rate.

See location 659 ($293), the RS-232 control register, for a description of how to set this location. Unfortunately, this is not implemented in the VIC-20, since RS-232 routines never reference this location.

663   **$297**   **RSSTAT**

(possible user storage)

RS-232: status register.

This status register is erroneously zeroed by ST if the current device in location 186 ($BA) is device 2.

The READST routine at location 65111 ($FE57) that is called by the vector at 65463 ($FFB7) zeros this byte and loses its contents.

Use RS=PEEK(663):POKE 663,0 to obtain the RS-232 status.

The Commodore 64 READST Kernal routine returns the status in .A as well as zeroing the byte, but this was overlooked in the VIC-20. The .A returned is zeroed in the VIC.

The user is responsible for checking and taking appropriate action for status bits 1, 2, 4, and 7. For bit 7, you'll want to stop sending and issue a GET#2 to see what the other end is trying to say. If bit 2 is being set, this is an indication that you're not issuing a GET#2 fast enough to clear the buffer. Even with BASIC, you should be able to keep up at 300 baud. Bit 2 or 1 should cause you to send again the last PRINT#2 byte.

Here's a description of this location's bit values and their meanings:

Table 3–5. Location 663 Bit Values

Bit	Decimal	Hex	Meaning
7	128	80	BREAK detected
6	64	40	DATASET READY message missing modem is not free for next task
5	32	20	unused
4	16	10	CLEAR TO SEND message missing (see below) modem is not ready for data to be sent to it
3	8	08	unused
2	4	04	receive buffer overrun
1	2	02	framing error
0	1	01	parity error

Because of the coding problem mentioned at 660 ($294), the *clear to send missing* bit will never be set.

664 $298   BITNUM
(possible user storage)

RS-232: number of bits to be sent/received.

This location is used to indicate how many zeros must be added to the data character to pad its length to the word length specified in location 659 ($293), the RS-232 control register.

Also see locations 180 ($B4) and 168 ($A8).

665–666 $299–29A   BAUDOF
(possible user storage)

RS-232: system clock divided by baud rate. Result is expressed in microseconds.

This location contains the amount of time needed to send one bit of information.

- 666 ($29A) is computed by:
INT((((CLOCK/baud rate)/2)−100)/256
- 665 ($299) is computed by:
((CLOCK/baud rate)/2)−100−(PEEK(662)*256)
where CLOCK=1,022,730 for NTSC (USA) or
1,108,224 for PAL (European)

The OPENRS* routine performs these calculations when the RS-232 device is OPENed. The computations are speeded by the Kernal using the provided BAUDTBL* table.

The resulting figures are copied to locations 37140–37141 ($9114–9115) and 37144–37145 ($9118–9119), which are VIA 1's timers, when needed.

667 $29B   RIDBE
(possible user storage)

RS-232: dynamic index to the end of the receive buffer.

This pointer references a 256-byte buffer and is used to place data in that buffer. The receive buffer is a *wraparound* buffer. At any time, the starting and ending locations can be anywhere within the 256-byte buffer.

If this location and 668 ($29C) are equal, then the buffer is empty; when this location's value is one greater than 668 ($29C), the buffer is full.

668   **$29C**   **RIDBS**
                                        *(possible user
                                        storage)*

RS-232: dynamic index to the start of the receive buffer.

This pointer is used to remove data from the receive buffer.

If location 667 ($29B) and this location contain the same value, the buffer is empty.

This byte is added to the pointer at location 247 ($F7) when storing data in the transmit buffer.

669   **$29D**   **RODBS**
                                        *(possible user
                                        storage)*

RS-232: dynamic index to the start of the transmit buffer.

This pointer is used to put data into the buffer and to detect an empty or overflow of the transmit buffer.

If this location and 668 ($29C) are equal, the buffer is empty; when this location's value is one greater than 668 ($29C), the buffer is full.

670   **$29E**   **RODBE**
                                        *(possible user
                                        storage)*

RS-232: dynamic index to the end of the transmit buffer.

This pointer is used to remove data from the buffer.

If 669 ($29D) and this location contain the same number, the buffer is empty.

This index is added to the pointer in location 249 ($F9) when storing data in the transmit buffer.

671-672   **$29F-2A0**   **IRQTMP**
                                                *(possible user
                                                storage)*

Temporary SAVE area for the normal IRQ vector during tape I/O.

The normal vector, held in location 788-789 ($314-315) for the IRQ routine at 60095 ($EABF), is stored here during tape reads and writes by the TAPE routine. It's restored by the TNIF routine. The table at 65009 ($FDF1) contains the three IRQ vectors for tape. The vectors are at locations 63886 ($F98E), 64523 ($FC0B), and 64680 ($FCA8).

The tape I/O IRQs skip the update of the clock, the STOP key test, and other IRQ duties during the actual moving of the tape. The tape routines call their own routine that tests for a pressed STOP key.

If you've changed the IRQ vector at 788–789 ($314–315), your vector will be restored from here after the tape I/O is completed.

The TSTOP routine, which tests for the STOP key, puts a zero in location 672 ($2A0) if the STOP key *was* pressed.

673-767 $2A1-2FF   USRVCTRS*
(user storage)

User indirect vectors or other storage area.

Ninety-four bytes of memory are available in this area to the user for 47 user-program indirect link addresses or for any other purpose. This is an excellent area to store a short ML routine without having to alter BASIC's pointers.

Note that locations 673–678 ($2A1–2A6) *are* used in the Commodore 64, so refrain from using these if you're writing a program meant for both the VIC-20 and the Commodore 64.

Location Range: 768-778 ($300-$30A)
BASIC Indirect Vectors

768-778 $300-$30A   BVECTORS
(handy location)

Table of indirect BASIC vectors.

In these locations, BASIC provides a table of the vectors that direct processing to the appropriate routine. These vectors are used by BASIC at the start of a routine. The instruction at the routine's entry point is a jump from the address contained in the corresponding vector location. This causes a branch back to the instruction following the jump. For example, the entry point for the BASIC error message handler is at location 50231 ($C437). The instruction at that address is a JMP to $0300. The vector at $0300 points to location 50234 ($C43A), which is the next instruction after the JMP. You can change these vectors to replace or front-end the BASIC routines.

Here are the individual vectors, their locations, labels, and descriptions:

768-769 $300-301   IERROR
Vector to the routine to print a BASIC error message from a table.

This vector points to the routine ERROR at location 50234 ($C43A).

770-771 $302-303   IMAIN
Vector to the BASIC main routine. Execute or store statement.

The routine MAIN at location 50307 ($C483) is pointed to by this vector.

772-773 $304-305   ICRNCH
Vector to the BASIC tokenization routine.

This vector points to the routine CRNCH at location 50556 ($C57C).

774-775 $306-307   IQPLOP
Vector to the BASIC routine that expands and prints tokens.

The vector points to the QPLOP routine at address 50970 ($C71A).

776-777 $308-309   IGONE
Vector to the BASIC routine that executes the next BASIC token.

This vector points to the routine GONE at location 51172 ($C7E4).

778-779 $30A-30B   IEVAL
Vector to the BASIC routine that evaluates a variable

Vector to the routine EVAL at 52870 ($CE86).

Location Range: 780-783 ($30C-$30F)
Register Save Area

The BASIC SYS command uses this area to save and load the 6502 registers between SYS statements. For example, SYS 60074 loads the values stored in this area into the appropriate registers and then performs a jump to the target location. When the ML instructions issue an RTS (return), the routine that processes the SYS statement stores the returned registers here, and BASIC continues with its next statement.

This feature allows you to set up the necessary registers prior to SYSing to a Kernal, BASIC, or ML user routine. It also lets you examine or save the resulting registers and pass them onto another routine via subsequent SYS commands.

Another example will clarify this. If you wanted to place the cursor and a red dollar sign on the seventh physical line in the fourth column, you could use the following routine:

- Position the cursor and obtain screen location
  POKE 783,0 Clear the processor flag register (.P)
  POKE 781,6 Select the 7th line in .X register
  POKE 782,3 Select the 4th column in .Y register
  SYS 65520 Call the PLOT Kernal routine vector. Kernal moved
  cursor to line 7, column 4, and updated 209-210 and 211
  (line-col.)
- Put the dollar sign in the screen location
  LINE=PEEK (209)+PEEK(210)*256 Start of line address

ADDR=LINE+PEEK(211) Add column to start of line
POKE ADDR,36 Place a dollar sign on the screen
● Color the dollar sign red
SYS 60082 Call the Kernal to determine color address. Kernal
updated locations 243–244 to color address
COLOUR=PEEK (243)+PEEK(244)*256 Start of line color.
COLOUR is speeled with a u to avoid the OR command
POKE COLOUR+PEEK(211),2 Dollar sign's color address

    The routine can fit in two BASIC program lines and can be generalized so that you can use it as a GOSUB subroutine. Using Kernal routines to do your work for you is both faster and easier than doing it yourself, once you know how to pass and retrieve the needed parameters in the 6502 registers. Suddenly all the Kernal routines are now available to you in a BASIC program.
    These are the individual registers in this area:

780	**$30C**	**SAREG**
		(handy location)

6502 .A register

781	**$30D**	**SXREG**
		(handy location)

6502 .X register

782	**$30E**	**SYREG**
		(handy location)

6502 .Y register

783	**$30F**	**SPREG**
		(handy location)

6502 .P processor status register

    Take a look at Table 3-6 for details on interpreting the .P flags.

Table 3–6. .P Register Flags

Bit Num	Dec	Hex	FLAG Name	PLP Sets	Bit Sets	CMP Sets	To CLEAR/SET	BRANCH ON 0/1
					For ML Programmers			
7	128	$80	N Negative	X	X	X	---/---	BPL/BMI
6	64	$40	V oVerflow	X	X	-	CLV/---	BVC/BVS
5	32	$20	----------	-	-	-	---/---	---/---
4	16	$10	B Break	X	-	-	---/---	---/---
3	8	$08	D Decimal	X	-	-	CLD/SED	---/---
2	4	$04	I Interrupt	X	-	-	CLI/SEI	---/---
1	2	$02	Z Zero	X	X	X	---/---	BNE/BEQ
0	1	$01	C Carry	X	-	X	CLC/SEC	BCC/BCS

784-787   **$310-313**   **PG3FREE***
                                          (user storage)

Four bytes of unused page 3 space.

On the Commodore 64, locations 785–786 are used for the USR jump vector described at location 1–2 ($1–2). Location 784 contains the JMP opcode. You should remember this when you're writing programs for both machines.

Location Range: 788-819 ($314-$333)
Kernal Indirect Vectors

788-819   **$314-$333**   **KVECTORS**
                                          (handy location)

Table of 16 Kernal indirect vectors.

This location range contains a 32-byte table of the vectors that direct the processing to the appropriate Kernal routine. At power-on/reset, or when pressing the RUN/STOP–RESTORE keys, the RESTOR routine copies the ROM copy of these vectors at VEC-TORS* into this location range. The VECTOR routine can be called to read or load these vectors, allowing you to front-end or replace vectored Kernal routines. See page 209 of the *VIC-20 Programmer's Reference Guide* for the description of how to use the VECTOR routine. Many of the routines that these vectors point to are also described in the reference guide, beginning on page 184.

Location 195–196 ($C3–C4) is used as a base address during the process of copying the ROM based vectors to this location range.

The 16 indirect vectors are:

788-789   **$314-315**   **CINV**
                                          (handy location)

Vector to the IRQ interrupt routine at 60095 ($EABF).

This location is where you would put the address of your own ML IRQ routine or front-end to be executed every jiffy. You must end your front-end routine with a JMP to location 60095 ($EABF).

The IRQROUT* routine determines if it was entered for a RUN/STOP–RESTORE key press or an elapsed jiffy and jumps off this vector.

You can POKE 37166,127 to allow this location to be changed in BASIC without being interrupted, or POKE 37166,192 to reenable the IRQ interrupts.

The routine that first receives the IRQ interrupt, prior to the routine that is vectored here, saves .A, .X, and .Y registers on the stack. To restore these registers at the end of your routine, PLA the registers and issue an RTI instruction, or proceed with the normal IRQ routines.

To disable the timer updating and STOP-key test by the IRQ routine, POKE 788,194. This will skip the timer update and keyboard scan routines by entering IRQ after the JSR for them. To reenable these functions, POKE 788,181. See the vector at 808 ($328) to disable *only* the STOP key.

Location 671–672 ($29F–2A0) is used to save this vector during tape processing and to restore this vector afterward.

The following program demonstrates the use of this location to "wedge" in your own routines in front of the normal target of a vector.

Program 3-3. IRQ Wedge

```
10 REM *** SAMPLE IRQ WEDGE THAT TURNS OFF QUOTE A
   ND INSERT MODE BY DETECTING F1 KEY ***
20 REM +++ ADAPTED FROM AN IDEA BY SHELDON LEEMON
   {SPACE}FOR THE C64. +++
30 T=256*PEEK(56)+PEEK(55)-25 : HI%=T/256 : LO=T-H
   I%*256 : REM FIND TOP OF RAM-25 (T)
40 FOR I=T TO T+24 : READ D : POKEI,D : NEXT : REM
    STORE INTERRUPT WEDGE IN HIGH RAM
50 POKE 56,HI% : POKE 55,LO : REM LOWER TOP OF RAM
    POINTER TO HIDE THE WEDGE ROUTINE
60 POKE 814,PEEK(788) : POKE 815,PEEK(789) : REM S
   AVE OLD IRQ VECTOR FOR INDIRECT JUMP
70 POKE 37166,127 : REM DISABLE IRQ INTERRUPT WHIL
   E CHANGING ITS VECTOR
80 POKE 788,LO : POKE 789,HI% : REM POINT THE IRQ
   {SPACE}VECTOR TO THE WEDGE
90 POKE 37166,192 : NEW : REM ENABLE IRQ INTERRUPT
    AND DISCARD THIS PROGRAM
100 DATA 165,215,201,133,208,16,162
110 DATA 0,134,212,134,199,134
120 DATA 216,232,134,198,169,20
130 DATA 141,119,2,108,46,3
140 ### ASSEMBLER CODE IN WEDGE ###
150 LDA $D7 ;LAST KEY PRESSED
160 CMP #$85 ;F1 KEYPRINT
170 BNE OUT{2 SPACES};EXIT IF NOT
180 LDX $#00 ;TURN OFF
190 STX $D4{2 SPACES};{2 SPACES}QUOTE MODE
200 STX $C7{2 SPACES};{2 SPACES}REVERSE MODE
210 STX $D8{2 SPACES};{2 SPACES}INSERT COUNT
220 INX{6 SPACES};PUT A 1 IN
230 STX $C6{2 SPACES};{2 SPACES}KBD BUFF COUNT
240 LDA $#14 ;AND A DELETE FOR
250 STA $0277 ;THE F1 KEY
260 OUT JMP ($032E) ;BACK TO THE NORMAL ROUTINE
```

790-791 $316-317   CBINV
Vector to the BREAK interrupt routine at 65234 ($FED2).

When the RUN/STOP and RESTORE keys are pressed simultaneously, or an ML BRK instruction ($00) is executed, this vector points to the address of the routine at location 65234 ($FED2). See the BREAK* routine for details of the processing performed.

792-793 $318-319   NMINV
Vector to the NMI interrupt routine at 65197 ($FEAD).

The NMI (Non-Maskable Interrupt) can be caused by the RUN/STOP–RESTORE keys and VIA timer interrupts. Multiple NMI interrupts and IRQ interrupts can occur while the first NMI is being processed. This entry point skips the preceding SEI instruction that disables IRQ interrupts.

794-795 $31A-31B   IOPEN
Vector to the open logical file routine OPEN at 62474 ($F40A).

796-797 $31C-31D   ICLOSE
Vector to the close logical file routine CLOSE at 62282 ($F34A).

798-799 $31E-31F   ICHKIN
Vector to the open input channel routine CHKIN at 62151 ($F2C7).

800-801 $320-321   ICKOUT
Vector to the open output channel routine CHKOUT at 62217 ($F309).

802-803 $322-323   ICLRCH
Vector to the reset all channels routine CLRCHN at 62451 ($F3F3).

804-805 $324-325   IBASIN
Vector to the input from device routine CHRIN at 61966 ($F20E).

806-807 $326-327   IBSOUT
Vector to the output to device routine CHROUT at 62074 ($F27A).

808-809 $328-329   ISTOP
(handy location)

Vector to the test STOP key routine STOP at 63344 ($F770).

This vector points to the address of the routine that tests the STOP key. The STOP key can be disabled by a POKE 808,100. This does not disable the RUN/STOP–RESTORE combination, however. To reset the STOP key test, POKE 808,112.

810-811　　　　**$32A-32B**　　　　**IGETIN**
Vector to the get from keyboard routine GETIN at 61941 ($F1F5).

812-813　　　　**$32C-32D**　　　　**ICLALL**
Vector to the abort all files routine CLALL at 62447 ($F3EF).

814-815　　　　**$32E-32F**　　　　**USRCMD**
(user storage)

This location seems to be a holdover from earlier PET computers, when the built-in machine language monitor would JMP through this vector when it encountered a command it did not understand. A user vector can be placed here instead.

This vector is initialized to serve as a vector to the BREAK interrupt routine BREAK at location 65234 ($FED2).

816-817　　　　**$330-331**　　　　**ILOAD**
Vector to the load from device routine LOAD at 62793 ($F549).

818-819　　　　**$332-333**　　　　**ISAVE**
Vector to the Kernal save to device routine SAVE at 63109 ($F685)

820-827　　　　**$334-33B**　　　　**USRCMDS**
(user storage)
Four user vectors or eight bytes of other data may be stored here.

828-1019　　　　**$33C-3FB**　　　　**TBUFFR**
(handy location)
Tape buffer area, 192 bytes, for headers and BASIC file data.

This area of memory is used to read and write tape headers regardless of the format of the tape data, and to buffer a BASIC program's data for INPUT#, GET#, and PRINT# commands when using the tape device. Each use of this area will be explained below.

Also, this area has traditionally been a favorite place to store short ML routines. Of course, these routines are erased whenever tape I/O is done. Serial devices do not use this area.

When the Kernal LOAD, VERIFY, and SAVE routines are called from ML or BASIC SYS statements, this area is used for the tape header but *not* for the data going to or coming from the tape. When calling the Kernal LOAD or VERIFY routine, a pointer to the beginning of the area used for LOADing is stored in location 172–173 ($AC–AD). When SAVE is called, an additional pointer to the end, plus one, of data to be saved is stored in location 174–175 ($AE–AF) by the Kernal. These pointers passed to the Kernal may point to any convenient location, except when saving data from above location 32768 ($8000) to tape. See location 172 ($AC) for an explanation of

this limit. The contents of RAM are then saved to the tape in one long continuous block.

BASIC uses the same Kernal routines in the same manner for SAVE, LOAD, and VERIFY of a BASIC program. BASIC's pointer at location 43–44 ($2B–2C) is used as the starting pointer, except in the case of a tape header type 3 LOAD or a LOAD with a secondary address of 1 after the device number. In either case, the data is stored in memory at the same location that it was saved from. SAVE additionally uses the BASIC pointer in location 45–46 ($2D–2E) to point to the end-plus-one of the BASIC program.

When a BASIC program issues INPUT#, GET#, or PRINT#, BASIC puts the data into 191-byte blocks, preceded by a tape header I.D. of 2. This is transparent to the user except for the periodic starting and stopping of the tape. BASIC calls the Kernal routines that input or output a byte of data (CHRIN or CHROUT) whenever it determines that the current block has been completed. INPUT# and GET# data is transferred to the BASIC buffer at 512 ($200) for processing.

There are several locations you can refer to for more information regarding the tape buffer. They are:

Locations 19 ($13), 158 ($9E), 159 ($9F), 160 ($A0), 166 ($A6), 167 ($A7), 170 ($AA), 172 ($AC), 174 ($AE), 178 ($B2), 183 ($B7), 185 ($B9), 187 ($BB), 192 ($C0), 193 ($C1), 195 ($C3), 256 ($100), 671 ($29F), and Appendix D, which includes a device, secondary address, and status codes chart.

Tape Buffer for Header Data

When the tape file is opened, the Kernal checks that the tape buffer area is not located below 512 ($200) by examining the pointer at location 178–179 ($B2–B3). If the address in the pointer is below that point, an ILLEGAL DEVICE NUMBER message is displayed.

828	**$33C**	**TPHDRID**[*]
		(handy location)

Tape header identifier byte (1–5).

This I.D. is the first byte of the tape header, except for I.D. 2, which is in the first byte of every record. This accounts for the tape buffer consisting of 192 bytes, even though there are only 191 bytes of user data.

Each I.D. byte is detailed below:

● 1 *Relocatable*: When the file is saved, if the secondary address specified was 0 or any even number, this I.D. is used to indicate that the program is to be loaded where the pointer at 43–44 ($2B–2C) indicates, *unless* the LOAD has a secondary address of 1 after the device number. RAM saved with a monitor program such as VICMON

will have this tape header I.D., use a secondary address of 1 to LOAD the data to its original location, or point 43–44 ($2B–2C) appropriately. See the explanation at location 829–830 ($33D–33E) for how to read and modify the tape header before it's acted upon by the LOAD routine.

   • **2** *BASIC Program Data Block*: Followed by 191 bytes of data.

   • **3** *Nonrelocatable*: When saved with a secondary address of any odd number, this I.D. indicates that the program always loads to the address contained in locations 829–830 ($33D–33E). Any start-of-load pointer passed to the Kernal is ignored. This starting pointer is obtained from location 43–44 ($2B–2C) by BASIC when the SAVE was used.

   • **4** *BASIC Program Data Header*: Indicates that data written by a BASIC program follows in 192-byte blocks, each with an I.D. 2 as the first byte.

   • **5** *Logical End of Tape*: The Kernal will stop searching the tape when this header is encountered. You may, however, have additional files beyond this point.

829-830 $33D-33E   TPHBGN*
(handy location)

Starting address for tape LOAD.

   One technique that can be used for loading data from tape to any desired address is to change the pointer at 43–44 ($2B–2C) before issuing the LOAD of a relocatable BASIC program. See the append example at location 43–44 ($2B–2C). An alternate method, and a method of overriding I.D. 3, is to read the tape header in, modify it to your liking, and then cause the LOAD routine to use it to perform the function as though you had entered LOAD "",1,1.

   First you need to call the routine FAH, which reads in the next tape header, by SYSing 63407. The Kernal will request that the PLAY button on the recorder be pressed if necessary, and the name of the file found will display. The .X register now contains the tape header I.D., and the tape header now sits in memory at locations 828–1019 ($33C–3FD). You can change the pointers at 829–830 ($33D–33E) and 831–832 ($33F–340) to indicate the starting and ending address that you want for the LOAD. SYSing 62980 finishes the LOAD "",1,1, placing the data where you indicated.

   Also see location 172–173 ($AC–AD) for related information.

831-832 $33F-340   TPHEND*
(handy location)

Ending address, plus one, of tape LOAD.

   See the explanation at location 829–830 ($33D–33E) for details

of how to read and modify the tape header before it is acted upon by the LOAD routine.

See location 174–175 ($AE–AF).

833–1019   **$341–3FD**   **TPHNAME***

(handy location)

Filename of tape data.

Padded with blanks, these 187 bytes hold the filename that is specified with SAVE, OPEN, or SETNAM.

See the discussions at locations 183 ($B7) and 187 ($BB).

You can have tape files with ML programs saved in them. See the article "Saving Machine Language Programs on PET Tape Headers," by Louis F. Sander, in the July 1981 issue of *COMPUTE!*.

Control codes—such as switch to lowercase (CHR$(14))—may be embedded in the filename and will be acted upon when the filename is displayed on the screen.

The contents of this location may be assigned to a string variable in a BASIC program. This is useful when you have opened a tape file with OPEN1,1 when you don't know what the tape filename is. Insure that NA$="""" is the first variable defined in your BASIC program. OPEN the file, then

N=PEEK(45)+PEEK(46)*256

This will set N to the location of the string descriptor for the variable NA$ in the variable pool. POKE N+2,187 to set the string descriptor length byte to the length of the padded-out tape filename.

POKE N+3,65:POKE N+4,3

will set the string descriptor to use the stored filename whenever NA$ is referred to. The filename will be overlaid when further tape I/O is done. NM$=NA$+"""" will save this name for future use, if issued before further tape I/O.

Take a look at Figure 3-1 for a moment. This illustrates a typical tape format for program SAVEs, not BASIC data.

Figure 3–1. Tape Format
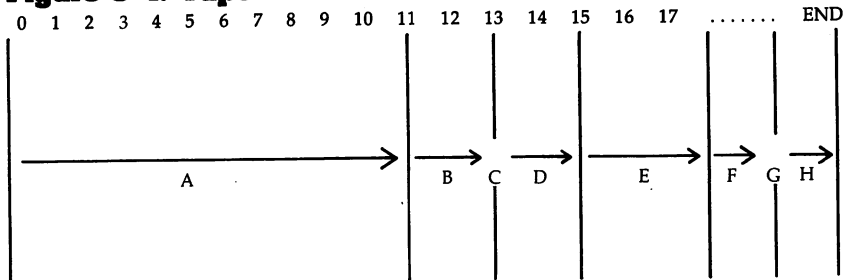
Details of each section of the tape format are:
- **A** Ten seconds of leader dipoles
- **B** Header Block #1. First nine bytes are block countdown characters (987654321) with high order bit on. Final byte is checksum.
- **C** Interblock Gap. Long bit, then 80 cycles of leader dipoles.
- **D** Header Block #2. First nine bytes are block countdown characters with high order bit off. Final byte is checksum.
- **E** Three seconds of leader dipoles between second block of header and first block of program.
- **F** Program Block #1. First nine bytes are block countdown characters with high order bit on. Final byte is checksum.
- **G** Interblock Gap. Long bit, then 80 cycles of leader dipoles.
- **H** Program Block #2. First nine bytes are block countdown characters with high order bit off. Final byte is checksum.

Tape Buffer for BASIC Program Data

828 $33C

(possible user storage)

Tape BASIC program data block identifier (2).

This I.D. is the first byte of every block and accounts for the fact that each 192-byte tape buffer has only 191 bytes of user data.
See location 166 ($A6) for more information.

829-1019 $33D-$3FB   TPBLOCK*

(possible user storage)

Tape block of 191 user data bytes from a BASIC program.

This area is a block buffer for PRINT#, INPUT#, and GET#.
Location 178 ($B2) points to this buffer, and 166 $A6 contains the number of characters contained in it.

1020-1023 $3FC-3FF

(user storage)

Four bytes of unused area.

Chapter 4

Built-in and Expansion RAM Character ROM

Built-in and Expansion RAM Character ROM

Location Range: 1024-4095 ($400-$FFF)
3K Expansion RAM

1024-4095 $400-FFF   RAMBLKO*
3072 bytes of expansion RAM area.

A VIC-20 with 3K of expansion RAM causes BASIC to start the user's BASIC program at the start of this area. Location 43-44 ($2B-2C) point to the start of this area and contains 00/04. An unexpanded VIC-20 starts BASIC at location 4096 ($1000), and an 8K+ expanded VIC starts BASIC at location 4608 ($1200). Once an 8K expansion is added to the VIC-20, this area, if filled with a 3K expansion RAM board, is not seen by BASIC, and you may use it to store your own information or for ML routines. The VIC chip cannot see this area, so it cannot be used for screen or character memory. Location 55-56 ($37-38) points to the end of continuous RAM.

Take a look at Figure 4-1, which illustrates the memory locations with no expansion, with 3K expansion, and with more than 8K expansion.

See Appendix B for details of the internal storage of BASIC and its programs, and Appendix E for explanations of the relocatable VIC-20 memory areas, user relocation of these areas, and memory configuration independent programming techniques.

You can also refer to "Supercharge Your VIC," by Dan Rubis, in the April 1983 issue of *Microcomputing* for details on assembling your own 3K expansion.

Location Range: 4096-8191 ($1000-$1FFF)
4K Built-in RAM

4096-8191 $1000-$1FFF   USRPGM3K*
4096 bytes of built-in RAM.

This memory location block varies greatly with the amount of memory expansion RAM added to the VIC-20. We'll explore the alternatives as expansion RAM is added.

Figure 4–1. Unexpanded and Expanded RAM

No Expansion   3K Expansion   8K+ Expansion

0000 ($0000)		
BASIC/Kernal 1K Work Areas	BASIC/Kernal 1K Work Areas	BASIC/Kernal 1K Work Areas

1024 ($0400)

3K Expansion RAM	User PGM Area 6656 RAM	3K User Area for POKEs, ML

4096 ($1000)

Screen Area

4608 ($1200)

| User PRM Area 3584 RAM | | |

7680 ($1E00)

User PGM Area

| Screen Area | Screen Area | |

8192 ($2000)
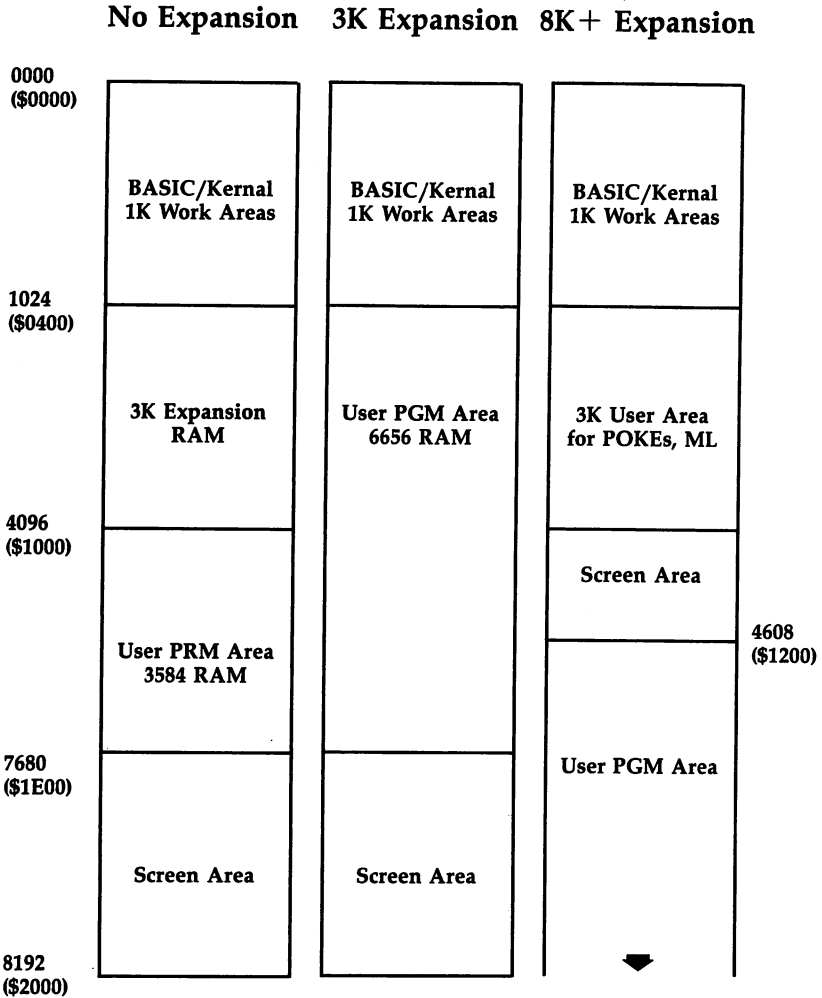
The VIC chip can address this area, so it's a good candidate for screen and character maps, once the pointer at location 43–44 ($2B–2C) has been adjusted. See that location for details. Location 55–56 ($37–38) points to the end of continuous RAM.

See Appendix B for details of the internal storage of BASIC and its programs, and Appendix E for explanations of the relocatable VIC-20 memory areas, user relocation of these areas, and memory configuration independent programming techniques.

The normal variations of this 4096-byte memory area are outlined in Figure 4-2.

Figure 4—2. 4K Built—in RAM

No Expansion   3K Expansion   8K+ Expansion

1024 ($0400)		
	User PGM Area 6656 RAM When 3K Filled	
4096 ($1000)		
User PGM Area 3584 RAM		Screen Area
		4608 ($1200)
7680 ($1E00)		
Screen Area	Screen Area	User PGM Area
8192 ($2000)		
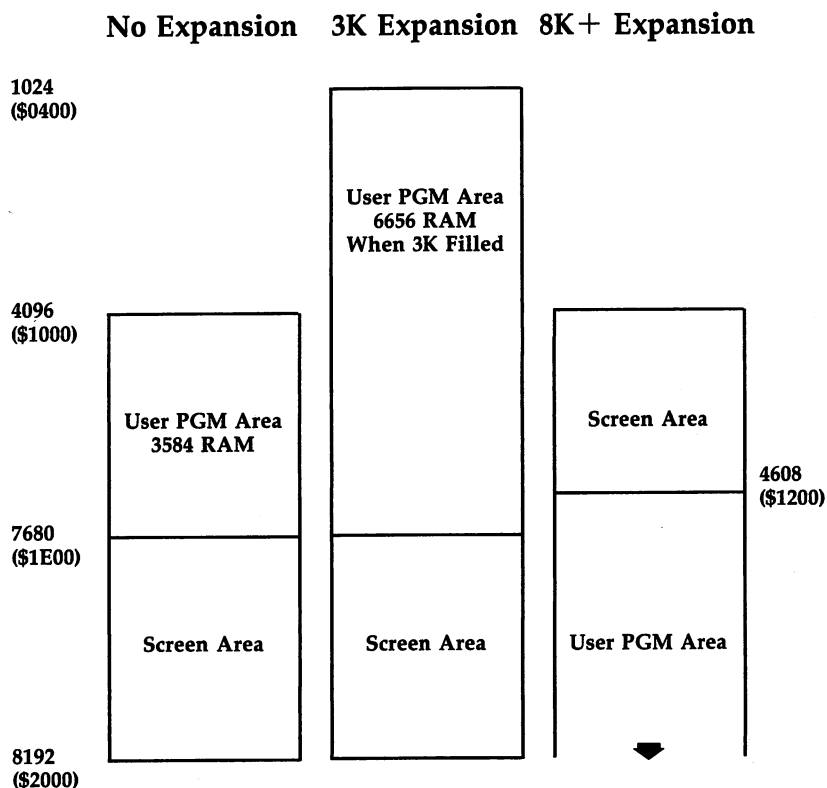
VIC—20 with no expansion added

4096–7679 $1000–1DFF   USRPGMOK*

3583 bytes of RAM for the BASIC program on an unexpanded VIC-20.

On an unexpanded VIC-20, the BASIC program area starts here. Location 43–44 ($2B–2C) points to the beginning of this area.

7680-8191 $1E00-1FFF   SCREEN*
(handy location)

Screen map RAM on VIC-20 with less than 8K expansion.

    The screen map for both an unexpanded VIC and a VIC with 3K of expansion is located in this area. On a VIC with 8K or more expansion, the screen map RAM is located at address 4096 ($1000). See the description at that location for an 8K+ expanded VIC-20 for an extensive discussion of the screen map RAM.

VIC-20 with 3K expansion RAM added

1024-7679 $400-1DFF
Continuation of RAM for the BASIC program on a 3K expanded VIC.

    A total of 6656 BASIC program RAM bytes are available when this expansion RAM block has been filled. Location 43-44 ($2B-2C) points to the beginning of the BASIC area at location 1024 ($400).

7680-8191 $1E00-1FFF   SCREEN*
(handy location)

Screen map RAM on VIC-20 with only 3K expansion.

VIC-20 with 8K or more of expansion RAM added

4096-4607 $1000-11FF   SCREENX*
(handy location)

Screen map RAM on VIC-20 with 8K or more expansion.

    Also called the video matrix, the screen map is managed by the Kernal screen editor using the table at 217 ($D9).

    The last six bytes of screen memory are not used for character indexes and you may use them for your own purposes.

    In this 512-byte area, the first 506 bytes contain indexes into the current character maps. Each index may be in the range of 0-255. Multiplying this index byte by eight and adding it to the address of the character map gives you the starting address of the dot-by-dot description of the character.

    The letter C is the fourth character in the ROM character maps, for example, so every C displayed on the screen would have 4 in the index byte for that position on the screen. Normally, eight bytes of the character map are used to define the character for screen display. A character is displayed as eight dots high and eight dots wide, the first byte defining the top row, the second byte defining the second row, and so on. This is why the index is multiplied by 8 to obtain the address within the character maps that the character begins on.

110

The byte can also index the first byte of an 8 by 16 character definition in the character maps. This is used when bitmapping the screen and is enabled by the value stored in location 36867 ($9003). In this case, only the first 253 bytes of the screen map are significant, each index byte being used to obtain the 8 by 16 definition of a double-sized character. To use this feature, you need to define a partial or full custom character set in RAM, pointed to by location 36869 ($9005). The double-sized character feature *does not* cause a normal character to be displayed twice as large, but rather allows the screen map to have a *unique index* value for every screen position, which is not available when 506 bytes are used to describe the screen. For more information about the character and screen maps, see locations 36869 ($9005), 32768 ($8000), and Appendix E. Bitmapping the screen is discussed in Appendix G, the relocatable screen map is described in Appendix E, and the color map for the screen is explored at location 37888–38399 ($9400–$95FF).

See location 36879 ($900F) for a background and border color chart and location 36878 ($900E) for the valid auxiliary colors.

The value for the screen map index byte for any given character in the ROM character maps can be derived from the ASCII code that you wish to place on the screen. This is useful when you are POKEing directly to the screen rather than PRINTing. See the character code chart in Appendix C.

If the high order bit of the screen POKE code is on, the reversed character set maps will be used for this character. If bit 6 is on, the symbol obtained when you also press the SHIFT key is used. For instance, a heart is seen on the screen for screen POKE code 19 (S), when 64 (bit 6) is added to it. However, if the alternate character set was selected, a capital S would appear. The symbol printed on the left-hand side of the key is displayed by pressing the key and the Commodore key simultaneously. This has its own screen code.

You can convert a character's ASCII value to its screen POKE code (P) by using the routine below in your own program.

Program 4-1. ASCII to Screen Code Conversion

```
10 C = ASC(X$) : REM ASCII VALUE OF FIRST CHARACTER
20 P = C : IF C > 127 AND C < 159 THEN P=0 : GOTO
   {SPACE}100
30 IF C < 64 THEN 100
40 P = C-32
50 IF C < 96 THEN P = P-32 : GOTO 100
60 IF C < 128 THEN 100
70 P = P-32
80 IF C > 191 THEN P = P-64
90 IF C = 255 THEN P = 30
100 ::::::::::
```

8K + 4096–4607

In order to see the values displayed on the screen, all you need to do is add two lines:

```
5 FOR I=32 TO 255:X$=CHR$(I):PRINT X$,
100 PRINT P:NEXT
```

Hold down the CTRL key to slow down the values as they scroll up the screen.

Character codes 128–159 have no POKE codes.

Routine SCRNOUT* may be examined for the techniques used, or you may JSR to it directly after you have positioned the cursor. See locations 780 ($30C) and 217 ($D9). Screen POKE codes (P) can be converted to ASCII (A) in a larger program by the following routine, with R being set to one if the screen POKE code was a reverse character:

Program 4-2. Screen Code to ASCII Conversion

```
50 P = PEEK(206) : R = 0
60 IF P > 127 THEN R = 1 : P = P AND 127
70 IF P < 32 OR P > 95 THEN A = P + 64 : GOTO 100
80 IF P > 31 AND P < 64 THEN A = P : GOTO 100
90 A = P + 32
100 RETURN
```

If you want to see the values on the screen without running a program of your own, simply insert these line in the routine above:

```
40 INPUT A$
50 P=PEEK(207):R=0:PRINT P
100 PRINT A:
110 GOTO 40
```

See Appendix C for a character code chart.

The screen on the VIC-20 is composed of 23 lines of 22 columns. If you know where you want to be on the screen, there are several ways to determine the screen map byte to put the POKE code in.

By using the formulas given, you can compute within a program the address of the particular byte on the screen map. You may also make these computations external to the program and store the values in DATA statements or variables to be used later. The latter technique is speedier but requires knowing in advance the positions required. Most likely you'll find a combination useful. Using these variables, you can calculate a byte's screen address:

SM=screen memory address
CM=color map address
SL=the desired line, expressed as 0–22

SC=the desired column, expressed as 0–21
CH=the character in screen POKE code
CL=the desired color code (0–7)
X = SL * 22 + SC : REM calculate displacement in maps
POKE SM + X, CH : REM place character on screen
POKE CM + X, CL : REM place color code in color map

You can then add or subtract one to move one position right or left, and add or subtract 22 to move one position up or down. Remember to test for the edges of the screen.

For turning a dot *on* or *off* on a high-resolution bitmapped screen with double-sized characters, you just need to alter the RAM custom character set byte that the dot is in. If BM is the address of the start of your custom character set, CM is the address of the start of the color map, X is the desired dot column minus one, and Y is the dot line minus one, then:

REM : find the 8 x 16 character within the character map
CR = INT (X / 8) + (INT (Y / 16) * 22)
REM : find the 8 wide row within the 8 x 16 character
RW = (Y / 16 − INT (Y / 16)) * 16
REM : find the byte number of the row in the character map
BY = BM + (16 * CR) + RW
REM : find the bit number within the byte
BT = 7 − (X − (INT (X / 8) * 8))
REM : turn on the bit
POKE BY, PEEK (BY) OR (2 ↑ BT)
REM : turn the bit yellow
POKE CM + CR, 7
REM: turn off the bit
POKE BY, PEEK (BY) AND (255 − (2 ↑ BT))

You can also use the PLOT routine in the Kernal to position the cursor to a given line/column. Pointers to the screen RAM and color RAM byte will be set by this routine for your use. Locations 217 ($D9) and 780 ($30C) have examples of using this and related routines. You'll find this the fastest method of determining a screen map position. In general, always try to use the provided, built-in ML subroutines in BASIC and the Kernal, since they are fast, debugged, and cost you very little previous RAM compared to programming the routine in BASIC.

See location 32768 ($8000) for the provided character maps. For information about creating your own character set maps, see Appendices E and G, and location 36869 ($9005).

See location 32768 ($8000) for a program to display or print any number of 8 x 8 character bytes in a large graphic matrix. This program will also note the decimal numbers used to define each pixel in each row.

113

The relocatable screen map is discussed in Appendix E. Appendix G describes bitmapping the screen, and location 36869 ($9005) describes changing the location of the screen map and using multiple screen and color maps.

Once a character map index is placed in the screen map, one more piece of information is needed to display the character on the screen—see the color map description starting at location 37888 ($9400). The relocatable color map is explained in Appendix E, and the use of the color map for the screen is explored at location 37888–38399 ($9400–$95FF).

See location 36879 ($900F) for a chart of background and border colors and 36878 ($900E) for the valid auxiliary colors.

4608-8191 $1200-1FFF   USRPGM8K*
First 3583 bytes of BASIC area on an 8K plus expanded VIC-20.

An 8K expanded VIC-20 starts BASIC at 4608 ($1200) and extends it to the end of expansion RAM, which can be as high as 32767 ($7FFF). Location 55 ($37) points to the end of expansion RAM.

If a 3K expansion is also present, BASIC will ignore it, so it's available for your own use. The VIC chip cannot address it, however, so screen or character maps cannot be placed there except as temporary SAVE areas invisible to the VIC chip.

Location Range: 8192-16383 ($2000-$3FFF)
8K Expansion Block 1

8192-16383 $2000-$3FFF   RAMBLK1*
8K RAM expansion block 1.

When this area is filled by RAM, BASIC starts at location 4608 ($1200) and any 3K expansion is ignored by BASIC. The total available user program space is 11,776 bytes. This area could also be filled with ROM, although I don't know of any cartridges now available that use this area.

The DIP switch on the Commodore 8K memory expansion board should be set so that switch 4 is on to fill this area. The other three switches should be in the off position.

A description of how to build your own 16K and 3K memory expansion boards can be found in *Tricks for VICs*.

Location Range: 16384-24575 ($4000-$5FFF)

8K RAM expansion block 2.

16384-24575 $4000-$5FFF   RAMBLK2*
8K RAM expansion block 2.

When this area is filled by RAM and the previous block is filled with RAM, BASIC still starts at 4608 ($1200) and any 3K expansion is ignored by BASIC. Total available user program space is 19,967 bytes. If the previous RAM expansion block is left empty, this block is ignored by BASIC since it stops looking for RAM as soon as it encounters an unfilled block. This can be useful when you wish to hide large amounts of memory from BASIC. As with block 1, this area could be filled with ROM instead of RAM.

Turn the Commodore 8K expansion board's DIP switch 3 on for this block.

Location Range: 24576-32767 ($6000-$7FFF)
8K Expansion Block 3

24576-32767 $6000-$7FFF   RAMBLK3*
8K RAM expansion block 3.

When this area and the previous two 8K blocks are filled by RAM, BASIC still starts at 4608 ($1200), and any 3K expansion is ignored by BASIC. 28,160 bytes are then available for user program space. If either of the two previous expansion blocks is empty, BASIC ignores this block, since it stops looking for RAM as soon as it encounters an unfilled memory block. You could use this to hide 8K blocks from BASIC. Cartridges frequently fill this area with ROM instead of RAM. Programmer's Aid, word processors, games, and some VICMONs may use this block. Autostart is not done for this block. Remember that only 40960 ($A000) resident cartridges autostart. Some cartridges reside at 40960 ($A000) and use this block also.

The Commodore 8K and 16K expansion boards should both have DIP switch 2 set on for this block.

Note. The Kernal prevents saving to locations above 32767 ($7FFF) to tape. However, disk is not restricted. See location 172 ($AC) for how the Kernal restricts tape saves to locations below 32768 ($8000)

Location Range: 32768-36863 ($8000-$8FFF)
4K ROM Character Maps

The following character pixel (picture element) maps in ROM are the built-in character set definitions used to form characters on

the screen. This is done by pointing via an index to the appropriate character definition. Each character is defined by an eight-by-eight bit grid, sometimes called a *character cell*. For example, locations 32776–32783 ($8008–800F) define the pixel modes for the capital *A* character:

Figure 4-3. The A Character Definition

Pixel Appearance	Location		Binary Image	Dec	Hex
	32776	($8008)	00011000	24	$18
	32777	($8009)	00100100	36	$24
	32778	($800A)	01000010	66	$42
	32779	($800B)	01111110	126	$7E
	32780	($800C)	01000010	66	$42
	32781	($800D)	01000010	66	$42
	32782	($800E)	01000010	66	$42
	32783	($800F)	00000000	0	$00

128 64 32 16 8 4 2 1

The DATA statement for this character, if you were using it in a custom character set, would be:

DATA 24,36,66,126,66,66,66,0:REM capital A

Character A is composed of eight bytes, one byte for each row. Each bit in each row represents a pixel and is set to one if the pixel is *on*. The *on* pixel image on the left side of Figure 4-3 is formed by the binary definitions on the right. Wherever a capital A is to appear on the screen map, the index of 1 is placed in the screen map RAM. 2 is the index for capital B, 3 for capital C and so on. 0 is the index for the character @. By adding the index multipied by eight to the address of the start of the character pixel map, the definition for that particular character can be found. Location 36869 ($9005) contains a pointer to the current character pixel maps. The index values stored in the screen map RAM are not the ASCII value for a character. They are referred to as screen POKE codes, and you'll find a chart of them in Appendix C. The ASCII code for a character can be converted to the screen POKE code and vice versa using the routines given at location 4096 ($1000).

When using the PRINT statement in BASIC, this conversion is done by the Kernal SCRNOUT* routine. For any device other than the screen, the ASCII type character code is used.

There are two separate and nonmixable character sets on the VIC-20: uppercase/graphics and lowercase/uppercase. Both sets are comprised of 256 characters, half of which are the reverse images of the characters. Two sets of 256 characters with 8 bytes used to map each character requires 4K (4096 bytes) of character pixel map ROM.

The keyboard can be used to determine which character pixel map is used. Press the Commodore key and the SHIFT key to switch between the two character sets and CTRL/RVSON or CTRL/RVSOFF as desired.

A program can PRINT the CTRL/RVSON or CTRL/RVSOFF, as well as PRINT CHR$(14) for lowercase, and PRINT CHR$(142) for uppercase. Using this, you can select the correct character set under program control. By using POKE 657,128 you can disable the SHIFT and Commodore key combination from switching to the alternate character set. The SHIFT, Commodore, and CTRL keys when pressed for any other purpose will operate normally. To reenable the SHIFT/Commodore combination, POKE 657,0. You can also achieve the same effect by PRINT CHR$(8) to disable character set switching and PRINT CHR$(9) to enable it.

The following program will display or print any number of eight-by-eight byte character map pixel description bytes. The program will diagram the pixels onto a large graphic matrix along with the associated decimal numbers that were used to define each pixel in each row. To cause the output to be printed rather than displayed, change OPEN 4,3 on line 10 to OPEN 4,4. Line 20 may be changed from 32768 to any other desired starting point. If left as is, the entire 128 uppercase nonreversed map will be diagrammed. To start the diagram program at a specific letter, multiply the screen POKE code of the first character by eight and add that to the character pixel map starting address. To diagram 12 letters starting with F, you would change line 20 to:

FOR BEGIN = 32768 + (8 * 6) TO 32768 + (8 * 18) STEP 8 because 6 is the POKE code for F, and 18 is the sum of 6 and 12.

Program 4-3. Character Diagrams and Bit Values

```
10 OPEN 4,3 :REM CHANGE 4,3 TO 4,4 FOR PRINTER OUT
   PUT
20 FOR BEGIN = 32768 TO 32768 + ( 8 * 128 ) STEP 8
30 PRINT#4,"PIXEL MAP OF CHARACTER"
40 PRINT#4,"STARTING AT"BEGIN
50 PRINT#4,"76543210 DECIMAL"
60 FOR X = 0 TO 7 : Z = PEEK(BEGIN+X) : FOR Y = 7
   {SPACE}TO 0 STEP -1
```

117

```
70 W = Z AND (2↑Y) : IF W THEN PRINT#4,"{RVS}
   {OFF}"; : GOTO 90
80 PRINT#4,".";
90 NEXT Y : PRINT#4,"{3 SPACES}"Z : NEXT X : PRINT
   #4 : NEXT :END
```

Character Set 1 Graphic Set

32768–33791 $8000–83FF   CASEU*
Uppercase and graphics nonreversed screen character map.

    This area consists of 1024 bytes describing the 128 uppercase and graphics character set when RVSOFF is selected and no SHIFT/Commodore key combination is in effect.

    This is the default character map, used when you turn the computer on. The first eight-byte pixel map corresponds to screen POKE code 0, the @ character, the second to the capital letter A, and so on. See Appendix C for a character chart in which the 128 character pixel maps in this area correspond to screen POKE codes 0 through 128. See the discussion at the beginning of this section for details of how to select this character map from a user program.

33792–34815 $8400–87FF   CASEURV*
Reversed uppercase and graphics screen character map.

    This area contains 1024 bytes describing the 128 reversed uppercase and graphic character set when RVSON is selected and no SHIFT/Commodore key combination is in effect. The pixel maps are in the same order as the character pixel maps at 32768 ($8000,) but each bit is reversed to display the character in a dark on light background pattern.

    You can use this character set for individual characters by ORing the high order bit of the screen POKE code for those screen indexes that you wish displayed in reverse. If SP were the screen map position you wished to reverse, POKE SP,1OR128 would cause that one position to be a reverse capital A. The same effect is accomplished by adding 128 to the screen POKE code. See Appendix C for a code chart of the screen codes.

Character Set 2 Text Set

34816–35839 $8800–88FF   CASEL*
Lowercase and uppercase nonreversed screen character map.

    This section of 1024 bytes describes the 128 lowercase and uppercase character set when both the RVSOFF and the SHIFT/Commodore key combination are in effect. This character set is

mutually exclusive with the first set, in that characters from the two character sets cannot be on the screen at the same time. An exception is with the raster interrupts, which are discussed at location 36868 ($9004). However, this set *does* include all the capital letters, numbers, punctuation, and a few of the graphic symbols from the first character set maps. Four additional graphic symbols are included, one of them a check mark. The lowercase characters (lowercase g, j, p, q, y) do not employ descenders (the part of the letter that extends below the bottom of other characters).

35840-36863 $8C00-8FFF   CASELRV*
Reversed lowercase and uppercase screen character map.

These 1024 bytes describe the 128 reversed lowercase and uppercase character set when the RVSON and SHIFT/Commodore key combination are in effect. The pixel maps are in the same order as the character pixel maps at 34816 ($8800), but each bit is reversed. (Previously *on* bits are instead set to 0, or *off*.)

You can use this character set for individual characters by ORing the high order bit of the screen POKE code for those screen indexes that you wish shown in reverse. You can also SHIFT the letter to uppercase by ORing the sixth bit. IF SP is the screen map position you wanted to reverse and SHIFT, POKE SP,1OR192 would cause that one screen position to be a reverse capital A. The same effect is accomplished by adding 192 to the screen code, as in POKE SP,1+192. See Appendix C for a code chart of the screen POKE codes, and the subject index for uppercase, lowercase, SHIFT, Commodore, RVSON, and RVSOFF.

Chapter 5

Video Interface Chip

Video Interface Chip

Location Range: 36864-37135 ($9000-$910F)
6560 Video Interface Chip

The 6560 Video Interface Chip, also known as the VIC chip, provides the color, sound, character pixel screen mapping, and paddle/light pen support for the VIC-20.

Jim Butterfield has thoroughly explored the inner workings of the VIC chip in his series of articles entitled "Visiting the VIC-20 Video," which began with the May 1983 issue of *COMPUTE!*. Of course, the *VIC-20 Programmer's Reference Guide* can serve as an excellent resource as you explore the VIC chip. For a more general exploration of the VIC chip, see *The VIC-20 User Guide* (not to be confused with the manual that comes with your VIC computer). MOS Technology publishes specification sheets for the VIC chip, and the schematic diagram in the back of the *VIC-20 Programmer's Reference Guide* is very useful for understanding the chip.

The VIC-20 output video signal can even be videotaped on a home Video Cassette Recorder (VCR) since NTSC standards are followed in the VIC-20.

The VIC chip also provides the 6502 and 6522 chips with a system clock of 1.1082 megahertz. This is derived from the 14.31818 megahertz crystal in the VIC-20.

The 6560 VIC Chip Registers

The 6560 VIC chip registers are actually located in the VIC chip itself. There is no RAM that corresponds to these registers. When POKEing or PEEKing these locations, you are actually looking inside the VIC chip itself.

Pressing the RUN/STOP-RESTORE keys causes these values to be reinitialized by the INITVIC* routine from the table at VICINIT*.

You'll be looking closely at various bits in each register byte, so for quick reference, look at Table 5-1. Bit positions and values are outlined in this table.

Table 5-1. Bit Positions and Values

BIT	7	6	5	4	3	2	1	0
---	---	---	---	---	---	---	---	---
DEC	128	64	32	16	8	4	2	1
HEX	$80	$40	$20	$10	$08	$04	$02	$01

36864 $9000   VICCRO*

(handy location)

Left edge of TV picture and interlace switch.

Bit 7: Interlace scan bit. Default value: 0.

When set to 1, this bit causes every other full sweep of the TV to be skipped.

A TV picture is formed by 60 screen scans every second. The screen is painted, then painted again in between the first set of lines. Each half of these lines is painted 30 times a second, producing 30 complete frames per second. There are 525 lines in the full TV picture, 262.5 lines per painting. At 30 frames per second, this means that 15,750 lines are painted every second. Setting this bit *on* (1) causes one of those halves to not paint the VIC-20 picture image. Flicker results unless another multiplexed input to the TV is used to create the composite pictures. This overlay effect requires special equipment.

Some TV sets may require turning this bit *on* to stop picture *jitters*. Many commercial programs for the VIC-20 provide you with a command to turn this bit on, in case your TV set shows these jitters.

To turn this bit *on*:

POKE 36864,PEEK(36864)OR 128

To turn bit *off*:

POKE 36864,PEEK(36864)AND 127

Bits 6-0: Horizontal TV picture origin. Default value: 5.

These bits can be used to adjust the position where the first character appears on the left side of the TV picture. Possible values in this location are between 0 and 127 (although those above 16 seem to confuse BASIC), with larger numbers moving the characters to the right. Every increase or decrease of this number by one shifts the TV display four pixels right or left. If this location is set to 0, column 3 would be on the extreme left edge of the TV picture. As the picture shifts on the TV, the border expands and contracts. Most of the more recent software packages for the VIC-20 allow the user to adjust this value for optimum centering of the picture on the TV, usually through cursor keys or joystick action.

To change these bits, you can POKE 36864,PEEK(36864)AND 128 OR XXX, where XXX is any value between 0 and 127. Values above 16, remember, seem to confuse BASIC.

Here's a sample routine to adjust bit 7 and center the TV picture using the cursor keys. Type it in and run it to see this register's effect on the TV picture.

Program 5-1. Centering the Picture

```
100 ::: REM USE CURSOR KEYS TO CENTER SCREEN
110 ::: REM USE UPARROW KEY TO TOGGLE INTERLACE
120 ::: REM USE EQUAL SIGN KEY TO RESTORE DEFAULT
    {SPACE}VALUES
130 BASE=36864
140 GET A$:IF A$="" THEN 140
150 A=ASC(A$)
160 IF A$="=" THEN POKE36864,5 : POKE36865,25 : GO
    TO 140
170 IF A=13 THEN 330
180 IF A<>29 AND A<>157 GOTO 250
190 H=PEEK(BASE) AND 127 : I=PEEK(BASE) AND 128
200 IF H>16 THEN H=16
210 IF H<2 THEN H=2
220 IF A=29 THEN POKEBASE,I OR (H+1) : GOTO 140
230 POKEBASE,I OR (H-1)
240 GOTO 140
250 IF A=94 AND PEEK(BASE)>127 THEN POKE(BASE),PEE
    K(BASE)-128 : GOTO 140
260 IF A=94 THEN POKE(BASE),PEEK(BASE)+128 : GOTO
    {SPACE}140
270 IF A<>17 AND A<>145 GOTO 140
280 V=PEEK(BASE+1)
290 IF A=17 THEN V=V+1 : IF V>136 THEN V=136
300 IF A=17 THEN POKEBASE+1,V : GOTO 140
310 V=V-1 : IF V<0 THEN V=0
320 POKEBASE+1,V : GOTO 140
330 END
```

36865 $9001   VICCRI*   25
(handy location)

Bits 7-0: Vertical TV picture origin.

This location specifies where the top line of characters is displayed on the TV. The picture can be relocated by the addition or subtraction of one from this location; subtraction raises the picture on the screen, and addition lowers it. Each change of one in this value moves the TV display two pixels. A value of zero here causes the middle of the fourth line to be at the top of the TV.

See location 36864 for a sample routine to adjust the picture centering with the cursor keys.

By entering the following, you can make the TV display a screen completely in the border color:

POKE 36865,255

You could use this to hide the screen while you formatted it, for example.

The article "VIC Memory—The Uncharted Adventure," by David Barron and Michael Kleinert, in *COMPUTE!'s First Book of VIC*, is a useful reference. Simply subtract 16 from the locations listed over 36880 and you're back to the VIC chips which perform the functions the authors have noted.

A sample program in this article demonstrates the use of this location to implement smooth scrolling from the bottom of the screen upwards.

To set this byte, just POKE the desired value without adding any ANDs or ORs.

You can try this technique of scrolling the screen by entering the following program.

Program 5—2. Screen Scrolling

```
10 REM **** SCREEN SCROLLING DEMO *** 36865
70 POKE36879,0+8
71 C$="{WHT}{RED}{CYN}{PUR}{GRN}{BLU}{YEL}"
80 FORZ=1TO7:C1$=MID$(C$,Z,1):POKE36865,132
100 PRINTC1$"{CLR}*THIS IS A DEMO OF A{2 SPACES}SC
    ROLLING EFFECT USING LOCATION 36865.*"
105 PRINT"{RVS}{22 SHIFT-SPACE}"
110 PRINT"CHANGE THE VALUE AT{3 SPACES}THE END OF
    {SPACE}THE 'Y' FORLOOP IN LINE 120 TO"
115 PRINT"CONTROL SCROLL SPEED."
120 FORX=131TO24STEP-1:POKE36865,X:FORY=1TO80:NEXT
    :NEXT:NEXT:GOTO80
```

36866　　$9002　　VICCR2*　　150
(handy location)

Number of columns displayed, part of screen map address.

Bit 7: Default value: 1. This bit serves as bit 9 of the 14-bit screen map address used by the VIC chip.

If this bit is set to 0, the screen map RAM is located on a 1024-byte boundary, and the color map begins at location 37888 ($9400). When this bit is set to 1, the screen map RAM starts on a 512-byte boundary and the color map is at location 38400 ($9600). See location 36869 ($9005) and also Appendix E for screen relocation and multiple screens/color maps details.

The Kernal routine INITSK* sets this bit on at power-on/reset and RUN/STOP–RESTORE if the screen map should be located on a 512-byte boundary.

Bits 6–0: Default value: 22. These bits contain the number of character columns displayed on each TV display line. This value can range from 0 to 25. Although some stunning graphics can be created by raising this value and eliminating the border, the Kernal doesn't really want anything to do with a lengthened screen line. So avoid PRINT, cursor movement, and the screen editor if you set this value to anything other than 22. You'll have to use screen POKE codes, develop your own formula for X/Y plotting, arrange for a larger screen map in RAM, and adjust the color map accordingly.

There's also the problem of having only 256 indexes in the screen map RAM. So when working in high resolution, there will have to be *some* duplicate characters on the screen, but the screen design can usually accommodate this. The amount of the line visible on the TV screen can vary among makes and models, so experiment with your TV set. Once you've seen a full-screen display, you'll probably want to explore this further. Despite the amount of work involved, it's worth the effort.

Lowering this value may prove handy to you. Twenty-two TV columns seem narrow enough, but maybe an idea will occur to you.

To set this value: POKE 36866,(PEEK(36866) AND 128) OR XXX where XXX is a value from 0 to 25.

36867 $9003   VICCR3*   174 or 46
(handy location)
Number of character lines displayed, part of raster location.

Bit 7: Default value: 1/0. Raster beam location bit 0. Combine this, as the low order bit, with location 36868 ($9004). See that location for more information.

Bits 6–1: Default value: 46. Number of character lines displayed on the TV picture multiplied by two. By varying this value, you can make the border expand or shrink and use fewer or more TV lines. The same considerations apply for these bits as for location 36866 ($9002) bits 6–0, so see that location for details. The value in this position is multiplied by two because the low order bit of this byte is used for other purposes. Thirty to thirty-two seems to be about the maximum useful number of screen lines.

To set these bits you can POKE 36867,(PEEK(36867)AND 129) OR (XXX*2), where XXX is the number of lines desired.

Bit 0: Default value: 0. Character size 8 x 8, or 8 x 16 pixels
When bitmapping the screen or doing other custom character set tasks, this bit can be set to specify the double-sized character option

(8 pixels wide by 16 high). The bottom of the border will drop off the TV, and you'll see only eleven and a half double-sized character lines (unless you also adjust the vertical TV picture origin in 36865, $9001). A zero in this bit position specifies 8 x 8 character size, while a one selects 8 x 16 characters. When double-sized characters are used, only the first half of the screen map is significant, each POKE code being used to obtain the 8 x 16 definition of a double-sized (twice as high—16—rather than 8 dots high) character to display.

To use this feature, you need to define a partial or full custom character set in RAM, pointed to by location 36869 ($9005). The double-sized character feature *does not* cause a normal character to be displayed twice as large. Rather, it allows the screen map to have a unique index value for every screen position, which is not available when 506 bytes, with a range of 0–255 in each, are used to describe the screen.

To set this value:

POKE 36867,PEEK(36867) OR 1

For more information about custom character sets, see locations 36869 ($9005), 32768 ($8000), and Appendix E. Bitmapping the screen and relocating the screen map are described in Appendix G.

36868	**$9004**	**VICCR4***
		(handy location)

Raster beam location bits 8–1.

Bits 7–0: When combined with the high order bit of location 36867 ($9003) as the low order bit, this value tracks the location of the electron beam as it refreshes the TV picture. If you were to use only this location to reference the raster location, you would only sense every other TV line. This raster location is used by the light pen sensing function of the VIC chip, and is latched into locations 36870 ($9006) and 36871 ($9007).

This location contains the line number that the raster beam is currently scanning. In an ML routine, you can test the location of the raster at any time and execute a command when a certain point is reached. You could switch color schemes, character map, or some other effect. A public domain program called "Colortrick2" by Joe Watson creates a rainbow border using this technique. You need to be able to constantly intercept the refresh of the screen, demanding an ML loop with severe timing restrictions. Experimentation is the key. The TV picture could be segmented with different color backgrounds displaying varying types of information. The possibilities are quite varied for using this location for TV picture customization.
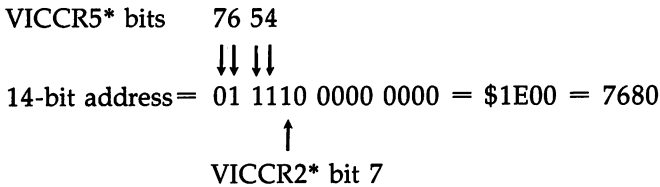
36869 $9005   VICCR5*   240

(handy location)

Screen map and character map addresses.

Bits 7–4: Default value: 240. These bits serve as bits 13–10 of the screen map address, combined with bit 7 of 36866 ($9002) to form the 14-bit VIC chip screen map address.

Let's put together the 14-bit address as seen by the VIC chip, using the 240 ($F0) default value in this location and the default 1 in the high order bit of VICCR2*:

Figure 5–1. 14—Bit Address

VICCR5* bits   76 54

                 ↓↓ ↓↓

14-bit address =   01 1110 0000 0000 = $1E00 = 7680

                        ↑

                VICCR2* bit 7

Notice that 7680 is the default unexpanded VIC-20 screen map location.

Also note that the high order bit of VICCR5* was changed to zero when it was put into the high order bit of the 14-bit address. This is the other aspect of the screen map address that confuses many programmers. Bit 7 of VICCR5* *must always be one*, but is *viewed* as though it is *zero*. I won't even try to rationalize this; it's just the way it works.

Remember that bit 7 of VICCR2* determines the color map address (37888 or 38400) and, as you can now see, indicates whether the screen map is on a 1024-byte or 512-byte boundary.

You can use the following formula to find the screen map:

SM = 4*(PEEK(36866) AND 128) + 64*(PEEK(36869) AND 112)

You can also use SM = PEEK(648)*256.

Screen Map Available Locations

(Any 1K or 512-byte boundary must be selected in unexpanded RAM, but the following locations are the unused areas.)

Table 5-2. Screen Map Locations

Screen Decimal	Location Hex	Color Map	36869 Bits 7-4	36866 Bit 7	POKE Values (see notes below) -X-	-Y-
----	-----	-----	----	-----	-X-	-Y-
4096	$1000	37888	1100	0	192	0
4608	$1200	38400	1100	1	192	128
5120	$1400	37888	1101	0	208	0
5632	$1600	38400	1101	1	208	128
6144	$1800	37888	1110	0	224	0
6656	$1A00	38400	1110	1	224	128
7168	$1C00	37888	1111	0	240	0
7680	$1E00	38400	1111	1	240	128

You'll note that, unfortunately, no expansion RAM locations can be used for this map.

To set the screen address to any of the valid addresses listed above, the value in the first POKE value column replaces X in the statement: POKE 36869,(PEEK(36869) AND 15) OR X. The second POKE value number replaces Y in the statement POKE 36866,(PEEK(36866) AND 127) OR Y. Finally, POKE 648,Screen Address/256:CLR.

The second POKE (using the Y value) specifies whether the map address starts on a 1K or 512-byte boundary.

For example, to set 7680 as the screen address you would enter:

POKE 36869,(PEEK(36869) AND 15) OR 240
POKE 36866,(PEEK(36866) AND 127) OR 128
POKE 648,7680/256:CLR

An alternate method is to POKE 648,Screen Address/256:SYS 58648.

Jim Butterfield has demonstrated setting this location to binary 1000 (by using 128 as an X value above) and turning off the high order bit of location 36866 ($9002) (by using 0 as a Y value above), to temporarily set the screen map at location zero so that you can observe the symbolic activity in the first two pages of memory.

On an unexpanded or 3K expanded VIC-20, the screen address default is 7680 ($1E00), while on an 8K plus expanded VIC-20 the screen address is set to 4096 ($1000).

Although only one screen map may be displayed at any given moment, you may have more than one area for a screen map, with its own color map dedicated to it. If SM is the screen map address that you want displayed, you need to:

- Change the address bits here
- Change the page number in location 648
- Alter locations 217–228 ($D9–E4) to SM/256+128
- Change locations 229–240 ($E5–F0) to SM/256+129
- And set bit 7 in location 36866 ($9002) for the screen to be currently displayed

The screen won't *flip* until a carriage return is sent to it. You may want to save the screen line link table 217–240 ($D9–F0) bytes somewhere rather than resetting them to unlinked lines. Be sure to pick an alternate screen address that uses the other color map. The easiest combination is two screen addresses, with this location having the same bits 7–4 and bit 7 of 36866 ($9002) reversed. Examples of these combinations would be 4096 and 4608; 5120 and 5632; 6144 and 6656; or 7168 and 7680. Then this location won't need to be altered.

For example:

```
SM=4096 : REM select screen one
POKE 36866,(PEEK(36866) AND 127)
POKE 648,SM/256
FOR X=217 TO 228:POKE X,PEEK(648)+128:NEXT
FOR X=229 TO 240:POKE X,PEEK(648)+129:NEXT
PRINT"home": REM screen one selected
```

```
SM=4608 : REM select screen two
POKE 36866,(PEEK(36866) AND 255)
POKE 648,SM/256
FOR X=217 TO 228:POKE X,PEEK(648)+128:NEXT
FOR X=229 TO 240:POKE X,PEEK(648)+129:NEXT
PRINT"home": REM screen two selected
```

Related screen fields are: 209 ($D1), 217 ($D9), 243 ($F3), 648 ($288), 4096 ($1000), 7680 ($1E00), and 32768 ($8000). Also see *screen* and *screen map* in the subject index.

Bits 3–0: Default value: 0. These serve as bits 13–10 of the character map 14-bit address. They are used to form the 14-bit VIC chip address that points to the beginning of the current 2048-byte (2K) character map or custom character set.

The Kernal routine SETKEYS* changes this value when the Commodore key and SHIFT are pressed together, to make sure the proper character map is used.

Take a look at Figure 5-2 for a moment to see the details of the 14-bit address that the VIC chip uses.

Figure 5–2. 14–Bit Address, Character Map

VICCR5* bits   32 10

⇃⇂ ⇃⇂

14-bit address= 00 0000 0000 0000 = $0000 = 32768

    The default unexpanded VIC-20 character map location begins at 32768, the value obtained in Figure 5-2.

    You probably suspect by now that there's some special rule that applies to this situation, since obviously the character map doesn't begin at address zero, and because $0000 is hardly the same as 32768. Because of the way the VIC chip sees the memory in the VIC-20, zero here means 32768 ($8000) to the VIC chip. If the peculiar vision of the VIC chip interests you, you'll want to read Jim Butterfield's series of articles entitled "Visiting the VIC-20 Video," which started in the May 1983 issue of *COMPUTE!* magazine. How this all works is interesting, but you need to know how to use this location. A few facts will help:

    ● The first two bits (bits 3 and 2) of these four bits specify the address of the start of the character map in 4096-byte increments.

    ● The second two bits (bits 1 and 0) contain the number of times that 1024 ($400) must be added to the value in bits 3 and 2.

    ● If bits 3 and 2 are both zero, 32768 ($8000) is used, plus the settings in bits 1 and 0.

    ● The character map *must* start on a 1024-byte boundary. You can see that there's no way to specify an amount smaller than 1K.

    Here's a table of the possible combinations for the character map.

Table 5–3. Character Map Locations

Bits 3210	Decimal Value	Hex Value	Decimal Address	Hex Address	Type of characters
0000	0	0	32768	$8000	uppercase
0001	1	1	33792	$8400	uppercase reversed
0010	2	2	34816	$8800	lowercase
0011	3	3	35840	$8C00	lowercase reversed
0100	4	4	36864	$9000	don't use—VIC chip
0101	5	5	37888	$9400	don't use—color map
0110	6	6	38912	$9800	don't use—I/O block
0111	7	7	39936	$9C00	don't use—I/O block
1000	8	8	0000	$0000	don't use—low RAM
1001	9	9	1024	$400	can't be accessed
1010	10	A	2048	$800	can't be accessed
1011	11	B	3072	$C00	can't be accessed
1100	12	C	4096	$1000	custom character set
1101	13	D	5120	$1400	custom character set
1110	14	E	6144	$1800	custom character set
1111	15	F	7168	$1C00	custom character set

To set the address, you can use POKE 36869,PEEK(36869)AND 240 OR X where X is the value from the decimal column in Table 5–3.

You'll note that no expansion RAM locations can be used for this map.

Bit 3 can be thought of as a ROM/RAM switch. When it's on (set to 1), RAM is being used; otherwise, the character map is in ROM.

There was a misprint for this location in the *VIC-20 Programmer's Reference Guide*. Page 215 in the edition I have shows the formula:

POKE 36869,PEEK(36869) AND 15 OR (X*16)

This should read: POKE 36869,PEEK(36869) AND 240 OR X.

Because of the way this pointer is used in the VIC chip, once you set the start-of-character map pointer, the next 128 characters are obtained from the *next higher (with wrap) bits 3–0 setting*. This is a great feature of the VIC chip. You can access the next 128 characters by pressing RVSON, the next by pressing RVSOFF/Commodore key/SHIFT, and yet another by pressing RVSON again. For example, if you composed a custom character set of 128 characters at location 7168 ($1C00) and pressed the RVSON key, the ROM uppercase/graphic set at location 32768 ($8000) would be used as long as RVSON is in effect. Pressing RVSOFF/Commodore key/SHIFT selects the 128 reversed uppercase/graphics. With this feature, you can use three-quarters, one-half, one-quarter, or none of the ROM-based character maps with your own custom character set. By setting the start of character map bits to refer to location 4096 ($1000), you will be able to refer to 512 of your own characters. This is the location used for the character set when bitmapping the screen.

If you need fewer than 128 custom characters, you can copy the others that you'll need from ROM into your own set, or simply not use the remaining characters. The latter technique is used when composing a custom character set for location 7168 ($1C00) on an unexpanded VIC-20, since there is only room for 64 custom characters (8*64=512 bytes) before the start of the screen RAM map area.

Rather than printing the CHR$ codes for character set switching and RVSON or RVSOFF, you can POKE this location, using the value for the needed character set if you want to change all the characters on the screen. Also see locations 199 ($C7) and 657 ($291).

Location 32768 ($8000) contains more information about the character map.

Refer to the color map description starting at location 37888 ($9400).

See location 32768 ($8000) for a program to display or print any

number of 8 x 8 character map pixel description bytes in a large graphic matrix with their decimal numbers noted.

See the appendices for additional information on bitmapping the screen, custom character sets, and relocating the screen and character maps.

36870　　　$9006　　　VICCR6*　　　0
(handy location)

Light pen horizontal screen location.

This address contains the pixel location of the light pen photocell from the left side of the TV. It's latched here when the raster passes the pen tip. It is the light pen that detects the beam of light from the raster and causes the position to be latched, not the other way around. From the values placed here, which range from 0 to 255, you can determine the pixel location by using
COL=PEEK(36870)*.69.

If your light pen has a switch on it, you will find that location 37151 ($911F) bit 5 is where it can be detected.

You will want to *debounce* this value as it is extremely sensitive to imperceptible movement. *Debouncing* is accomplished by saving this value, waiting a reasonable amount of time (maybe a half second—30 jiffies), and comparing the current value to the last-saved value until they are equal or as equal as necessary to determine the selected location/option.

Incidentally, light pen detection on the VIC chip is an optional feature. Since the VIC-20 has this optional feature, the VIC chip is technically a 6560-101. That's why there's a white spot on the chip.

You can refer to several excellent articles on the VIC's light pen option. "A Light Pen for Under $10," by William Hale, in the August 1982 issue of *COMPUTE!*; "Basics of Light Pen Operation," by Robert Peck, in the March 1981 issue of *COMPUTE!*; and "An Inexpensive Light Pen for the VIC-20, C-64, and Atari," by David Bryson, in the June 1983 issue of *Micro*, are three such articles.

See location 36868 ($9004) for additional information.

36871　　　$9007　　　VICCR7*　　　0
(handy location)

Light pen vertical screen location.

This address contains the pixel location of the light pen photocell from the top of the TV. It's latched here when the raster passes the pen tip.

You can determine the pixel location by using
ROW=PEEK(36871)*.722.

See location 36870 ($9006) for debouncing suggestions.

36872 $9008   VICCR8*   255
(handy location)

Potentiometer X/Paddle X value.

The analog-to-digital converters in the VIC chip can convert variable resistance to digital values ranging from 0 to 255, incremented by one for every 1K ohms of resistance. No connection means infinite resistance, so 255 is the default value in this location. The game port pin 9 is used for this X value, pin 5 for the Y value, and pin 8 for a common ground.

Paddle values: right=0, left=255.

See the articles "Using the VIC Game Paddles," by David Malmberg, in the April 1982 issue of *COMPUTE!*, and "$20 VIC Digitizer," by Jeff Knapp, in the September 1982 issue of *COMPUTE!*, for details on the use and construction of game paddles.

36873 $9009   VICCR9*   255
(handy location)

Potentiometer Y/Paddle Y value.

Paddle values: right=0, left=255.

See location 36872 ($9008) for additional information.

36874 $900A   VICCRA*   0
(handy location)

Relative frequency of sound oscillator 1 (bass).

Bit 7: Switch to enable (1) or disable (0) this oscillator.

By turning off the sound for this oscillator with POKE 36874,PEEK(36874) AND 127, you can leave the desired value in this register for later use. To turn this back on, you can use POKE 36874,PEEK(36874) OR 128.

Bits 6–3: Low sound voice (sawtooth waveform).

This is the softest sound oscillator, with a range of values from 0 (no sound) to 127 (highest tone). The range of tones produced runs from 31 to 3995 hertz. To compute the value for this location from a given frequency, use the following line:

X=INT(128-(3995/FREQ))

where X is the value for this location.

(Use 4329 rather than 3995 if you are using the European PAL television system.)

The octave ranges for the three VIC-20 oscillators overlap, giving a combined range of five octaves, in this way:

36875

Figure 5–3. Octave Range Overlap on the VIC

Location		Values

```
  Location              Values
                7———►67———►97———►111—►127
36874  $900A   octave1  octave2  octave3    extra
                  7———►67———————►97———►111—►127
36875  $900B        octave1  octave2  octave3   extra
                      7———————►67———————►97—►111—►127
36876  $900C            octave1    octave2   octave3   extra
            Low notes ——————————————————————————►high notes
```

You can also convert particular notes to numerical values to enter in the sound registers.

Table 5–4. Musical Note to Numerical Value

Note	Octave 1	Octave 2	Octave 3	
C	7	67	97	112
C#	15	71	99	113
D	19	73	100	
D#	23	75	101	⇓
E	31	79	103	⇓
F	35	81	104	Too close to
F#	39	84	105	distinguish
G	47	87	107	⇓
G#	51	89	108	⇓
A	55	91	109	⇓
A#	59	93	110	127
B	63	95	111	

36875 $900B   VICCRB*   0
(handy location)

Relative frequency of sound oscillator 2 (alto).

Bit 7: Switch to enable (1) or disable (0) this oscillator.

By turning off the sound for this oscillator with POKE 36875,PEEK(36875) AND 127, you can leave the desired value in this register for later use. You can turn this back on with POKE 36875,PEEK(36875) OR 128.

Bits 6–0: Medium sound voice (pulse waveform).

These bits serve as the medium sound oscillator, with a range of values from 0 (no sound) to 127 (highest tone). Its range is from 63 to 7990 hertz. To compute the value for this location from a given frequency, you can use:

$X = INT(128 - (7990/FREQ))$

(Use 8659 rather than 7990 if you are using the European PAL television system.)

See location 36874 ($900A) for an octave range figure and a musical note to numerical value table.

36876 $900C   VICCRC*   0
(handy location)

Relative frequency of sound oscillator 3 (soprano).

Bit 7: Switch to enable (1) or disable (0) this oscillator.

Using POKE 36976,PEEK(36876) AND 127 turns off the sound for this oscillator and lets you leave the desired value in this register for later use. To turn this back on, simply enter POKE 36876,PEEK(36876) OR 128.

Bits 6-0: High sound voice (pulse waveform).

This oscillator's values range from 0 (no sound) to 127 (highest tone). Its range is 127–15,980 hertz. To compute the value for this location from a given frequency, you can use:
$X = INT(128 - (15980/FREQ))$
(Use 17320 rather than 15980 if you are using the European PAL television system.)

See location 36874 ($900A) for an octave range figure and musical note to numerical value table.

36877 $900D   VICRD*   0
(handy location)

Relative frequency of sound oscillator 4 (noise).

Bit 7: Switch to enable (1) or disable (0) this oscillator.

You can turn off the sound for this oscillator with POKE 36877,PEEK(36877) AND 127. This will leave the desired value in the register for use later. POKE 36877,PEEK(36877) OR 128 will turn it back on.

Bits 6-0: Noise voice (square waveform).

The sharpest sound oscillator, with values ranging from 0 (no sound) to 127 (highest tone), its range is from 252 to 31960 hertz. Use the following formula to compute the value for this location from a given frequency:
$X = INT(128 - (31960/FREQ))$
(Use 34640 rather than 31960 if you are using the European PAL television system.)

36878 $900E   VICCRE*   0
(handy location)

Sound volume and auxiliary color.

Bits 7–4: Auxiliary color for multicolor mode. A bit value of 11 selects this auxiliary color. You can use any of the 16 colors:

Black	0	*0*	Orange	8 *128*
White	1	*16*	Lt Orange	9 *144*
Red	2	*32*	Pink	10 *160*
Cyan	3	*48*	Lt Cyan	11 *176*
Purple	4	*64*	Lt Purple	12 *192*
Green	5	*80*	Lt Green	13 *208*
Blue	6	*96*	Lt Blue	14 *224*
Yellow	7	*112*	Lt Yellow	15 *240*

   However, to set these four bits, you don't simply POKE the above numbers into location 36878 ($900E). Instead, to set these bits, you'd use: POKE 36878, (PEEK(36878) AND 15) OR (X*16), where X is the color value shown above. For instance, to select light yellow (15) as the auxiliary color, you'd POKE 36878,(PEEK(36878) AND 15) OR 240.

Bits 3–0: Sound volume 0 (low) to 15 (high).
   These four bits set the combined volume of all sound oscillators.
   Turning the sound volume to zero does not turn *off* the oscillators; this is done by turning bit 7. The difference between two settings, an increment apart, of this value may be hardly noticeable on a TV or monitor due to its limited sound system.
   A stereo system, however, will demonstrate the full range of volume and tonal control. The VIC-20 puts out a high quality monophonic sound signal. Use AUX IN or TAPE IN on your sound system, not a turntable input.

36879 $900F   VICCRF*   27
(handy location)

Background color, border color, inverse color switch.

Bits 7–4: Default value: 1. These four bits determine the background color of the screen. Colors available on the VIC are:

Black	0	Orange	8
White	1	Lt Orange	9
Red	2	Pink	10
Cyan	3	Lt Cyan	11
Purple	4	Lt Purple	12
Green	5	Lt Green	13
Blue	6	Lt Blue	14
Yellow	7	Lt Yellow	15

   In multicolor mode, this color is selected with bit values of binary 00.

To set the upper four bits in this location, you would POKE 36879,(PEEK (36879) AND 15) OR (X*16), where X is a color value from the table above.

Bit 3: Default value: 1. This bit serves as the inverse color switch. When set to one, the background and foreground colors are in their respective places. Setting this to zero, however, inverts that scheme. The foreground color will be used for the background and all the characters are shaded in the background color.

This bit has no effect when multicolor mode is in effect for individual characters.

See locations 37888–38911 ($9400–$97FF), Screen Color Maps, and 646 ($286) for foreground color descriptions.

To set to one: POKE 36879, PEEK (36879) OR 8
To set to zero: POKE 36879, PEEK (36879) AND 247

Table 5-5. Color Codes for Location 36879

Back-ground	BLK	WHT	RED	Border CYAN	PUR	GRN	BLU	YEL	Back-ground
BLK	8	9	10	11	12	13	14	15	BLK
WHT	24	25	26	27	28	29	30	31	WHT
RED	40	41	42	43	44	45	46	47	RED
CYAN	56	57	58	59	60	61	62	63	CYAN
PUR	72	73	74	75	76	77	78	79	PUR
GRN	88	89	90	91	92	93	94	95	GRN
BLUE	104	105	106	107	108	109	110	111	BLUE
YEL	120	121	122	123	124	125	126	127	YEL
ORG	136	137	138	139	140	141	142	143	ORG
L.ORG	152	153	154	155	156	157	158	159	L.ORG
PINK	168	169	170	171	172	173	174	175	PINK
L.CYN	184	185	186	187	188	189	190	191	L.CYN
L.PUR	200	201	202	203	204	205	206	207	L.PUR
L.GRN	216	217	218	219	220	221	222	223	L.GRN
L.BLU	232	233	234	235	236	237	238	239	L.BLU
L.YEL	248	249	250	251	252	253	254	255	L.YEL

Bits 2-0: Default value: 3. These three bits control the border color surrounding the screen.

Available border colors are:

Black 0   Purple 4
White 1   Green 5
Red 2   Blue 6
Cyan 3   Yellow 7

See locations 36864 ($9000) through 36868 ($9004) for additional information.

To set these three bits, you would enter POKE
36879,(PEEK(36879) AND 240) OR X, where X is a color value from
the list above.

If you would like an amber on black screen, for instance, you
would use:

POKE 36879,8:POKE646,2 (actually red)

If you would like a black on amber screen, try:

POKE 36879,136:POKE646,0

36880-37135 $9010-$910F
Future expansion RAM/ROM space.

This area of the VIC-20 contains apparent *reflections* of the VIC
chip registers. These reflections are not reliable and should not be
used. This area is not on RAM or ROM, and it's only because of the
address decoding scheme used that the VIC chip registers *seem* to be
reflected here. This area really is available for future expansion. You
are really accessing the VIC chip registers when POKEing here, and
a PEEK to this area has a good chance of returning erroneous values.
You can use the VIC chip registers directly and avoid many
problems.

If you refer to the article "VIC Memory—The Uncharted Adven-
ture," by David Barron and Michael Kleinert, in *COMPUTE!'s First
Book of VIC*, subtract 16 from the locations listed over 36880. You'll
then have the VIC chips which perform the functions the authors
have noted.

Chapter 6

Versatile Interface Adapters VIA 1 and VIA 2

Versatile Interface Adapters VIA 1 and VIA 2

Location Range: 37136-37151 ($9110-$911F)
6522 Versatile Interface Adapter 1

The 6522 Versatile Interface Adapters (VIAs) provide keyboard, tape, joystick, serial, RS-232, and user port input/output control, as well as RUN/STOP-RESTORE key and IRQ interrupt timing facilities. Interrupts, timing, and input/output are keys to the successful functioning of the VIC-20. Serial to parallel and parallel to serial shift registers are also provided in the VIAs, but are not used on the VIC-20. The IEEE-488 bus of the earlier PET computers has been stripped down to the serial bus on the VIC-20. However, IEEE-488 adapters are available and allow the use of PET peripherals on the VIC-20.

Programming the VIAs is not as simple as for the other chips in the VIC-20. VIAs are complex, and require some study and experimentation before practical use can be made of them. The Kernal uses the VIAs extensively, and you'll need to learn how to work around the Kernal's use of the VIAs. When the function associated with each bit is not being used by the Kernal, you're free to use that bit for your own purposes, within the scope of the architecture of the 6522. These locations should not be used to store user data that is unassociated with 6522 operations.

Nick Hampshire investigates the VIA and how it is used in the VIC-20 in his book *VIC Revealed*. The *VIC-20 Programmer's Reference Guide* includes a description, although only the first VIA is mentioned and no relationship is shown to the actual use by the VIC-20 of the various locations. The schematic in the back of the book provides many insights into the use of the 6522 chips. *Programming the PET/CBM* by Raeto Collin West explains the VIA from the perspective of PIAs (PETs have two PIAs and VIAs). Specification sheets for the 6522 are available from MOS Technology. Rodnay Zaks has explored the 6522 architecture in his 6502 series of books. Another resource is Marvin L. DeJong's *Programming and Interfacing the 6502, with Experiments*; his article "Timing and Counting with the 6522," in the July 1982 issue of *Micro*, is also something you might want to refer to.

Let's get into the specifics of how the VIC-20 uses these VIAs.

The 6522 VIA chip registers are actually located in the VIA chips themselves. No RAM locations correspond to these VIA registers. When POKEing or PEEKing values, you are actually accessing the VIA chips themselves. Special rules for reading and writing bytes may sometimes apply.

Power-on/reset and the RUN/STOP–RESTORE keys cause these values to be reinitialized by the INITVIA* routine.

The First 6522 VIA Chip's Registers

This VIA's IRQ (interrupt request) line is connected to the 6502 IRQ interrupt line. The VIA generates an IRQ interrupt for many reasons, but the most visible is the RESTORE key. This VIA manages most of the pins on the user port/RS-232 port. Additionally, the tape switch and motor, most serial port pins, the light pen/fire button pin, and most of the joystick pins are controlled by this VIA. This VIA offers the most available lines for your use. When the function associated with each bit is not being used by the Kernal, you're free to use that bit for your own purposes.

The serial port pins and their associated bits in this VIA are:

Table 6–1. Serial Port Pins

Port Pin	Port /bit	Line	VIA Use
1	(VIA2,CB1)		serial service request in
2			ground
3			serial attention in, tied to user port 9
	A7	PA7	inverse serial attention out
4	A0	PA0	serial lock in
	(VIA2,CA2)		inverse serial clock out
5	A1	PA1	serial data in
	(VIA2,CB2)		inverse serial data out
6	reset		RESET (ground this to pin 2 for a RESET SWITCH)

The user port pins and their associated bits in this VIA are detailed in Table 6-2.

Table 6–2. User Port Pins

Port Pin	Port /bit	VIA Line	Use	In/ Out	X-Line 3-Line	DB25 Pin	EIA Name
A			protective ground		3,X	1	AA
B		CB1	received data (SIN)	IN	3,X	3	BB
C	B0	PB0	received data (SIN)	IN	3,X	3	BB
D	B1	PB1	request to send (RTS)	OUT	X,3=hi	4	CA
E	B2	PB2	data terminal ready (DTR)	OUT	X,3=hi	20	CD
F	B3	PB3	ring indicator (RI)	IN		22	CE
H	B4	PB4	received line signal (DCD)	IN	X	8	CF
J	B5	PB5	unused				
K	B6	PB6	clear to send (CTS) *	IN	X	5	CB
L	B7	PB7	data set ready (DR)	IN	X	6	CC
M		CB2	transmitted data (Sout)	OUT	3,X	2	BA
N			signal ground		X	7	AB
1			ground				
2			+5 volts DC				
3			reset RESET (ground this to pin 1, A, or 12 to cause a RESET SWITCH function)				
4	A2	PA2	joy 0, also on game port pin 1				
5	A3	PA3	joy 1, also on game port pin 2				
6	A4	PA4	joy 2, also on game port pin 3				
7	A5	PA5	light pen/fire button, also on game port pin 6				
8	A6	PA6	tape sense switch tied to this pin				
9	A7	PA7	inverse serial attention out				
10			9 volts AC				
11			9 volts AC (incorrectly labeled GND in the VIC-20 Programmer's Reference Guide)				
12			ground				

*Because of the coding problem mentioned at 660 ($294), the clear to send missing bit will never be set in the RS-232 status byte at 663 ($297). You can still use this pin by testing this bit yourself.

37136

Table 6-3 details the game port pins and their associated bits in this VIA.

Table 6-3. Game Port Pins

Port Pin	Port /bit	VIA Line	Use
1	A2	PA2	joy 0, also on user port 4
2	A3	PA3	joy 1, also on user port 5
3	A4	PA4	joy 2, also on user port 6
4	(VIA2,PB7)		joy 3
5	(see VIC chip)		potentiometer Y
6	A5	PA5	light pen/fire button
7			+5 volts DC
8			ground
9			(see VIC chip) potentiometer X

The tape port pins and their associated bits in this VIA are displayed below in Table 6-4.

Table 6-4. Tape Port Pins

Port Pin	Port /bit	VIA Line	Use
A-1			ground
B-2			+5 volts DC
C-3	CA2		tape motor
D-4	(see VIA2,CA1)		tape read
E-5	(see VIA2,PB3)		tape write
F-6	A6	PA6	tape switch also on user port 8

The keyboard connector pins are diagrammed in the preface to VIA 2 at location 37152 ($9120).

The individual registers are outlined below, listed by location.

37136　　　　　$9110　　　　　VIAIPB*

Port B I/O register.

This port is used for parallel user port data transfer, in and out. In the VIC-20, the user port is designed for RS-232 protocol handshaking and data transfer.

For the electronics-oriented user: This port is a push-pull type with high impedance input, with a 1 not greater than +2.4 volts; TTL compatible, but not CMOS; supporting direct connection to Darlington transistor switches for relays.

Handshaking control lines CB1 and CB2 are controlled by this VIA port, and when reading or writing these bits, bits 4 and 3 of

location 37149 ($911D) are automatically reset. Location 37148
($911C) is used to control the CB1 and CB2 status.

The RS-232 standard specifies that a logical 1 is a voltage more
negative than −3 volts, and a logical 0 is greater than +3 volts.
Since the VIC-20 uses 0 to 0.8 volts for logical 0 and 2.4 volts and
greater for a logical 1, a TTL level to RS-232 level interface of two
integrated circuit chips is needed to connect a *true* RS-232 device to
the VIC-20. These are commercially available as an RS-232 interface
board providing a DB25 connector for the desired RS-232 device to
plug into. United Microware Industries in Pomona, California, can be
contacted, or you can build your own. See "The Enhanced VIC-20:
Part Four," in the May 1983 issue of *BYTE* magazine for details of
constructing an interface. The VIC Modem has these ICs built in,
and several other modem manufacturers include them on VIC-20
models.

A typical application for this location is an RS-232 printer. For
this type of use, be sure that Data Set Ready (DSR) from the printer
is connected to the VIC-20 Data Terminal Ready (DTR) pin so the
printer can pause the VIC-20 until the printer catches up. X-line
handshaking is needed for this type of printer use. See the article
"VIC RS-232 Printer," by Michael V. Tulloch, in the February 1983
issue of *Micro*, for another approach to the printer hold-off. It's a
good idea to consult your printer manual for interface cable require-
ments, baud rate specifications, control characters, linefeed options,
and switch settings.

The following bits can be used as input or output, depending on
the setting of the data direction register at 37138 ($9112). At power-
on/reset, all these bits are set to the input mode. When an RS-232
device is opened or closed, DTR and RTS (Request To Send) are set
on. The direction during RS-232 use is shown below for each bit.
Bit 7: Data Set Ready (DSR) IN X-line

If set as an output line, you can optionally pulse or invert this
line when VIA timer 1 expires. This can happen continuously when
the *free-running* mode is selected.

See locations 37147 ($911B) and 37138 ($9112).
Bit 6: Clear To Send (CTS) IN X-line

Because of the coding problem mentioned at 660 ($294), the
clear to send missing bit will never be set in bit 4 of 663 ($297). You
may test this bit yourself when using the line.

If this is used as an input line, you can optionally count neg-
ative pulses on this line when VIA timer 2 is in the free-running
mode.

See location 37147 ($911B) and 37138 ($9112).
Bit 5: unused
Bit 4: Data Carrier Detect (DCD) IN X-line

This is sometimes called *received line signal.*

Bit 3: Ring Indicator (RI) IN
Bit 2: Data Terminal Ready (DTR) OUT X-line
  This bit, DTR, is always *on* in three-line mode.
Bit 1: Request To Send (RTS) OUT X-line
  RTS is always *on* in three-line mode.
Bit 0: Received Data Signal (Sin) IN X-line, 3-line

By making or obtaining the proper connector, a second joystick can be plugged into the user port using:

PB1 = joy 3
PB2 = joy 0
PB3 = joy 1
PB4 = joy 2
PB5 = fire

See "Fighter Aces: Add a Second VIC Joystick," by John Parr, in the March 1981 issue of *COMPUTE!*, for details.

37137 $9111   VIAIPAI*

Port A I/O register.

This port is used for serial port and game port data transfer, both in and out. In addition, the tape sense switch is controlled through this port.

Handshaking control lines CA1 and CA2 are managed by this port, and when reading or writing these bits, bits 1 and 0 of location 37149 ($911D) are automatically reset. Location 37148 ($911C) is used to select the CA1 and CA2 status.

A mirror port A register is located at location 37151 ($911F) and can be used instead of this register when you don't want to reset CA1 and CA2 by accessing the register.

For the electronics-oriented user: This port is a pull-up type with a resistive nature. Even in input mode, current is supplied to the pins, and these lines represent a TTL LOAD.

The following bits can be used as input or output, depending on the setting of the data direction register at location 37139 ($9113).
Bit 7: serial attention out.
  Serial attention in is reflected on user port pin 9.
Bit 6: sense tape button down.
  A value of 1 in this bit means that there are no tape buttons down, while a value of 0 means that there are *some* down.
  Note that user port pin 8 is also tied to this line. (Line number) IF (PEEK(37151) AND 64)=1 AND ST=0 THEN (Line number) can be used as a method of determining if all data has been sent on the RS-232 line before closing device number 2. Location 37151 ($911F), bit number 6, is a mirror of this bit.
Bit 5: light pen/fire button.

148

A value of 1 means the light pen switch or the fire button was not pressed.

You can make your program wait for the fire button or light pen switch to be pressed by including WAIT 37137,32,32 in the program. If you'd like your program to wait only until the fire button is *released*, you would use WAIT 37137,32 instead.

Bit 4: joy 2 (left/west).

A value of 1 means this direction was not selected.

Bit 3: joy 1 (down/south).

A value of 1 in this bit signals that this direction was not selected.

Bit 2: joy 0 (up/north).

A 1 in this bit means that this direction was not selected.

Bit 1: serial data in.

Serial data out is reflected in VIA1 CB2.

Bit 0: serial clock in.

Serial clock out is reflected in VIA1 CA2.

The standard joystick reading routine for the VIC in BASIC is shown below. Note that actual line numbers have not been included. You can add them yourself so that you can place this routine anywhere within your own program.

```
POKE 37154,127 : REM set data direction register,
        : REM some keyboard keys ignored.
A=(PEEK(37137) AND 28) OR (PEEK(37152) AND 128)
        : REM combine joystick values into variable A
POKE 37154,255 : REM reestablish keyboard scan
B=PEEK(37137) AND 32
        : REM B=0 if fire button pressed
A=ABS((A-100)/4)-7
        : REM reduce variable A to manageable range of values
ON A GOSUB 100,110,120,,130,140,150,,,,160,170,180
        : REM go do the proper routine

100 PRINT "down-left (south-west)" : RETURN
110 PRINT "up-left (north-west)" : RETURN
120 PRINT "left (west)" : RETURN
130 PRINT "down (south)" : RETURN
140 PRINT "up (north)" : RETURN
150 PRINT "still (center)" : RETURN
160 PRINT "right (east)" : RETURN
170 PRINT "up-right (north-east)" : RETURN
180 PRINT "down-right (south-east)" : RETURN
```

This routine is detailed in the article "The Joystick Connection: Meteor Maze," by Paul L. Bupp and Stephen P. Drop, in *COMPUTE!'s First Book of VIC*.

Additional joystick articles can be found in "Using a Joystick," by David Malmberg, in *COMPUTE!'s First Book of VIC*; "Using Atari Joysticks with Your VIC," by Christopher Flynn, in the June 1982 issue of *COMPUTE!*; and the *VIC-20 Programmer's Reference Guide*, page 246.

37138 $9112 VIAIDDRB*

Data direction register for port B.

Each of these eight bits corresponds to the same-numbered bit in port B. When a bit in this register is set to 0, the corresponding port B bit is used for input. A 1 in a bit of this register indicates an output function in the port B related bit. Port B is located at 37136 ($9110).

Also see 37147 ($911B) bit 1 for port B latch enable bit. If input latching is disabled, any input directed port B bits will, at any time, show the current associated pin status of high or low. Latched mode reads data only when a transition on CB1 line occurs. See location 37148 ($911C).

Power-on/reset and the RUN/STOP–RESTORE keys cause initialization to 00 ($00), all lines input, by the INITVIA* routine.

37139 $9113 VIAIDDRA*

Data direction register for port A.

Each bit in this register corresponds to the same-numbered bit in port A. When one of these bits is set to 0, the corresponding bit in port A is used for input; bits in this location set to 1 signify an output function in the port A related bit. Port A is located at 37137 ($9111) and is also reflected at location 37151 ($911F).

Also see 37147 ($911B) bit 0 for port A latch enable bit. If input latching is disabled, any input directed port A bits will, at any time, show the current associated pin status of high or low. Remember that latched mode reads data only when a transition on CA1 line occurs. See location 37148 ($911C).

Power-on/reset and the RUN/STOP–RESTORE keys reinitialize this register to 128 ($80) (serial attention out for output, rest for input) through the INITVIA* routine.

37140 $9114 VIAITICL*

Timer 1 least significant byte (LSB) of count.

This timer is used for RS-232 and user port transmit/receive and tape write timing.

When setting this byte, the timer 1 latch at location 37142 ($9116) is set with the desired value rather than this location. Refer to location 37143 ($9117) for the sequence of loading timer 1 and

the resulting 6522 actions. Location 37159 ($9127) details the technique used to calculate the value to place in timer 1 to accomplish the desired interval before an interrupt occurs.

37141　　　　　　$9115　　　　　　VIAIT1CH*
Timer 1 most significant byte (MSB) of count.

This location is used with the LSB of count that was stored in location 37140 ($9114). See the description and references at that location. The timer may be started again without an interrupt by storing a new value in this location.

Setting this byte of timer 1 starts the timer. See location 37143 ($9117) for a description of this effect.

The OPENRS* routine calculates the values for timer 1 and timer 2 and stores the result in location 665–666 ($299–29A) until they're needed for timing of the RS-232 session protocol.

37142　　　　　　$9116　　　　　　VIAIT1LL*
Timer 1 low order (LSB) latch byte.

Refer to location 37143 ($9117) for the description of this byte's use.

37143　　　　　　$9117　　　　　　VIAIT1LH*
Timer 1 high order (MSB) latch byte.

This timer is used by the Kernal for tape and RS-232 timing.

The INITVIA* routine disables the PB7 output mode and selects the free-running mode.

The steps and effects involved in using timer 1 are:

● The programmer chooses the time interval desired, whether it is to be a one-shot time interval or free-running mode, and whether PB7 is to be pulsed when the timer expires. If PB7 output is selected, be sure to set bit 7 of 37138 ($9112) to 1, which allows output on PB7.

● The control register at 37147 ($911B) is set to the options desired.
The interrupt enable bits are set or cleared in location 37149 ($911D). See that location for details.

● Store the LSB of the timer period desired in 37140 ($9114).

● Store the MSB of the time period in 37141 ($9115). This causes the VIA to also copy the value into *this* location, and starts the countdown of timer 1. This is preceded by the VIA copying the values in location 37142 ($9116) into location 37140 ($9114), the low order latch into the low order counter. The flag register at address 37149 ($911D), bit 6, is set to 0, and if the PB7 output option was specified, the PB7 line is pulled low to 0.

• The counter in timer 1 counts down at the rate of the system clock. See the description of the clock rate at location 37159 ($9127).

• When the count in timer 1's LSB and MSB reaches zero, the flag in 37149 ($911D) bit 6 is set to 1, an NMI interrupt is signaled to the 6502 if the bits in 37150 ($911E) allow it, and PB7 line is pulled high or inverted. It's inverted if this was opted for in location 37147 ($911B), whether or not the interrupt has been enabled in location 37149 ($911D).

In free-running mode, the values from the latches are reloaded into the counter bytes automatically and the countdown starts anew, generating an interrupt every time the count expires, and generating a waveform on PB7 if its output was selected. A complex waveform can be generated on PB7 by the following steps:

1. Set timer 1 count bytes with value A.
2. Set timer 1 latch bytes with value B.
3. When bit 6 of 37149 ($911D) is set by the VIA, load the next desired value into the latches. This value is loaded into the counters when the current counters expire.
4. Read the timer 1 count LSB to clear the bit 6 flag in 37149 ($911D).
5. Go to step 3.

• The interrupt flag is not cleared until the timer 1 counter LSB is read or the MSB is modified by the program.

37144 $9118   VIAIT2CL*
Timer 2 low order (LSB) counter and LSB latch.

Putting a value in this byte initializes the latch component, while reading this location obtains the value stored here and resets the 37149 ($911D) interrupt flag.

See the more detailed description of this byte at location 37145 ($9119).

37145 $9119   VIAIT2CH*
Timer 2 high order (MSB) counter and MSB latch.

Timer 2 can be used as an interval timer or a pulse counter.

The Kernal uses timer 2 to time the receive side of RS-232 for RS-232 NMI routines. The INITVIA* routine sets timer 2 to be an interval timer. However, bit 5 of location 37147 ($911B) can select the mode of timer 2 as either an interval timer or PB6 pulse counter. Bit 5 of location 37149 ($911D) is the interrupt flag, and bit 5 of location 37150 ($911E) is used to enable the interrupt generated by timer 2.

Putting a value here initializes the latch component and stores the LSB latch into the LSB counter component. The interrupt flag in 37149 ($911D) is cleared, the IRQ line is reset, and the timer is

started. This occurs whether timer 2 is active or not, and so can be used to retrigger it. Reading this location obtains the value stored here.

In order to use this as an interval timer, several things need to be done:

- Set the enable flag in bit 5 of 37149 ($911D), if desired.
- Set the mode for timer 2 in location 37147 ($911B).
- Store the desired value in the LSB byte.
- Store the MSB of the time period in location 37145 ($9119). This begins the countdown of timer 2. First the VIA copies the low order latch into the low order counter, and then the flag register 37149 ($911D) bit 5 is set to 0.
- The count in timer 2 is counted down at the rate of the system clock. See the description of the clock rate at location 37159 ($9127).
- When the count in timer 2's LSB and MSB reaches zero, the flag in 37149 ($911D) bit 5 is set to 1 and an NMI interrupt is signaled to the 6502 if the bits in 37150 ($911E) allow it.
- The counter in timer 2 is allowed to roll over and begin counting down from 65535 ($FFFF). This tells an interrupt routine how long ago the interrupt occurred.

The steps are somewhat different when timer 2 is used as a PB6 pulse countdown counter:

- Set the enable flag in bit 5 of 37149 ($911D), if desired.
- Set the mode for timer 2 in 37147 ($911B).
- Set the data direction register for input on PB6.
- Determine the number of pulses to be counted. Timer 2 counts down from this value and sets the interrupt flag at rollover from 0 to 65535. The NMI interrupt is signaled if the interrupt for this timer is enabled at location 37150 ($911E).
- Store the LSB of the count desired.
- Store the MSB of the count desired. This starts the count.
- When a negative pulse is detected on PB6, the counter will be decremented.
- The interrupt bit can be cleared by reading the LSB of the count; the countdown can be retriggered by writing to the MSB.

Timer 2 can also be used as a shift clock for the shift register at 37146 ($911A). See the description of that location.

37146 $911A   VIAISR*
Shift register for parallel/serial conversion.

The Kernal does not use this shift register.

The shift register provides a mechanism to convert between serial and parallel data and may be used for communicating with devices on the user port of the VIC-20. This VIA does not have a connection from CB1 and CB2 to the serial port, so this shift register

may be used for the user port only.

Serial I/O is slower but simpler than parallel I/O. A 4040 disk drive on an IEEE-488 card installed in the VIC-20 parallel user port is at least eight times as fast as the serial 1540/1541 disk drive.

The shift register can also be used to perform variable frequency pulsing of an output line.

Bit 2 of both the interrupt enable register and the interrupt flag register corresponds to the shift register.

Bits 4–2 of the auxiliary control register at 37147 ($911B) are used to select the shift register mode.

The shift register shifts out bit 7 onto the CB2 line. Storing a value in the shift register starts the shift out. When it's used for input, the bit from CB2 is put into bit 0.

Reading or writing the shift register starts the shift in. To shift in the next byte of data, the value shifted into the register is read. Bits are shifted and stored/sent whenever a pulse is detected on line CB1. The shift register is shifted one bit to the left afterward.

This pulse can be generated in several ways. An external clock (i.e., disk drive) can pulse the CB1 line up to 511.36 kHz. Or the system clock can be converted to a value up to 511.36 kHz and used to clock CB1. The LSB of timer 2 can even be used as a delay between CB1 pulses. A shift counter counts the CB1 pulses and sets the interrupt flag after the eighth shift. In free-running mode, the pulse counter is not used and the bits circulate within the shift register.

See the *VIC-20 Programmer's Reference Guide*, page 232, for details of using the shift register to obtain sound from the user port.

37147 $911B   VIAIACR*
Auxiliary control register.

The auxiliary control register is used to control the options associated with timer 1, timer 2, the shift register, and the way that port A and B are latched with data.

The Kernal initializes this register to 64 ($40) when power-on/reset or the RUN/STOP–RESTORE keys are pressed. See the bit descriptions below marked with an asterisk (*) for the default values.

The bits and possible values in this location are:

Bits 7–6: Timer 1 options
See the description of timer 1 starting at location 37140 ($9114)
00 single interval mode, no PB7 output pulses
*01 free-running mode, no PB7 output pulses
10 single interval mode, PB7 negative pulsed
11 free-running mode, PB7 square wave (invert last pulse)
Bit 5: Timer 2 options
See the description of timer 2 starting at location 37144 ($9118).

*0 single interval timing
    1 countdown incoming PB6 pulses
Bits 4-2: Shift register options
    See the description of the shift register starting at location 37146 ($911A).
    *000 shift register disabled
    001 input data on line CB2 to shift register bit 0, using timer 2 LSB; output clock pulses on line CB1.
    010 input data on line CB2 to shift register bit 0, using system clock; output clock pulses on line CB1.
    011 output data on line CB2 from shift register bit 7, using timer 2 LSB; recirculate bit 7 to bit 0, free running. CB2 can be connected to an amplifier for sound.
    100 output data on line CB2 from shift register bit 7, using timer 2 LSB as delay clock.
    110 output data on line CB2 from shift register bit 7, using system clock
    111 output data on line CB2 from shift register bit 7, using the external clock input on CB1.
Bit 1: Port B latch enable options
    Port B is located at 37136 ($9110).
    *0 Port B to reflect changing values on pins
    1 Port in latch mode. If used for input, port B will show the status of the lines when a CB1 interrupt occurred. Otherwise, the changing status of the lines is not reflected in the port.
Bit 0: Port A latch enable options
    Port A is located at 37137 ($9111) and 37151 ($911F).
    *0 Port A to reflect changing values on pins
    1 Port in latch mode. If used for input, port A will show the status of the lines when a CA1 interrupt occurred. Otherwise, the changing status of the lines is not reflected in the port.

37148 $911C   VIAIPCR*
Peripheral control register for handshaking.
    This byte contains the options for CA1, CA2, CB1, and CB2 lines. The mirror of port A that is at location 37151 ($911F) can be used instead of port A, if you do not want to affect the port A related control line.
    The following Kernal routines modify the Peripheral Control Register (PCR):

• IRQ examines bits 3-1 to determine if the tape motor should be turned off.
• CLOSE for an RS-232 device sets this back to initial values.
• OPENRS for an RS-232 device sets this back to initial values.
• TAPE turns on the tape motor.

155

- TNOFF turns off the tape motor.
- SEROUT1* turns off bit 5 to send a 1 to the device.
- SEROUT0* turns on bit 5 to send a 0 to the device.

The INITIA* Kernal routine sets this byte to the bit values marked with an * below when power-on/reset or the RUN/STOP-RESTORE keys are pressed.

Bits 7–5: CB2 line control. CB2 is used for serial data out, interrupt input, device output, or shift register input or output. In the latter case, these bits are ignored. The interrupt flag register (IFR) is at location 37149 ($911D). Bit 3 of the IFR is used for a CB2 interrupt.

 000 Input mode
This value sets IFR bit 3 on a high to low transition of CB2 and clears the IFR bit 3 when port B is read or written to.

 001 Input mode
These bit values set IFR bit 3 on a high to low transition of CB2, but do not clear IFR bit 3 when port B is read or written to. IFR bit 3 is cleared by writing a 1 to it.

 010 Input mode
This value sets IFR bit 3 on a low to high transition of CB2 and clears IFR bit 3 when port B is read or written to.

 011 Input mode
This sets IFR bit 3 on a low to high transition of CB2, but does not clear IFR bit 3 when port B is read or written to. IFR bit 3 is cleared by writing a 1 to it.

 100 Output mode (handshake)
This sets CB2 line to low when port B is written to. CB2 will be set high when a CB1 transition occurs.

 101 Output mode (pulse)
CB2 line is set low for one cycle when port B is written to with this bit value.

 110 Output mode (manual)
This value sets CB2 to be held low.

 *111 Output mode (manual)
CB2 to be held high when this value is selected. This is the default setting for these three bits.

Bit 4: CB1 line control. CB1 is used to accept an interrupt for received data, as a transition of voltage control line, and as output for the shift register clocking pulses. The fourth bit of the IFR (Interrupt Flag Register) at 37149 ($911D) is used to flag a CB1 interrupt.

 *0 IFR bit 4 is set on a high to low transition of CB1 when the bit is set to this value. This is the default setting of this bit.

 1 IFR bit 4 is set on a low to high transition of CB1.

Bits 3–1: CA2 line control. CA2 is used for tape motor control.

 (By VIA design, it could be used for interrupt input or device output.)

The 0 bit of the IFR at 37149 ($911D) is used to flag a CB1 interrupt.

    000 Input mode

This value sets IFR bit 0 on a high to low transition of CA2 and clears CA2 if port A is read or written to.

    001 Input mode

Sets IFR bit 0 on a high to low transition of CA2, but does not clear it if port A is read or written to. IFR bit 0 is cleared by writing a 1 to it.

    010 Input mode

Setting the three bits to this value sets IFR bit 0 on a low to high transition of CA2 and clears CA2 if port A is read or written to.

    011 Input mode

Sets IFR bit 0 on a low to high transition of CA2, but does not clear it if port A is read or written to. IFR bit 0 is cleared by writing a 1 to it.

    100 Output mode (handshake)

Selecting this value sets CA2 low when port A is read or written to.

    101 Output mode (pulse)

This value outputs a one-cycle pulse of 0 following a read or write of port A.

    110 Output mode (manual)

CA2 is held low with this value.

    *111 Output mode (manual)

    The default setting for these three bits, this holds CA2 high.

Tape motor: 12,14=on; 2,4,6,10=off

To read: 110=on; 111=off

To set: 111=on; 110=never on

111,110=on; any non-11x=off

    A nonzero value in 192 ($C0), which is possible only if some tape buttons are down, prevents any change of tape motor switch—within the default IRQ routine only.

    During tape read or write, location 192 ($C0) is set to nonzero once a tape button has been pressed and will be reset to 0 once the tape action is completed. A 0 value here, which is possible with either some buttons down or no buttons down, allows the tape motor to be turned on within the normal IRQ routine if location 37148 ($911C) has bits 2 and 3 on. This location has no control over tape motor settings outside of the default IRQ handler.

Bit 0: CA1 line control. This is directly wired to the RESTORE key.

    A CA1 line can normally be used to generate an interrupt on a high to low or low to high transition of CA1. Bit 1 of the IFR at 37149 ($911D) is used to flag a CA1 interrupt. As you realize by now, a CA1 interrupt on VIA 1 causes an NMI interrupt on the 6502 chip. The VIA can still generate interrupts while the NMI is processed by the 6502. An NMI interrupt can interrupt a previous one, stacking the previous interrupt's information for later processing. The

expansion port also has a line (pin W) that connects to the 6502 NMI pin.

 0 IFR bit 4 is set on a high to low transition of CB1.

 *1 IFR bit 4 is set on a low to high transition of CB1.

 The effect of placing values in this location, using the statement

POKE 37148,(PEEK(37148)AND 241)OR *nn*

where *nn* is:

0, 2, 4, 6	Stops the motor
8	No change
10	Stops the motor
12, 14	Starts the motor

37149 $911D VIA1IFR*
Interrupt flag register (IFR).

 This register is used to generate a VIA IRQ to the 6502 NMI line when any of bits 1–6 are on. The corresponding bit in the interrupt enable register (IER) at location 37150 ($911E) must be set at 1, signifying an interrupt enabled, and bit 7 of the IER must also be on. This is accomplished by the VIA tying bit 7 of this location to the 6502 NMI line. Bit 7 is turned on when any other bit in this byte is flagged, or turned on. Note that when reading bit 7 of the IER, it will always be presented as a one, but could in fact be a zero. It takes an explicit write to bit 7 to insure its state.

 The conditions that set and clear the interrupt flag bits are reviewed here and are also discussed at location 37148 ($911C), but you can also set or reset a flag by simply setting the appropriate bit in this location.

 When the NMI routine gets control, it tests for the RESTORE key being pressed; if the STOP key is not also pressed, it ignores the RESTORE key. It then checks for and starts any ROM at location 40960 ($A000). If no ROM is present, it checks for a timer 1, timer 2, or RS-232 receive interrupt. If the shift register, CA2, or an RS-232 output interrupt is present, it's not handled by the NMI routines.

 The interrupt flags have the following meaning:

Bit 7: IRQ (NMI)

 This bit is set by any enabled interrupt flag bit being set. By storing a 0 here, you clear all interrupts. It's also reset if all other interrupt flags are currently set to 0.

Bit 6: Timer 1 interrupt

 This bit is set by an expiration of timer 1, and reset when a read of timer 1 LSB or a write of timer 1 MSB takes place.

Bit 5: Timer 2 interrupt

 Set by expiration of timer 2, this bit is reset when a read of timer 2 LSB or a write of timer 2 MSB occurs.

Bit 4: CB1 transition interrupt

Bit 4 is set by the transition of the CB1 line and reset by a read or write on port B.

Bit 3: CB2 transition interrupt

This bit is set by the transition of the CB2 line and reset by a read or write on port B.

Bit 2: Shift register interrupt

Set by eight shifts of the shift register, this bit is then reset by a read or write to the shift register.

Bit 1: CA1 transition interrupt

Bit 1 is set by a transition of the CA1 line and reset by a read or write of port A.

Bit 0: CA2 transition interrupt

This bit is set by a transition of the CA2 line and reset by a read or write of port A.

37150 $911E   VIAIIER*   82
Interrupt enable register (IER).

This byte is used to indicate which interrupt flags in location 37149 ($911D) (the IFR) should cause a VIA IRQ to the 6502 NMI line. The interrupt bits in the IFR are set and reset regardless of the programmer's wish to ignore or detect a particular condition. This byte provides the programmer with a means to control the generation of the VIA IRQ.

Each bit in this byte corresponds to the same-numbered bit in the IFR, except for bit 7, which is a control bit. If bit 7 is set to 1, all the following bits in this byte that contain a 1 enable the corresponding bit in the IFR. For example, if this byte contained binary 1010 0101, then interrupt bits 5, 2, and 0 of the IFR would be enabled. This would cause a VIA IRQ if those IFR bits were turned on.

If bit 7 is set to 0, the remaining bits in this byte that are set to 1 indicate the correponding bits in the IFR that are to be disabled. Binary 0101 1010 placed into this byte, for instance, would cause bits 6, 4, 3, and 1 to be disabled in the IFR.

In practice, you would disable the bits of the IFR that you wish to ignore, then enable those that you wish to use.

A value of binary 0111 1111 ($7F) would disable all interrupts, while binary 1111 1111 ($FF) would enable all.

This byte must be set by POKEing a value into the location; changing an individual bit with an AND or OR will have no effect. Note that when reading bit 7 of the IER, it will always be presented as a 1, but could in fact be a 0. It takes an explicit write to bit 7 to insure its state.

See location 37149 ($911D) for conditions that set the VIA IRQ flags.

The Kernal RS-232 routines test and set this byte extensively. The serial and tape routines, however, leave this byte as initialized.

The Kernal INITVIA* routine sets the IER so that only a RE-STORE key (CA1) interrupt is enabled. The bits and their settings are:

Bit 7: Enable/disable control bit

  0 disable IFR bits corresponding to bits in this byte set to 1

  1 enable IFR bits corresponding to bits in this byte set to 1

  In each of the remaining bits, a 0 indicates that the function is *disabled*, while a 1 signifies that the function is *enabled*.

Bit 6: Timer 1 interrupt

Bit 5: Timer 2 interrupt

Bit 4: CB1 interupt

Bit 3: CB2 interrupt

Bit 2: Shift register interrupt

Bit 1: CA1 (RESTORE key) interrupt

Bit 0: CA2 interrupt

37151 $911F   VIAIPA2*

This is a mirror of port A I/0 register at location 37137 ($9111), except that the CA1 and CA2 control lines are not affected when using this port A reflection. This is described at location 37148 ($911C).

Location Range: 37152–37167 ($9120–$912F)
6522 Versatile Interface Adapter 2

VIA 2 is used by the VIC-20 for keyboard scanning, jiffy interrupt generation, serial service request interrupt detection, tape I/O, and joystick joy 3 reading.

When timing is being performed for the tape drive, the IRQ timing is suspended, thereby causing the STOP key, and the updating of the BASIC variables TI and TI$, to be temporarily ignored.

STOP key scanning is implemented by leaving PB3 active, which is done by setting bit 3 to 0 and all other bits to 1. When a test for the STOP key is called for, PA0 is checked for a value of 0. The routine STOP is used to detect the STOP key and is vectored from location 808 ($328). The BASIC routine NEWSTT calls STOP after each BASIC statement processed during run mode. Location 145 ($91) discusses the saved result of the keyboard STOP key test that senses other keys as a by-product.

Unlike VIA 1, VIA 2 ports are dedicated to the keyboard scan function and only the CA1, CA2, CB1, and CB2 control lines are accessible to the user. These are available through the serial and tape ports. VIA 1 is the better choice for user control of VIA ports, unless you're thinking of attaching to the keyboard connector.

Whereas VIA 1 presents a VIA IRQ to the 6502 as an NMI

(Non-Maskable Interrupt) signal, VIA 2 IRQ use the 6502 IRQ line which may be masked by setting the proper flag in the processor status register (.P). See location 783 ($30F) for details of setting and testing .P.

The Second 6522 VIA Chip's Registers

The 6522 VIA chip registers are actually located in the VIA chips themselves. No RAM corresponds to these VIA registers. When POKEing or PEEKing values, you are actually accessing the VIA chips themselves. Special rules for reading and writing bytes may sometimes apply because of this.

Power-on/reset and the RUN/STOP–RESTORE keys cause these values to be reinitialized by the INITVIA* routine. The initialization value will be shown at each VIA 2 location.

Schematic diagrams of the various VIC-20 ports and the VIA line connections to them are shown in the introduction to VIA 1. VIA 2 lines are included on the diagrams. However, the keyboard connector port is managed by VIA 2, and so is shown below in Table 6-5.

Table 6-5. Keyboard Connector Pins

Port Pin	Port /bit	VIA Line	Use
1			ground
2			(KEY)
3		(VIA1,CA1)	RESTORE key
4			+5 volts
5	B7	PB7	column 7 of keyboard; also joy 3 on game port pin 4
6	B6	PB6	column 6 of keyboard
7	B5	PB5	column 5 of keyboard
8	B4	PB4	column 4 of keyboard
9	B3	PB3	column 3 of keyboard, also tape write on tape port E-5 left on for STOP key scanning
10	B2	PB2	column 2 of keyboard
11	B1	PB1	column 1 of keyboard
12	B0	PB0	column 0 of keyboard
13	A7	PA7	row 7 of keyboard
14	A6	PA6	row 6 of keyboard
15	A5	PA5	row 5 of keyboard
16	A4	PA4	row 4 of keyboard
17	A3	PA3	row 3 of keyboard
18	A2	PA2	row 2 of keyboard
19	A1	PA1	row 1 of keyboard
20	A0	PA0	row 0 of keyboard

Refer to location 197 ($C5) for more information on the way the routine IRQ uses the keyboard matrix. The IRQ routine calls the SCNKEY routine, as you can also, to perform the keyboard scan. Location 145 ($91) STKEY discusses the saved result of the keyboard STOP key test that senses other keys as a by-product.

37152 $9120   VIA2PB*
Port B I/O register.

This port is used for scanning the keyboard columns.

Handshaking control lines CB1 and CB2 are controlled by this port, and when reading or writing these bits, bits 4 and 3 of 37165 ($912D) are automatically reset. Location 37164 ($912C) is used to select the CB1 and CB2 status.

The INITVIA* routine sets this port to output mode in the data direction register at 37154 ($9122). No other routines change that setting.

To scan a particular column, you need to turn off the bit corresponding to that column and turn on all the other bits. Refer to port A at location 37153 ($9121) for more information on the mechanics of keyboard column/row scanning.

This byte normally contains 247 ($F7) (11110111 in binary), except when SCNKEY or UDTIM is examining the keyboard.

The bits' functions are:
Bit 7: Column 7 of keyboard; also joy 3 on game port pin 4
Bit 6: Column 6 of keyboard
Bit 5: Column 5 of keyboard
Bit 4: Column 4 of keyboard
Bit 3: Column 3 of keyboard; also tape write: tape port E-5
Left on (0) for STOP key scanning.
Bit 2: Column 2 of keyboard
Bit 1: Column 1 of keyboard
Bit 0: Column 0 of keyboard

To read the joystick joy 3 switch, you could use:
POKE 37154,127 : REM set data direction register,
     : REM some keyboard keys ignored.
J3=PEEK(37152) AND 128 : REM 0=on; 1=off
POKE 37154,255 : REM restore the data direction

37153 $9121   VIA2PA1*
Port A I/O register.

This port is used for keyboard row scanning.

Handshaking control lines CA1 and CA2 are managed by this port, and when reading or writing these bits, bits 1 and 0 of 37165 ($912D) are automatically reset. Location 37166 ($912C) is used to select the CA1 and CS2 status.

The INITVIA* routine sets this port to input mode in the data direction register at 37155 ($9123). No other routines change that setting.

A mirror port A register is located at 37167 ($912F) and can be used instead of this register when you don't want to reset CA1 and CS2 by accessing the register.

Here's what each bit of this location scans:

Bit 7: Row 7 of keyboard
Bit 6: Row 6 of keyboard
Bit 5: Row 5 of keyboard
Bit 4: Row 4 of keyboard
Bit 3: Row 3 of keyboard
Bit 2: Row 2 of keyboard
Bit 1: Row 1 of keyboard
Bit 0: Row 0 of keyboard

Scanning for keys pressed is a two-step process. First, a *column* of keys is selected by turning on all bits of port B and turning off the bit corresponding to the column to be examined. Column 0 is normally the first column scanned. Then by examining port A, the key pressed is detected by seeing which bit is set to 0. If no keys were pressed in that column, port A would contain 255 ($FF). The next column is selected and the row is again read from port A.

No more than one key per column can be sensed at the same time. However, a key pressed on a different column can be detected, since all eight columns are examined consecutively. The SCNKEY routine performs this scanning of the keyboard, leaving the resulting value in location 203 ($CB). You may call this routine yourself. It is invoked automatically on every IRQ routine entry. The value in 203 ($CB) is stored as an index value into the keyboard decoding tables starting at NORMKEYS* 60510 ($EC5E).

The key values placed in location 203 ($CB) are listed in location 197 ($C5). Refer to the table in that location for each key's value.

The STOP key is represented by a code of 24 in this location. You need to disable the STOP key in a program to see this value. See the vector description at 808 ($328) for details of how to disable the STOP key.

The SCNKEY routine is used to translate the values in location 203 ($CB) into ASCII by picking up the *n*th value in the keyboard decoding tables at NORMKEYS* 60510 ($EC5E), where *n* is the contents of 203 ($CB). SHIFT, Commodore key, and CTRL key flags at location 653 ($28D) are also used to determine the table. The resulting ASCII value is placed in the keyboard buffer at location 631 ($277).

For STOP key testing purposes, the column in port B is always

163

left as 247 ($F7) so that a read of port A can quickly determine if PA0 is 0. The subroutine to perform this test is STOP, which sets the zero flag in .P if the STOP key is pressed. Otherwise, it returns the value of port A in .A, which could be another key in column 3. Location 145 ($91) is used to save the results of the STOP routine returned accumulator. Every other key on the bottom row of the keyboard may be tested for in this location, without doing a GET in BASIC.

Values in location 145 ($91) signify:

255 ($FF)	no key pressed
254 ($FE)	STOP key pressed, STOP routine will find and act on
253 ($FD)	left SHIFT key pressed
251 ($FB)	X key pressed
247 ($F7)	V key pressed
239 ($EF)	N key pressed
223 ($DF)	comma key pressed
191 ($BF)	slash key pressed
127 ($7F)	cursor up key pressed

Table 6–6. Keyboard Scanning Using Port B and Port A of VIA 2

	ROW loaded from $9121 or $912F							
Bit name	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0
Decimal	127	191	223	239	247	251	253	254
Hex	$7F	$BF	$DF	$EF	$F7	$FB	$FD	$FE
Binary	0111	1011	1101	1110	1111	1111	1111	1111
	1111	1111	1111	1111	0111	1011	1101	1110

COLUMN stored in $9120

Bit	Dec	Hex	Binary								
PB7	127	$7F	0111 1111	f7	HOME	-	0	8	6	4	2
PB6	191	$BF	1011 1111	f5	↑	@	O	U	T	E	Q
PB5	223	$DF	1101 1111	f3	=	:	K	H	F	S	Commo
PB4	239	$EF	1110 1111	f1	right SHIFT	.	M	B	C	Z	space
PB3	247	$F7	1111 0111	cursor down	/	,	N	V	X	left SHIFT	STOP
PB2	251	$FB	1111 1011	cursor right	;	L	J	G	D	A	CTRL
PB1	253	$FD	1111 1101	RETURN	*	P	I	Y	R	W	
PB0	254	$FE	1111 1110	DELETE	£	+	9	7	5	3	1

37154 $9122   VIA2DDRB*

Data direction register for port B.

Each of these eight bits corresponds to the same-numbered bit

in port B. When a bit in this location is set to 0, the corresponding port B bit is used for input. A bit set to 1 in this address indicates an output function in the port B related bit. Port B is located at 37152 ($9120).

Power-on/reset and the RUN/STOP–RESTORE keys cause initialization to 255 ($FF), signifying all lines as output, by the INITVIA* routine.

To read the joystick joy 3 switch, you could use the following routine:

POKE 37154,127 : REM set data direction register
    : REM some keyboard keys ignored
J3=PEEK(37152) AND 128 : REM 0=on; 1=off
POKE 37154,255 : REM restore the data direction

37155 $9123   VIA2DDRA*
Data direction register for port A.

Each of the bits in this register has a corresponding bit in port A. When one of these bits is set to 0, the corresponding port A bit is used for input. Setting a bit in this location to 1 indicates an output function in the port A related bit. Port A is located at 37153 ($9121) and is also reflected at 37167 ($912F).

Also see location 37163 ($912B) bit 0 for port A latch enable bit. If input latching is disabled, any input directed port A bits will, at any time, show the current associated pin status of high or low. Latched mode will read data only when a transition on CA1 line occurs. See location 37164 ($912C).

Power-on/reset and the RUN/STOP–RESTORE keys cause initialization to 00 ($00) by the INITVIA* routine.

37156 $9124   VIA2T1CL*
Timer 1 least significant byte (LSB) of count.

This timer is used to generate the IRQ 60 times per second and to time the tape drive read and write functions.

Timer 1 is initialized to 17033 ($4289) by the VIA initialization routine. See location 37159 ($9127) for the method used to determine the value placed in timers which will obtain the time interval required.

When setting this byte, the timer 1 latch at 37142 ($9126) is set with the desired value rather than this location. See location 37143 ($9117) for the sequence of loading timer 1 and the resulting 6522 actions. You can also refer to location 37159 ($9127) for the technique used to calculate the value to place in timer 1 to accomplish the desired interval before an interrupt occurs.

37157 $9125   VIA2T1CH*
Timer 1 most significant byte (MSB) of count.

    This timer is used to generate the IRQ 60 times per second and to time the tape drive read and write functions.

    The tape read and write routines replace the value used here for the IRQ and the vector for the IRQ routine, and use timers 1 and 2 for timing of the tape function. This is why time is lost in the BASIC variable TI and TI$ during tape functions. The timer 1 jiffy IRQ value, along with the associated vector, is replaced by the tape routines when they have finished their task.

    This byte is used with the LSB of the count stored in the timer 1 LSB byte at location 37156 ($9124). See the description and references at that location. The timer may be started again without an interrupt (retriggered) by storing a new value in this location.

    Setting this byte of timer 1 starts the timer running. See location 37143 ($9117) for a description of this effect. You'll need to adjust the addresses mentioned in that location by adding 16 to have the correct locations for VIA 2.

37158 $9126   VIA2T1LL*
Timer 1 low order (LSB) latch byte.

    Timer 1 is used to generate the IRQ 60 times a second and to time the tape drive read and write functions.

    The calculations used to determine the value for timer 1 are explored at location 37159 ($9127).

    See the description of VIA 1 timer 1 latches at 37143 ($9117) for details of how the timer latches are associated with the timer 1 counts. Note the differences in addresses for VIA 1 and 2 locations.

37159 $9127   VIA2T1HL*
Timer 1 high order (MSB) latch byte.

    Timer 1 is initialized to 17033 ($4289) by the VIA initialization routine.

    The tape read and write routines replace the value used here for the IRQ and the vector for the IRQ routine, and use timers 1 and 2 for timing of the tape function, restoring the timer 1 value and the IRQ vector when done.

    In the article "Slow LIST on the VIC-20," by Ken Bowd, in the June 1983 issue of *COMPUTE!*, the techniques described for slowing the LIST output to the screen are actually modifying the VIA 2 timer 1 located at this location. Bowd's article described the modification of location 37879 ($93F7), which is actually a reflection of this VIA timer byte, caused by the address decoding that sees the VIA registers when an empty block of RAM/ROM is accessed.

The idea explored in the article involved changing the IRQ timer MSB latch so that the BASIC LIST command is interrupted by the IRQ routine often enough that LIST output to the screen is slowed, making the program easier to read. By using POKE 37159,0, the cursor becomes a blur because it is flashing so rapidly. When a LIST is done under this condition, it is slowed considerably.

Holding down or locking the SHIFT key causes the lines to be displayed at the rate of one line every one and one-half seconds. In addition, pressing the CTRL key with SHIFT down acts as a pause key. The CTRL key by itself will slow the LIST to one line every 13 seconds. The cursor control keys are a little difficult to control with this technique because they move the cursor so quickly. The STOP key can still be used to break the LIST output, the printer still prints at the normal rate if LIST is directed to it with a CMD statement, and RUN/STOP–RESTORE will reset the VIA timer latch byte back to its original value of 66 ($42).

The jiffy clock is obviously not accurate if the IRQ rate is modified from its original value. This technique of changing the VIA 2 timer 1 high order latch in order to speed up or slow down the IRQ interrupt can be used in other applications as well. The slowing down of the Super Expander cartridge in order to customize its drawing rate for games was suggested in the article. The lower the number put into this location, the more often the IRQ interrupts are called, while a number higher than the original value of 66 causes the interrupts to happen more slowly, which will speed things up. The cursor blink rate, which speeds up as the value is lowered, also indicates the rate of the keyboard scan. When this location's value is changed to 255, the cursor and keyboard keys react at about one-quarter of their normal rate.

Keep in mind that a free-running timer is reloaded from the latches once the interrupt has been flagged in the interrupt flag register. By modifying the latch LSB/MSB, you are changing more than just the current time interval value.

The LSB and MSB counters and latches are related to each other and used in conjunction to accomplish the desired timing/counting function. See location 37143 ($9117) for a description of the relationships. Make sure you adjust the addresses mentioned there by adding 16 for the corresponding VIA 2 locations.

The timers in the VIAs can be used to time an interval between about 225 microseconds and 64 milliseconds. The minimum of about 225 microseconds is due to the overhead involved in processing the interrupt request presented when the timer expires. This range of time intervals may be extended by using a location in memory to count the number of interrupts that have occurred. In a BASIC program, the variables TI and TI$ access the jiffy clock at location 160 ($A0). This jiffy clock is updated by the IRQ routine, and is, in

essence, a software clock that counts the timer 1 expirations.

The 6560 VIC chip includes a clock generator circuit that receives the input from a 14.31818 megahertz (million cycles per second) two-phase oscillator clock. In Europe, the PAL standard oscillator clock is 4.436187 megahertz. This frequency is used to generate the correct TV picture frequency. In the USA, this 14.31818 frequency is divided by the 6560 VIC chip, producing a 1.02272 megahertz clock line (1.108224 megahertz for PAL European) for use as a +5 volt clock for the 6502 and 6522 chips.

To calculate the values for the LSB and MSB of timer 1 or 2 in either VIA, so that you can obtain the desired time interval, use the formula:

(microseconds per interrupt * 1.02272) −2

Let's go through an example of an interrupt every millisecond. One millisecond equals 1000 microseconds, so our formula for this example is:

(1000 * 1.02272)−2

which equals 1021 if we round off. The MSB value would be the integer value of 1021/256, or 3. The LSB value is 1021−(256*3), which equals 253. In hex, our LSB/MSB calculated values are ($FD/$03).

The value used in timer 1 to cause the 60-per-second IRQ is 17033 ($4289), expressed in LSB/MSB as 137/66 ($89/$42). We can determine the amount of time this represents by reversing the formula given, arriving at:

(17033 + 2)/1.02272 = microseconds per interrupt

This is 16,656 microseconds, 16.6 milliseconds, or .0166 seconds.

The maximum value that the LSB/MSB can hold is $FFFF. We can determine the amount of time this represents by again reversing the formula, arriving at:

(65,535 + 2)/1.02272 = microseconds per interrupt

This equals 64,081 microseconds, 64 milliseconds, or .064 seconds.

Keep in mind that a free-running timer is reloaded from the latches once the interrupt has been flagged in the interrupt flag register. By modifying the latch LSB/MSB, you are changing more than just the current time interval value.

The four VIA timers can be used to "wake up" a routine after a certain amount of time, or to time external events such as tape movement, or even how long the lawn sprinklers have been on. You may also want to use a VIA timer as a stopwatch to see how much time an event takes, or as a counter of events. Timer 2 can count pulses coming in on PB6, and a timer 1 expiration can send a pulse out on PB7.

The auxiliary control register at location 37136 ($912B) is used to select the interrupt options associated with the timers. See that location for the available options, and refer to location 37161 ($9129) for a further description of setting the timers. The interrupt enable register at 37166 ($912E) is used to select whether an interrupt is generated for an expiration of timers 1 and 2. Location 37166 ($912E) describes in detail the use of this register. The IRQ vector at 788 ($314) can be changed to point to your routine to count the interrupts that have occurred and do any processing you wish done.

37160 $9128   VIA2T2CL*
Timer 2 low order (LSB) counter and LSB latch.

This timer is used by the Kernal for the timing involved with detecting serial device timeouts, such as failure to respond to a command. It's also used with timer 1 tape read/write timing.

Putting a value here initializes the latch component, while reading this location obtains the value stored here and resets the 37165 ($912D) interrupt flag.

See the further description of this byte at location 37161 ($9129).

37161 $9129   VIA2T2CH*
Timer 2 high order (MSB) counter and MSB latch.

Timer 2 is used by the Kernal for the timing involved with detecting serial device timeouts—for instance, failure to respond to a command—and with timer 1 tape read/write timing.

Bit 5 of location 37163 ($912B) selects the mode of timer 2 as either an interval timer or a PB6 pulse counter. Bit 5 of location 37165 ($912D) is the interrupt flag, and bit 5 of location 37166 ($912E) is used to enable the interrupt generated by timer 2.

Putting a value here initializes the latch component, stores the LSB latch into the LSB counter component, clears the interrupt flag in location 37165 ($912D), resets the IRQ line, and starts the timer. This occurs whether timer 2 is already active or not, and thus can retrigger it. Reading this location obtains the value stored here.

In order to use this as an interval timer, you need to:

- Set the enable flag in bit 5 of 37165 ($912D), if desired.
- Set the mode for timer 2 in 37163 ($912B).
- Store the desired value in the LSB.
- Store the MSB of the timer period in 37161 ($9129). This begins the countdown of timer 2. Before the countdown starts, however, the VIA copies the low order latch into the low order counter, and sets the flag register 37165 ($912D) bit 5 to 0.
- The counter in timer 2 is counted down at the rate of the system clock. See the description of the clock rate at location 37159 ($9127).

• When the count in timer 2's LSB and MSB reaches 0, the flag in 37165 ($912D) bit 5 is set to 1 and an IRQ is signaled to the 6502 if the bits in 37166 ($912E) allow it.
• The counter in timer 2 rolls over and starts counting down from 65535 ($FFFF) so an interrupt routine can tell how long ago the interrupt occurred.

In order to use timer 2 as a PB6 pulse countdown counter, you would:
• Set the enable flag in bit 5 of 37165 ($912D), if desired.
• Set the mode for timer 2 in 37163 ($912B).
• Set the data direction register for input on PB6.
• Determine the number of pulses to be counted. Timer 2 counts down from this value and sets the interrupt flag when the count rolls over from 0 to 65535. The IRQ is signaled if the interrupt for this timer is enabled at location 37166 ($912E).
• Store the LSB of the count desired.
• Store the MSB of the count desired. This starts the count.
• When a negative pulse is detected on PB6, the counter will be decremented.
• The interrupt bit can be cleared by reading the LSB of the count or the countdown can be retriggered by writing to the MSB.

Timer 2 can also be used as a shift clock for the shift register at 37146 ($911A). See the description of that location.

37162 $912A   VIA2SR*
Shift register for parallel/serial conversion.

The Kernal does not use this shift register.
The shift register provides a mechanism to convert between serial and parallel data and can be used for communicating with devices on the user port of the VIC-20. This VIA does not have a connection from CB1 and CB2 to the serial port, so this shift register may be used for the user port only.

Serial I/O is slower but simpler than parallel I/O. A 4040 disk drive on an IEEE-488 card installed in the VIC-20 parallel user port is at least eight times faster than the serial 1540/1541 disk drive.

The shift register can also be used to perform variable frequency pulsing of an output line.

Bit 2 of both the interrupt enable register and the interrupt flag register corresponds to the shift register.

Bits four through two of the auxiliary control register at 37163 ($912B) are used to select the shift register mode.

This register shifts out bit 7 onto the CB2 line. Storing a value in the register starts the shift out. When used for input, the bit from CB2 is put into bit 0.

Reading or writing the shift register starts the shift in. To shift in

the next byte of data, you need to read the value shifted into the register. Bits are shifted and stored/sent whenever a pulse is detected on line CB1. The shift register is shifted one bit to the left afterward. This pulse can be generated in several ways.

An external clock, for instance a disk drive, can pulse the CB1 line up to 511.36 kHz. The system clock can also be converted to a value of up to 511.36 kHz and used to clock CB1. The LSB of timer 2 can be used as a delay between CB1 pulses. A shift counter counts the CB1 pulses and sets the interrupt flag after the eighth shift. In free-running mode, the pulse counter is not used, and the bits circulate within the shift register.

See the *VIC-20 Programmer's Reference Guide*, page 232, for details of using the shift register to obtain sound from the user port.

37163 $912B   VIA2ACR*   64
Auxiliary control register.

The auxiliary control register is used to control the options associated with timer 1, timer 2, the shift register, and the way that port A and B are latched with data.

The Kernal initializes this register to 64 ($40) at power-on/reset or when the RUN/STOP–RESTORE keys are pressed. The bit values marked with an asterisk (*) are those initialized when the computer is first turned on, or after the RUN/STOP–RESTORE keys are pressed.

The bit values and these values' meanings in this register are:

Bits 7–6: Timer 1 options. (See the description of timer 1 starting at location 37156, $9124.)
  00 single interval mode; no PB7 output pulses
 *01 free running mode; no PB7 output pulses
  10 single interval mode; PB7 negative pulses
  11 free running; PB7 square wave, invert of last pulse

Bit 5: Timer 2 options. (See the description of timer 2 starting at location 37160, $9128.)
 *0 single interval timing
  1 countdown incoming PB6 pulses

Bits 4–2: Shift register options. (See the description of the shift register starting at location 37162, $912A.)
 *000 shift register disabled
  001 input data on line CB2 to shift register bit 0 using timer 2 LSB; output clock pulses on line CB1.
  010 input data on line CB2 to shift register bit 0 using system clock; output clock pulses on line CB1.
  011 output data on line CB2 from shift register bit 7 using timer 2 LSB; recirculate bit 7 to bit 0, free running. CB2 can be connected to an amplifier for sound.

100 output data on line CB2 from shift register bit 7 using timer 2 LSB as delay clock.

110 output data on line CB2 from shift register bit 7 using system clock.

111 output data on line CB2 from shift register bit 7 using the external clock input on CB1.

Bit 1: Port B latch enable options. Port B is located at 37152 ($9120).

*0 Port B to reflect changing values on pins.

1 Port in latch mode. If used for input, port B will show the status of the lines when a CB1 interrupt occurs. Otherwise, the changing status of the lines is not reflected in the port.

Bit 0: Port A latch enable options. Port A is located at 37153 ($9121) and 37167 ($912F).

*0 Port A to reflect changing values on pins.

1 Port in latch mode. If used for input, port A will show the status of the lines when a CA1 interrupt occurs. Otherwise, the changing status of the lines is not reflected in the port.

37164 $912C   VIA2PCR*   222

Peripheral control register for handshaking.

This byte contains the options for CA1, CA2, CB1, and CB2 lines. The mirror of port A at location 37167 ($912F) can be used instead of port A, if you do not want to change the related control lines CA1 and CA2.

The Kernal serial device routines modify this location to select the handshaking with the current serial device.

The INITVIA* Kernal routine sets this byte (as marked with an * below) at power-on/reset or when the RUN/STOP–RESTORE keys are pressed.

Each bit and its possible values are:

Bits 7–5: CB2 line control. CB2 is used for serial data out, since it's attached to the serial port pin 5, as well as for interrupt input, device output, or shift register input or output, in which case these bits are ignored. The interrupt flag register (IFR) is at 37165 ($912D). Bit 3 of the IFR is used for a CB2 interrupt flag.

000 Input mode

Set IFR bit 3 on a high to low transition of CB2 and clear IFR bit 3 when port B is read or written to.

001 Input mode

Set IFR bit 3 on a high to low transition of CB2, but do not clear IFR bit 3 when port B is read or written to. IFR bit 3 is cleared by writing a 1 to it.

010 Input mode

Set IFR bit 3 on a low to high transition of CB2 and clear IFR bit 3 when port B is read or written to.

011 Input mode

Set IFR bit 3 on a low to high transition of CB2, but do not clear IFR bit 3 when port B is read or written to. IFR bit 3 is cleared by writing a 1 to it.

100 output mode (handshake)

CB2 line will be set low when port B is written to, and it will be set high when a CB1 transition occurs.

101 Output mode (pulse)

CB2 line will be set low for one cycle when port B is written to.

*110 Output mode (manual)

CB2 line to be held low.

111 Output mode (manual)

CB2 line to be held high.

Bit 4: CB1 line control. CB1 is used to accept an interrupt for received data, as a transition of voltage control line, and as an output for the shift register clocking pulses. This control line is attached to the serial port service request in pin. The fourth bit of the IFR (Interrupt Flag Register) at location 37165 ($912D) is used to flag a CB1 interrupt.

0 IFR bit 4 is set on a high to low transition of CB1.

*1 IFR bit 4 is set on a low to high transition of CB1.

Bits 3-1: CA2 line control. CA2 is used for the serial clock output line and is attached to serial port pin 4. By VIA design, it could be used for interrupt input or device output. Bit 0 of the IFR at location 37149 ($911D) is used to flag a CB1 interrupt.

000 Input mode

Set IFR bit 0 on a high to low transition of CA2, and clear this bit if port A is read or written to.

001 Input mode

Set IFR bit 0 on a high to low transition of CA2, but do not clear it if port A is read or written to. IFR bit 0 is cleared by writing a 1 to it.

010 Input mode

Set IFR bit 0 on a low to high transition of CA2, and clear the bit if port A is read or written to.

011 Input mode

Set IFR bit 0 on a low to high transition of CA2, but do not clear it if port A is read or written to. IFR bit 0 is cleared by writing a 1 to it.

100 Output mode (handshake)

Set CA2 low when port A is read or written to.

101 Output mode (pulse)

Output a one cycle pulse of 0 following a read or write of port A.

110 Output mode (manual)

CA2 is held low.

*111 Output mode (manual)

CA2 is held high.

37165

Bit 0: CA1 line control. This bit is used for tape reading in the VIC-20. It is attached to the tape port pin D-4.

A CA1 line can normally be used to generate an interrupt on a high to low, or low to high transition of CA1. Bit 1 of the IFR at location 37149 ($911D) is used to flag a CA1 interrupt.

The expansion port also has a line (pin W) that connects to the 6502 NMI pin.

  *0 IFR bit 4 is set on a high to low transition of CB1.

  1 IFR bit 4 is set on a low to high transition of CB1.

37165 $912D   VIA2IFR*

Interrupt flag register.

This register is used to generate a VIA IRQ to the 6502 IRQ line when any of bits 1–6 are on and the corresponding bit in 37166 ($912E), the IER (Interrupt Enable Register) is set to 1, and bit 7 of the IER is also on.

This is accomplished by the VIA tying bit 7 of this location to the 6502 IRQ line. Bit 7 is turned on when any other bit in this byte is flagged, or turned on. Note when reading bit 7 of the IER that it will always be presented as a 1, but could in fact be a 0. It takes an explicit write to bit 7 to insure its state.

The conditions that set and clear the interrupt flag bits are received here and were discussed at location 37164 ($912C), but you may also set or reset a flag by simply setting the appropriate bit in this location.

The interrupt flags in this register have the following meanings:

Bit 7: IRQ occurred

This bit is set by any enabled interrupt flag bit being set. By storing a 0 here, you clear all interrupts. It is also reset if all other interrupt flags are currently 0's.

Bit 6: Timer 1 interrupt

This bit is set by an expiration of timer 1, and reset when a read of timer 1 LSB or a write of timer 1 MSB takes place.

Bit 5: Timer 2 interrupt

Set by expiration of timer 2, this bit is reset when a read of timer 2's LSB or a write of timer 2's MSB occurs.

Bit 4: CB1 transition interrupt

This bit is set by the transition of the CB1 line and reset by a read or write on port B.

Bit 3: CB2 transition interrupt

Set by the transition of the CB2 line, bit 3 is reset by a read or write on port B.

Bit 2: Shift register interrupt

This bit is set by eight shifts of the shift register and reset by a read or write to the shift register

Bit 1: CA1 transition interrupt

174

Bit 1 is set by a transition of the CA1 line, while it is reset by a read or write of port A.

Bit 0: CA2 transition interrupt

This bit is set by a transition of the CA2 line and reset by a read or write of port A.

37166 $912E   VIA2IER*

Interrupt enable register (IER).

This byte is used to indicate which interrupt flags in location 37165 ($912D) (IFR) should cause a VIA IRQ to the 6502 IRQ line. The interrupt bits in the IFR are set and reset regardless of the programmer's wish to ignore or detect a particular condition. This byte provides you with a means to enable/disable the generation of the VIA IRQ.

The Kernal serial and tape routines set this byte extensively to accomplish their appointed tasks.

The Kernal INITVIA* routine sets the IER so that only a timer one (IRQ) interrupt is enabled. Its value at initialization is then 192 ($C0).

Each bit in this byte corresponds to the same-numbered bit in the IFR, except for bit 7 which is a control bit. If bit 7 is a 1, all the following bits in this byte that contain a 1 enable the corresponding bit in the IFR. For example, if this byte contained binary 1010 0101, then interrupt bits 5, 2, and 0 of the IFR would be enabled and would cause a VIA IRQ if those IFR bits were turned on.

If bit 7 is set to 0, the remaining bits in this byte that are set to 0 indicate those corresponding bits in the IFR that are to be disabled. Binary 0101 1010 placed into this byte would cause bits 6, 4, 3, and 1 to be disabled in the IFR.

In practice, you would disable the bits of the IFR that you wish to ignore, then enable those that you wish to use.

A value of binary 0111 1111 ($7F) would disable all interrupts, while binary 1111 1111 ($FF) would enable all.

The whole byte must be set at once by POKEing a value into this location. Changing a particular bit individually with an AND or OR will have no effect. Note that when reading bit 7 of the IER, it will always be presented as a 1, but could in fact be a 0. It takes an explicit write to bit 7 to insure its state.

See location 37165 ($912D) for conditions that set the VIA IRQ flags.

The individual bits of this register, along with each bit's function, are:

Bit 7: Enable/disable control bit

   0 Disable IFR bits corresponding to 1 bits in this byte.

   *1 Enable IFR bits corresponding to 1 bits in this byte.

Bit 6: Timer 1 interrupt
    0 disabled
    *1 enabled
Bit 5: Timer 2 interrupt
    The remaining bits of this byte are all initialized at 0, or set as
disabled. A 1 in any of the following bits enables that function.
Bit 4: CB1 interrupt
Bit 3: CB2 interrupt
Bit 2: Shift register interrupt
Bit 1: CAl (tape I/O) interrupt
Bit 0: CA2 interrupt

37167 $912F   VIA2PA2*

    This register is a mirror of port A I/O register at 37153 ($9121),
except that the CA1 and CA2 control lines are not affected when
you use this reflection.

Chapter 7

Input/Output Expansion Blocks and Screen Color Map

Input / Output Expansion Blocks and Screen Color Map

Location Range: 37168-37887 ($9130-$93FF)
Unused Input/Output Expansion Block 0

This block of 719 bytes can be used for future expansion RAM/ROM space.

This area of the VIC-20 contains apparent reflections of the VIA chip registers. These reflections are not reliable and should not be used. This area is not on RAM or ROM, and it's only because of the address decoding scheme used that VIA chip registers even *seem* to be reflected here. This area really is available for future expansion and should not be used in the absence of that expansion.

You're really accessing the VIA chip registers when POKEing here, and a PEEK to this area has a good chance of returning incorrect values. Exhaustive testing with this area has proven its use to be extremely unreliable and it should be avoided. You can use the VIA chip registers directly and avoid problems.

Location Range: 37888-38911 ($9400-$97FF)
Screen Color Maps

37888-38399   **$9400-$95FF**   **COLORMAPS***
(handy location)

Screen color map (8K+ expanded VIC-20).

Bits 4-7 = not there
Bit 3 = multicolor if set to 1 or normal if set to 0
Bits 0-2 = foreground color value 0-7

The VIC-20 contains two RAM color maps, the first at this location and the second at location 38400-38911 ($9600-$97FF). Both are used in exactly the same way. Which map is to be used is based on the amount of expansion memory added to the VIC-20 and is resolved during the power-on/reset routines. If the VIC-20 is unexpanded or has only a 3K expansion, the color map at 38400

($9600) is used. Otherwise, this color map is selected.

Location 36866 ($9002) can be examined to see which color map is currently being used: If bit 7 is off, the color map used starts at location 37888; otherwise, it starts at 38400. To see which color map (CM) is used, you can enter the following line:

CM=37888+(4*(PEEK(36866) AND 128))

where CM is the beginning of the color map selected.

If you redefine the default screen map size, location, bitmap the screen, or define alternate screens, your use of the color map will need to be adjusted to correspond. See location 36869 ($9005) and Appendix E for details of those color map adjustments.

Bits 3–1. Each color map is used by the 6560 VIC chip as a group of 506 bytes, ignoring the remaining six bytes in this area. In this 506 byte-block, each byte's bits 2–0 contain a foreground color number that is used to color any pixels that are turned *on* in that correspond-ing position of the screen. The background color selected (see loca-tion 36879, $900F, VIC chip register) is used to color the pixels that are turned *off* in the same character position.

Foreground color codes on the VIC are:
Black 0
White 1
Red 2
Cyan 3
Purple 4
Green 5
Blue 6
Yellow 7

The inverse color bit at location 36879 ($900F) can be used to reverse the foreground and background colors, making all the characters the background color, and the area behind them colored by the color map contents.

When double-sized characters (8 x 16) are selected by setting the bit at location 36867 ($9003), each position of the color map will correspond to a double-sized character. In this case, the color map need be only 253 characters long.

Bit 3 of the color map byte indicates whether the character is to be displayed in normal high-resolution mode (if the bit is 0) or in multicolor mode (if the bit is set to 1). If multicolor mode is selected for a character, the ones and zeros in that character's eight-byte pixel map determine the colors used.

Every bit-pair in the pixel map for that character represents a color code to be used, not the actual color number. Take a look at Figure 7-1 for an example.

Multicolor and normal high resolution may both be used on the

screen at the same time. Location 32768 ($8000) describes the pixel maps of characters.

Figure 7—1. Multicolor Bit Settings

Bits	Color chosen	Location set	Example PIXEL MAP	COLORS
00	BackGround color	36879	11 11 11 11	AU AU AU AU
01	BOrder color	36879	01 00 00 11	BO BG BG AU
10	ForeGround color	Color Map	01 00 00 11	BO BG BG AU
11	AUxiliary color	36878	01 00 00 11	BO BG BG AU
			01 10 10 11	BO FG FG AU
			01 10 10 11	BO FG FG AU
			01 10 10 11	BO FG FG AU
			01 01 01 11	BO BO BO AU
		Bit number	76 54 32 10	

See location 36879 ($900F) for a chart for background and border colors, and 36878 ($900E) for the valid auxiliary colors.

Normally, you would choose the multicolor mode only when you have defined your own custom character set. See 36869 ($9005).

By using the multicolor mode, half the possible character-width resolution is lost because there are only four double bits across in each row of the character, although it is still eight bits high. But the color effects possible can create the illusion of a greater resolution.

The RAM chip that the color maps are located in is a four bit by 1K RAM chip. This means that each byte is actually made up of only the four low-order bits. The four high-order bits of each byte only appear to be present and are not usable in any way.

See the screen map description at location 4096 ($1000) for formulas that you can use to calculate the byte of the color map that corresponds to a particular byte or bit in the screen map.

The foreground color code is kept in location 646 ($286) and can be either placed there directly or selected by the CTRL key and one of the color keys. After the VIC-20 finishes power-on/reset, the foreground color has been set to blue, or color code 6. When you type a character on the keyboard, or PRINT it from a BASIC program, the Kernal causes the color code in 646 ($286) to be placed at the appropriate color map location for the character's screen position.

If all the color codes are left as set by the VIC-20, a blue character will appear on a white background. When you clear the screen with the CLR key, the screen is filled with spaces and the color map set to all white. These white color codes placed in the color map by the Kernal when clearing the screen have little effect. This is because the Kernal also fills the screen map with the space character which, because it has no pixels turned on, causes the entire screen to be the background color that you selected.

181

If you have selected a background color other than white and now place a character in the screen map, it will display in white on your selected background color, unless you also place a color code in the corresponding color map location. If you select a foreground color that is the same as the background color, the characters typed will be on the screen, but you won't be able to see them. This can be a handy trick at times. For instance, you could make a whole screen of information suddenly appear by filling the color map with a contrasting foreground color code.

Other locations related to the screen maps are:

201 ($C9) Current logical line, column of cursor. A summation of the page 0/1 locations used by the Kernal screen editor and other routines is also included at this location.
209 ($D1) Current screen map line
211 ($D3) Column number that the cursor is on
217 ($D9) Screen line link table
243 ($F3) Pointer to start of line in color RAM
243 ($F3) Address of the current line in color map
647 ($287) Original color under cursor
648 ($288) Screen map page number
780 ($30C) Plotting and color setting example

The routine COLORSET* uses the table COLORTBL* to find the appropriate color code to store in this location when the CTRL and color keys are pressed.

38400-38911 $9600-$97FF
Screen color map (unexpanded or 3K expanded VIC-20).

Please see location 37888–38399 ($9400–$95FF) for a complete description of the use of both VIC-20 color maps.

Location Range: 38912-39935 ($9800-$9BFF)
Unused Input/Output Expansion Block 2

This 4096-byte area, which can be used for future RAM or ROM expansion of the VIC-20, contains apparent reflections of other areas. These reflections are not reliable and should not be used. This area is available for future expansion and should not be used in the absence of that expansion.

Location Range: 39936-40959 ($9C00-$9FFF)
Unused Input/Output Expansion Block 3

A 4096-byte area, this contains reflections of other areas. These reflections are not reliable and should not be used. In the absence of any expansion in this block, this should not be used.

Location Range: 40960-49151 ($A000-$BFFF)
8K Expansion Block 4

40960-49151　　　$A000-$BFFF　　RAMBLK4*
8K RAM expansion block 4.

This block is primarily used for autostart ROM cartridges such as games and other cartridge-based software. However, it may also be used for user RAM expansion, even though BASIC will never see this expansion since it's not contiguous with other RAM. The screen map and character map should *not* be placed in expansion RAM. The ROM or RAM at this location of memory is not required to be autostarted. The following items must be present to activate the autostart code in the Kernal:

40960 ($A000) Vector: power-on/reset routines
40962 ($A002) Vector: NMI (RESTORE key) routines
40964 ($A004) Data: $41 30 C3 C2 CD
　　　　　　　　　　　　A　0　C　B　M

with the high order bit *on* in the last three characters. Since the VIC-20 hasn't finished its initialization routines when the cartridge is autostarted, the following routines should be JSRed to:

$FD8D RAM test
$FD52 set vectors
$FDF9 initialize I/O, CLI
$E518 initialize screen

A BASIC program can be in the ROM cartridge. An ML routine is used to initialize BASIC, modify the pointer to the beginning of the ROM BASIC program (43, $2B), and place RUN in the keyboard buffer.

See the article "RAM/ROM on the VIC for $20," in the December 1982 issue of *Commander* for a description of how to modify RAM to appear as ROM when needed.

At power-on/reset, the START routine at location 64802 ($FD22) initializes the stack pointer, disables interrupts, and calls the CHKAUTO* routine to check for an autostart ROM. If the A0CBM characters were found by that routine, a JMP ($A000) is performed, starting the ROM software.

The NMI routine will also check for an autostart ROM if the RESTORE key has been pressed and will JMP ($A002) if the A0CBM characters are found.

Turn the Commodore 8K and 16K expansion boards' DIP switch 1 *on* for this block.

Chapter 8

BASIC ROM

BASIC ROM

Location Range: 49152-57343 ($C000-$DFFF)
8K BASIC ROM

This area contains the routines that comprise the BASIC interpreter. BASIC is an integral part of the VIC-20 and is started and given control automatically at power-on/reset unless an autostart cartridge is plugged in. When LOAD is entered from the keyboard, it is BASIC that calls for this action to be performed by the Kernal. To allow ML programs to run in the VIC-20, BASIC is inevitably used to modify its own range of memory and to load the ML instructions.

Even if a machine language monitor is used, SYS (a BASIC word) is used to start the monitor. Since BASIC is always present in the VIC-20, its routines for numeric and string manipulation, mathematical functions, and I/O can be used to save you from having to redesign the same procedures, or at the very least can serve as a model for your own ML routines.

If you're comparing the VIC-20 version of BASIC to PET BASIC, look at the PET's BASIC 2.0. The low memory work areas are a bit different, but the BASIC routines are quite similar in purpose and location. However, BASIC 2.0 does not have the serial, color, sound, and RS-232 routines that VIC-20 BASIC contains. The BASIC in the Commodore 64 is functionally identical to that in the VIC-20, although split into two sections.

The differences between the two machines are most noticeable in the Kernal rather than BASIC. BASIC 4.0 for the PET includes disk commands not in VIC-20 BASIC, and PET BASIC 1.0 has a different low memory pointer structure. BASIC 1.0 allows spaces within a keyword (such as IN PUT), while BASIC 2.0 does not.

As you become more familiar with the internal structure of the VIC-20, you'll become adept at translating the bulk of material published for the PET into its equivalent on the VIC-20. *Programming the PET/CBM*, by Raeto Collin West, published by COMPUTE! Books in 1982, is a storehouse of PET/CBM information. Not for the beginner, this is perhaps the most complete volume of information available for Commodore products.

BASIC is actually longer than 8K and spills over into the next 8K ROM where the Kernal routines are located.

In examining the routines in BASIC, be aware that any routine can be entered or jumped out of at virtually any point. The routine descriptions below are individualized by the entry and exit points that are most often utilized. A routine to perform a given function

may in fact be entered at a point that causes what is almost a different function to be performed. BASIC is very tricky and likes to use an existing routine or part of an existing routine if possible, rather than having to include the same instructions at two locations. Many levels of JSRs, stack manipulation of the return address, and other tricky techniques may be confusing when you're first examining a routine.

It's typical to use one routine to set up parameters for an existing routine to process. The descriptions below attempt to identify the fact that another routine is called to perform the task attributed to a particular routine. They also indicate places where BASIC *falls through,* or doesn't exit at the end of a routine, but instead continues on to the next sequential routine.

49152 $C000   COLDST*
Vector to the routine for the cold start of BASIC, 58232 ($E378).

This vector is used to start BASIC at the end of the system's power-on/reset routine at 64802 ($FD22) START. A JMP ($C000) is performed as the last instruction of that routine. Any autostart cartridge would cause the START routine to branch to it, rather than to BASIC via this vector.

See the START routine for details of the power-on/reset functions performed.

See location 58232 ($E378) for the activities performed during the cold start of BASIC.

49154 $C002   WARMST*
Vector to the routine to the warm start of BASIC, 58471 ($E467).

When the RUN/STOP–RESTORE keys are both pressed, the BREAK* routine at 65234 ($FED2) uses this vector, via a JMP ($C002), to go to the WARMBAS* routine after it has completed its duties.

See the BREAK* routine at location 65234 ($FED2) for details of the RUN/STOP–RESTORE and ML BRK instruction functions performed.

See 58471 ($E467) WARMBAS* for the activities performed during the warm start of BASIC.

49156 $C004   CBMBASIC*
CBMBASIC characters.

49164 $C00C   STMDSP
(handy location)
Keyword dispatch vector table, in token order.

This area contains vectors to the routines that handle each

BASIC *keyword*. Functions and math operation vectors are in separate tables at locations 49234 ($C052) and 49280 ($C080), while the BASIC words themselves are in a table at location 49310 ($C09E). The vectors in this table point one byte before the actual routine, so add one to the vectors in memory. This minus-one vector is used so that the address may be placed on the stack, and so that an RTS instruction will add one to the address on the stack and jump three. The RTS is issued at the end of CHARGET.

This table is used by the routine at 51172 ($C7E4) that reads and executes the next BASIC statement.

You can use this table to locate the routine that processes BASIC keywords, then disassemble the routine to examine its methods and dependencies.

The following table shows the contents of this area, *except* that the LSB/MSB has been reversed, and that one has been added to the routine address to reflect its true location.

Table 8—1. BASIC Keyword Handler Vector Table

BASIC Keyword	Vector at Dec	Hex	Routine Dec	Hex
END	49164	$C00C	51249	$C831
FOR	49166	$C00E	51010	$C742
NEXT	49168	$C010	52510	$CD1E
DATA	49170	$C012	51448	$C8F8
INPUT#	49172	$C014	52133	$CBA5
INPUT	49174	$C016	52159	$CBBF
DIM	49176	$C018	53377	$D081
READ	49178	$C01A	52230	$CC06
LET	49180	$C01C	51621	$C9A5
GOTO	49182	$C01E	51360	$C8A0
RUN	49184	$C020	51313	$C871
IF	49186	$C022	51496	$C928
RESTORE	49188	$C024	51229	$C81D
GOSUB	49190	$C026	51331	$C883
RETURN	49192	$C028	51410	$C8D2
REM	49194	$C02A	51515	$C93B
STOP	49196	$C02C	51247	$C82F
ON	49198	$C02E	51531	$C94B
WAIT	49200	$C030	55341	$D82D
LOAD	49202	$C032	57701	$E165
SAVE	49204	$C034	57683	$E153
VERIFY	49206	$C036	57698	$E162
DEF	49208	$C038	54195	$D3B3
POKE	49210	$C03A	55332	$D824
PRINT#	49212	$C03C	51840	$CA80

BASIC Keyword	Vector at Dec Hex	Routine Dec Hex
PRINT	49214 $C03E	51872 $CAA0
CONT	49216 $C040	51287 $C857
LIST	49218 $C042	50844 $C69C
CLR	49220 $C044	50782 $C65E
CMD	49222 $C046	51846 $CA86
SYS	49224 $C048	57639 $E127
OPEN	49226 $C04A	57787 $E1BB
CLOSE	49228 $C04C	57796 $E1C4
GET	49230 $C04E	52091 $CB7B
NEW	49232 $C050	50754 $C642

The following keywords follow the dispatchable keywords in the keyword table at 49310 ($C09E). They are not in the keyword dispatch vector table since they never begin a BASIC statement.

FN
NOT
SPC
STEP
TAB
THEN
TO

49234 $C052   FUNDSP
Function dispatch vector table, in token order.

This area contains vectors to the routines that handle each BASIC *function*. Keywords and math operation vectors are in separate tables at locations 49164 ($C00C) and 49280 ($C080), while the BASIC words themselves are in a table at location 49310 ($C09E).

Functions are defined as those BASIC words that are followed by parentheses. The expression within the parentheses is resolved before the function is called.

Notice that the vector for USR is location 0000, the JMP opcode and vector that the user set for the ML routine. The expression evaluation routines beginning at 52638 ($CD9E) uses this table to set the jump vector at location 84–86 ($54–56) to the routine for the needed function. A JSR ($0054) is then done to the routine, equivalent to a GOSUB in BASIC.

You can use this table to locate the routine that processes BASIC functions, then disassemble the routine to examine its methods.

The following table shows the contents of this area, *except* that the LSB/MSB has been reversed.

Table 8—2. BASIC Function Handler Vector Table

BASIC Function	Vector at Dec Hex	Routine Dec Hex
SGN	49234 $C052	56377 $DC39
INT	49236 $C054	56524 $DCCC
ABS	49238 $C056	56408 $DC58
USR	49240 $C058	00000 $0000
FRE	49242 $C05A	54141 $D37D
POS	49244 $C05C	54174 $D39E
SQR	49246 $C05E	57201 $DF71
RND	49248 $C060	57492 $E094
LOG	49250 $C062	55786 $D9EA
EXP	49252 $C064	57325 $DFED
COS	49254 $C066	57953 $E261
SIN	49256 $C068	57960 $E268
TAN	49258 $C06A	58033 $E2B1
ATN	49260 $C06C	58123 $E30B
PEEK	49262 $C06E	55309 $D80D
LEN	49264 $C060	55164 $D77C
STR	49266 $C072	54373 $D465
VAL	49268 $C074	55213 $D7AD
ASC	49270 $C076	55179 $D78B
CHR	49272 $C078	55020 $D6EC
LEFT	49274 $C07A	55040 $D700
RIGHT	49276 $C07C	55084 $D72C
MID	49278 $C07E	55095 $D737

49280 $C080   OPTAB
Math operation dispatch vector table, in token order.

This area contains vectors to the routines that handle BASIC *math operations*. Keywords and function vectors are in separate tables at locations 49164 ($C00C) and 49234 ($C052), while the BASIC words themselves are in a table at location 49310 ($C09E).

The math operation vectors are accompanied by a byte indicating the order of precedence for that math operation. Those with higher precedence are performed before those with lower precedence. When two operations with equal precedence are encountered on the same line, they're performed in order from left to right.

Order of precedence of expression evaluation
1. Formulas enclosed in parentheses
2. Exponentiation (↑)
3. Negation ($-xyz$, where xyz is an expression)
4. Multiplication and division
5. Addition and subtraction

6. Relational tests: =, <>, <, >, <=, => have equal precedence
7. NOT logical and bit operations
8. AND logical and bit operations
9. OR logical and bit operations

The formula evaluation routine FRMEVL at 52638 ($CD9E) uses this table to find the routine to process the math operation and to determine the order of precedence within the expression. FRMEVAL sets the jump vector at location 34–35 ($22–23) to the routine for the math operation. A JMP ($0022) is then done to the routine, equivalent to a GOTO in BASIC. The vectors in this table point one byte before the actual routine, so add one to the vectors in memory.

You can use this table to locate the routine that processes BASIC math operations then disassemble the routine to examine it.

The following table shows the contents of this area, *except* that the LSB/MSB has been reversed. One has been added to the routine address to reflect its true location.

Table 8—3. BASIC Math Operation Handler Vector Table

BASIC Operation	Precedence: largest first	Vector at Dec Hex	Routine Dec Hex
+	121 $79	49280 $C080	55402 $D86A
−	121 $79	49283 $C083	55379 $D853
*	123 $7B	49286 $C086	55851 $DA2B
/	123 $7B	49289 $C089	56082 $DB12
uparrow	127 $7F	49292 $C08C	57211 $DF7B
AND	80 $50	49295 $C08F	53225 $CFE9
OR	70 $46	49298 $C092	53222 $CFE6
mondatic-	125 $7D	49301 $C095	57268 $DFB4
NOT	90 $5A	49304 $C098	52948 $CED4
<=>	100 $64	49307 $C09B	53270 $D016

49310 $C09E   RESLST
(handy location)

BASIC keyword table in token number order.

The complete vocabulary of BASIC keywords, functions, and math operators, minus the PI symbol, are stored here in token number order. Each word ends with the high order bit on, a value of 128 ($80) added to the value for the ASCII character. The table is ended by a byte containing zero.

This table is used to tokenize the BASIC words when they are entered in direct mode or added or changed in edit mode. The routine that does this tokenization is at 50553 ($C579). When the BASIC program is LISTed, the routine at 50970 ($C71A) uses this

table to detokenize the BASIC words.

Token number order corresponds to the order of words listed in the following three tables. For instance, the last word in the first table comes before the first word in the second table. Note the change of order in the tables; it differs from their order in memory. You can also refer to Appendix C, which lists the tokens in number order. Along with the three tables, there are two other elements in the token number order; they follow the tables in this order.

- 49164 ($C00C) STMDSP Keywords

The keywords FN, NOT, SPC, STEP, TAB, THEN, and TO follow the dispatchable keywords in the keyword table, since they never begin a BASIC statement. Yet these keywords precede the first words in the next table.

- 49280 ($C080) OPTAB Math operations
- 49234 ($C052) FUNDSP Functions
- GO
- 0 as end of table marker, causing SYNTAX ERROR message if the token wasn't found in the table.

The word GO is added to allow GO TO to be considered a valid word. The routine GONE, which reads and executes the next BASIC statement, includes instructions to cause GO TO to use the same routine as GOTO.

If you know that you want the word that corresponds to a given token number (TN for example), the following routine will set X$ to the corresponding word. The token number must be between 128 and 203:

Program 8—1. Token Number to Token Word

```
10 OPEN4,4
15 Y=0
20 X = 49310 : A = 49310
25 TN=139
30 Z =TN - 127
40 IF PEEK(X) > 127 THEN Y = Y+1 : E = X : B = A :
   A = E+1 :IFY=ZTHEN55
45 X = X + 1
50 GOTO 40
55 X$="" : FOR X = B TO E
60 X$ = X$+CHR$(PEEK(X) AND 127)
65 NEXT: PRINT#4,TN,X$: PRINT#4: CLOSE 4
```

Use OPEN 4,3 for screen display, rather than printing it out.

Lines 15–20 set values for Y, X, and A. Respectively, these are initialized as the word counter, and as the start of the keyword table. Line 25 sets TN (Token Number) to 139 for this demonstration. You can use any number greater than 127 and less than 204.

When the end of the word is found, line 40 counts the word, saves it, ends it, begins again, and then sets the next beginning. X is incremented in line 45 to look for the next byte of the table, while line 50 simply loops the program back to line 40 until the word is found. The range of the word in the table is set in line 55, and then line 60 adds the character to X$, minus any high order bit.

By changing a few lines, you can print the entire table out in token number order:

```
25 FOR TN=128 TO 203: REM PRINT ALL TOKEN WORDS
65 NEXT:PRINT#4,TN,X$:NEXT TN:PRINT#4:CLOSE
```

Depending on your particular need, you may want to consider the technique outlined in location 153 ($99) for creating a detokenized program listing on tape.

If you're programming in ML, you'll want to examine the routine at $C724–C740 as a model for your own routine. It has too many LIST dependencies to JSR to directly.

See Appendix C for a list of the tokens and their corresponding token numbers.

49566 $C19E   ERRTAB
Table of BASIC error messages.

BASIC preempts the Kernal error messages in program mode. BASIC has its own error messages, and prefers them over the Kernal message of I/O ERROR plus an error number. See the list of Kernal messages at location 61812 ($F174).

The following program can be used to produce a reference chart showing the message number, vector address, message address, and complete text of each message:

Program 8–2. Error Message Display

```
10 OPEN 4,4 : REM CHANGE TO 4,3 TO DISPLAY ON THE
   {SPACE}SCREEN
20 PRINT#4,"NUM VECTOR MSG @{2 SPACES}-----MESSAGE
   -----" : PRINT#4
30 FOR V = 49960 TO 50019 STEP 2 : N = N + 1 : N$=
   " "+RIGHT$(STR$(N),2)
40 S = PEEK(V) + ( PEEK(V+1)*256 )
50 PRINT#4,N$;V;S"{SHIFT-SPACE}"; : T=0
60 PRINT#4,CHR$( PEEK(S+T) AND 127 ); : IF PEEK(S+
   T) <128 THEN T=T+1: GOTO 60
70 PRINT#4 : NEXT
80 FOR X=1 TO 20 : PRINT#4 : NEXT : END
```

Table 8—4. BASIC Error Messages

The last letter of each message has the high order bit on.

Dec	Hex	Error Message
1	$01	TOO MANY FILES
2	$02	FILE OPEN
3	$03	FILE NOT OPEN
4	$04	FILE NOT FOUND
5	$05	DEVICE NOT PRESENT
6	$06	NOT INPUT FILE
7	$07	NOT OUTPUT FILE
8	$08	MISSING filename
9	$09	ILLEGAL DEVICE NUMBER
10	$0A	NEXT WITHOUT FOR
11	$0B	SYNTAX
12	$0C	RETURN WITHOUT GOSUB
13	$0D	OUT OF DATA
14	$0E	ILLEGAL QUANTITY
15	$0F	OVERFLOW
16	$10	OUT OF MEMORY
17	$11	UNDEF'D STATEMENT
18	12	BAD SUBSCRIPT
19	$13	REDIM'D ARRAY
20	$14	DIVISION BY ZERO
21	$15	ILLEGAL DIRECT
22	$16	TYPE MISMATCH
23	$17	STRING TOO LONG
24	$18	FILE DATA
25	$19	FORMULA TOO COMPLEX
26	$1A	CAN'T CONTINUE
27	$1B	UNDEF'D FUNCTION
28	$1C	VERIFY
29	$1D	LOAD
30	$1E	BREAK (located in 50020 $C364 table)

49960 $C328   BMSGS*

BASIC error message table vectors.

This area contains 30 pointers to the start of each message in location 49566 ($C19E), in message number order. The BASIC routine ERROR, which displays error messages, uses this table to determine the location of the message text by the number given to it in .X. Because of this, other BASIC routines need not be concerned with error message text, only error numbers.

See location 49566 ($C19E) for a program to print all the BASIC error messages and these pointers.

50020 $C364   MISCMSG*

Miscellaneous messages.

Each message is ended with a byte containing zero. The messages (which include carriage returns, spaces, and linefeeds), are:

● <carriage return> *OK* <carriage return>

This message is displayed when VERIFY is successful in the BASIC routine BLOAD, which also performs the VERIFY for the BVERIF routine.

● <carriage return> <space> *ERROR*

The BASIC routine PRDY, which is part of the ERROR sequence, causes this message to follow a message already displayed.

● <space> IN <space>

After the message ERROR has been displayed, the BASIC routine PRDY calls the BASIC routine PRTIN to print this message and the line number.

● <carriage return> <linefeed> *READY.* <carriage return> <linefeed>

The BASIC routine READY displays this message for several reasons, and goes to the routine MAIN, the main BASIC processing loop.

● <carriage return> <linefeed> *BREAK*

BASIC's routine PRDY prints this message and calls PRTIN to print *IN* and the line number. These are printed when a program encounters STOP or the RUN/STOP key is pressed.

50058 $C38A   SCNSTK

Find FOR and GOSUB entries on the stack.

The stack at 256–511 ($100–1FF) STACK is searched for a particular FOR variable, or for the first one. This area is called by NEXT, FOR, and RETURN routines.

RETURN is looking for the return pointer on the stack and deletes any FOR information it finds. This can be used to your advantage by placing a loop you wish to terminate early in a GOSUB routine.

See locations 73–74 ($49–4A) (FORPNT) and 256 ($100) (STACK).

50104 $C3B8   MAKSPC

Open space in memory for a new BASIC line or variable.

This area is called when a new scalar variable is created or when a BASIC line is stored or replaced. It calls RAMSPC ($C408) to insure that the space is available, and to do garbage collection if not enough space is free. This also adjusts the STREND pointer at location 49–50 ($31–32), which indicates the start of free area. It then falls through to the next routine.

196

50111 $C3BF   MOVEBL
Move a block of memory.

When a new scalar variable is created, a BASIC line is stored or replaced, or when garbage collection needs to move memory, this is called.

This area calls RAMSPC ($C408) to check that space is available and moves the program and/or variables upward to make room. By calling this routine with .A/.Y set to the end of the BASIC dynamic area desired, you could allocate space in that area for your own purposes, or use the technique to move blocks of memory upward. However, this is an advanced topic to explore, and much coordination is required with other pointers. See location 49 ($31), the pointer to the start of the free area.

50171 $C3FB   STKSPC
Check stack requested space available.

Produces OUT OF MEMORY message if the requested amount of stack space, multiplied by two, is unavailable. GOSUB, FOR, and formula evaluation call this routine.

This can be a convenient routine to call from ML.

50184 $C408   RAMSPC
Check that requested space in dynamic area is available.

This routine compares the to-be-allocated end address with the contents of location 51–52 ($33–34), the pointer to the bottom of BASIC active strings.

It calls for garbage collection if not enough space is free and produces an OUT OF MEMORY message if the requested amount of memory is still unavailable after that.

50229 $C435   MEMERR
Set OUT OF MEMORY error message code.

Falls through to the next routine.

50231 $C437   ERROR
BASIC error message routine.

The number of the desired error message is passed to this routine in .X. The routine PRTOS at ($CB3B) is called to display a question mark.

This routine looks up the message number vector in the vector table at 49960 ($C328) and displays the message the address points to. The address points within the BASIC error message table at 49566 ($C19E). The first instruction of this routine is a JMP ($0300),

50281

so you can change the vector at 768 ($300) to point to a front-end or alternate routine if you want.

This routine calls for a close of all input/output channels, resets the current channel in 19 ($13) to the screen (the error messages always appear on the screen—this cancels any CMD device that was specified), and causes the stack to be cleared.

The routine at 51998 ($CB1E) is called to actually display the error message on the screen.

INPUT routines issue messages with a different routine, since processing can continue in their case.

This routine then falls through to the next routine.

50281 $C469   PRDY
Display *ERROR*, or another message pointed to.

This routine is also used to display the BREAK message instead of the ERROR message.

It calls the routine PRTIN at 56770 ($DDC2) to display the word *IN* and line number message if not in direct mode.

This routine then falls through.

50292 $C474   READY
Display *READY.* message.

The READY. message is displayed and the Kernal control messages are enabled by this routine. See location 157 ($9D) for details of these messages.

The routine PRTSTR at 51998 ($CB1E) is called to actually display the error message on the screen.

This could be a convenient routine to call from ML.

This routine then falls through.

50304 $C480   MAIN
Main BASIC loop, receive and execute or store BASIC line.

The first instruction in this routine is JMP ($0302), which normally points back to the next sequential instruction. By changing 770 ($302), you can intercept the keyboard input to BASIC.

This routine calls GETLIN at location 50528 ($C560) to obtain a line from the keyboard, then goes to the routine NEWLIN (the next routine) if a line number is present on the entered line, or to the routine CRNCH if no line number is present. Direct commands and program lines are separated at this point because of this.

50332 $C49C   NEWLIN
Store/replace a BASIC program line.

This routine tokenizes the line by calling CRNCH at 50553

($C579), then tries to locate the same line number in the BASIC program. If it's found, this routine deletes it. The newly entered line is added to the program in line number sequence, unless only a line number was entered. The latter does not cause a new line to be entered after the old line was deleted.

This routine resets the CHRGET 122–123 ($7A–7B) pointer to the start of the BASIC program when a line of BASIC is entered. This is done by calling STXTPT ($C68E) to copy 43–44 ($2B–2C) TXTTAB into TXTPTR.

The CLR routine at 50782 ($C65E) is called, losing all current program variables.

The BASIC lines are also rechained by calling LNKPRG 50483 ($C533).

50483 $C533   LNKPRG
Rechain BASIC program lines.

This routine recalculates and stores program line link fields by examining each line of the BASIC program, from where pointer 43–44 ($2B–2C) TXTTAB is pointing up to an old link field containing zeros. That signals the end of the program.

50528 $C560   GETLIN
Receive input from device and fill the BASIC text buffer.

This routine is also used for INPUT and INPUT#.

It calls the Kernal routine CHRIN 61966 ($F20E) through the vector at 804 ($324) to obtain input characters from the open channel until a carriage return or 89 characters have been received. A STRING TOO LONG error message is displayed if no carriage return is found within the 89-character input stream. The characters are stored in the BASIC input buffer at 512–600 ($200–258). The carriage return stores a 0 byte in that buffer. (The BASIC 2.0 feature of a 15 ($0F) character on input suppressing the display of the characters to the screen has been dropped.) The length of the BASIC input buffer and the coding of this routine are the cause of the 88-byte INPUT restriction.

50553 $C579   CRNCH
Tokenize the BASIC line in BASIC text buffer.

This routine tokenizes the line from and back into the BASIC text buffer at location 512 ($200), using the table of tokens at 49310 ($C09E). Bytes within quotes are not tokenized, and the ? word is replaced by the PRINT token.

This is the routine that recognizes abbreviations for BASIC keywords. TXTPTR at location 122 ($7A) is used as a pointer through the process.

The length of the resulting tokenized line is stored in 11 ($B). A vector to this routine is at location 772–773 ($304–305).

50707　　　　　$C613　　　　　FINLIN
Find the BASIC line from its line number.

The two-byte integer line number in location 20–21 ($14–15) is searched for in the BASIC program lines by this routine. Location 95–96 ($5F–60) is set to the address of the link field for that line, if found, and the carry is set in .P. This routine is called by the routine NEWLIN, and the BASIC keywords LIST, GOTO, and GOSUB.

The RTS at the end of this routine is used by many routines.

50754　　　　　$C642　　　　　NEW
BASIC NEW.

This routine stores zeros in the first BASIC line field and sets the end-of-BASIC program pointer at 45–46 ($2D–2E) to the contents of 43–44 ($2B–2C) plus two bytes. It also calls the routine STXTPT 50830 ($C68E) to set TXTPTR to the beginning of the BASIC program.

See location 43–44 ($2B–2C) for methods of recovering from an inadvertent NEW.

This routine is called by routine INITBA 58276 ($E3A4) at power-on/reset.

This falls through to the next routine.

50782　　　　　$C65E　　　　　CLR
BASIC CLR.

First, a call is made to the ICLALL vector at 812–813 ($32C–32D), which points at the abort-all-files routine CLALL at 62447 ($F3EF).

The BASIC strings are eliminated by changing the pointer to the bottom-of-BASIC active strings at 51–52 ($33–34) to the current contents of 55–56 ($37–38), which is the pointer to the end-of-BASIC memory. The pointer to the end-of-BASIC program at 45–46 ($2D–2E) is copied to 47–48 ($2F–30), the pointer to the end-of-BASIC variables, as well as copied to 49–50 ($31–32), the pointer to the end-of-BASIC arrays, start of free area. This eliminates all scalar and array variables. The variables have not been actually erased, though. RESTORE is called to reset 65–66 ($41–42) to the beginning of the BASIC program.

This routine is called when RUN is entered in direct mode, an RS-232 OPEN is done, or NEW is issued.

The temporary string stack at 22 ($16) TEMPPT is reset, and the 6502 stack pointer (.S) is also reset. For that reason, you won't want to call this routine from within a subroutine. The stack pointer is set

to 506 ($1FA), leaving the most recent JSR return address on the stack. This stack clearing is also called for by a warm restart (RUN/STOP–RESTORE key) or an error message being issued by ERROR.

50830　　　　　　$C68E　　　　　　STXTPT
Back up TXTPTR to the start of the program.

This routine copies the contents of location 43–44 ($2B–2C) into location 122–123 ($7A–7B) so the program will be scanned from the beginning.

If SYS 50830 is executed in a BASIC program, the program will branch to the first line of the program.

This is called by the LOAD and NEW routines.

50844　　　　　　$C69C　　　　　　LIST
BASIC LIST.

This performs the LIST function with its various formats of: no line numbers, line number range, starting at line number, up to a given line number, or only a specific line number.

The ending line number (or $FFFF) for all) is stored in location 20–21 ($14–15). The LIST routine calls the routine FINLIN at 50707 ($C613) to find the starting specified BASIC line. Location 95–96 ($5F–60) is used as pointer through the BASIC program as it is listed. Link fields on the BASIC lines are used to find the next line. Routine PRTFIX 56781 ($DDCD) is used to print the line number. The stop key is tested for after every line displayed.

LIST calls QPLOP 50970 ($C71A) to perform detokenizing of the BASIC lines.

See location 153 ($99), the input device number, for instructions for reading tape as though it were the keyboard, using LIST.

LIST does not recognize the fact that it is within a REM statement and will detokenize any tokens found there. This can cause a SYNTAX ERROR message if the character has a code greater than 127, but not 255 (PI). Try including a shifted L in a REM, and you'll receive a SYNTAX ERROR message.

REM-embedded cursor controls, color controls, and reverse controls will be printed as found, causing differences between the actual program and displayed listing. This can be used to customize or disguise the program listing. When the same LIST is directed to a printer with CMD, most of the manipulations done with REM-embedded controls will be revealed.

50970　　　　　　$C71A　　　　　　QPLOP
List detokenized BASIC keywords.

This routine is logically part of the LIST routine ($C69C) and may not be used as a separate routine. However, it is fairly short

and may be used as a model.

All characters are sent to this routine, which branches to two addresses, depending on whether the input was displayed as untokenized or was not a token.

A JMP off the vector at 774 ($306) is the first instruction in the routine, normally jumping back to the next instruction. That vector may be changed to intercept this routine, providing the capability to LIST added keywords.

51010 $C742   FOR

BASIC FOR.

FOR is one of the most powerful BASIC keywords. This routine sets up the FOR environment and NEXT controls the repetition of the desired loop. Eleven JSR calls are contained within FOR, demonstrating the large amount of work done for you by this word and routine.

FOR saves the details of the requested loop on the stack; see the description of the stack entry for FOR to location 256 ($100).

The variable used in FOR must be a scalar floating point. In other words, there can be no % in the name. By reissuing a FOR loop for the same variable name, the previous FOR and all inter-nested FORs are cancelled. The variable used in FOR may be changed within the loop to control the number of repetitions. However, a variable used to express the upper limit of the loop will not cause an early end of the loop if it's changed within the FOR loop. This is because the upper limit is stored within the stacked items.

A FOR loop is always executed at least once.

If you leave off the name of a NEXT statement, you must insure that the correct nesting of FOR loops is maintained.

51118 $C7AE   NEWSTT

Finds (for execution) the next BASIC statement.

This routine tests for the STOP key being pressed, updates the CURLINE location (57–58, $39–3A), which holds the current BASIC line number, if not in direct mode, and positions TXTPTR to the beginning of the statement.

Location 776–777 ($308–309) contains a vector to this routine. If the end of program is detected by a 0,0 in the link field, the END routine is jumped to.

This calls the following routine (GONE) to execute the statement.

Direct mode statements skip the CURLIN update.

51172 $C7E4   GONE

Execute the current BASIC statement.

If the statement doesn't start with a token, the routine LET is jumped to.

A vector to this routine is located at 776–777 ($308–309).

GO TO tokens are treated as though they were GOTO.

The dispatch vector tables starting at location 49164 ($C00C) are searched and the proper vector for the token is pushed onto the stack for the next RTS to cause a branch to that address.

51229　　　　$C81D　　　　RESTORE
BASIC RESTORE.

This resets the beginning of DATA statement scan by simply copying the current pointer to the start of the BASIC program to location 65–66 ($41–42), the pointer to the DATA statement.

51244　　　　$C82C　　　　TSTSTOP
Test for STOP key.

This routine is a JSR to the Kernal jump vector at 65505 ($FFE1), falling through to the next routine.

51247　　　　$C82F　　　　BSTOP
BASIC STOP.

This routine also falls through to the next routine, skipping the clear carry instruction to differentiate between STOP and END.

51249　　　　$C831　　　　END
BASIC END.

This routine clears the carry flag to indicate an END was issued. The current BASIC line number is moved from 57 ($39) to 59 ($3B) for a possible CONT being entered later. The current TXTPTR value is also saved for the same reason. The value at location 57 ($39) is copied to 59 ($3B). The return address is dropped from the stack and *READY.* or *BREAK IN nnn* is displayed.

51287　　　　$C857　　　　CONT
BASIC CONT.

The saved TXTPTR at 61 ($3D) and the saved CURLIN at 59 ($3B) are restored from 59 ($3B), where STOP, END, or the RUN/ STOP key saved them in routine END ($C831). Location 62 ($3E) is tested for a 0 and a CAN'T CONTINUE error message is issued if this value is found. (An ERROR caused the break—or program lines were changed.)

Otherwise, CONT allows GONE ($C7E4) to execute BASIC from where it was interrupted.

A GOTO line number may work when CAN'T CONTINUE is received, depending on file and variable requirements of the program.

51313 $C871   RUN
BASIC RUN.

This routine resets the TXTPTR to the beginning of the program if no line number was entered with RUN, or causes the specified line number to be acted upon by the GOTO routine. Either way, a CLR is performed, losing all current variables.

51331 $C883   GOSUB
BASIC GOSUB.

The following information is pushed onto the stack in the order listed by this routine:
1. The current TXTPTR (location of the characters being scanned by BASIC, so it can resume here at RETURN)
2. The CURLINE value (so that the current line can be resumed)
3. The constant value of 141 ($8D) (to identify the GOSUB entries on the stack)
Then a JSR to the GOTO routine is performed.

A subroutine may call itself (called *recursion*), but some type of exit logic must be included to prevent the stack from being filled by GOSUB entries.

See the search direction discussion at location 20–21 ($14–15) and the subroutine location discussion at 43–44 ($2B–2C) TXTTAB.

51360 $C8A0   GOTO
BASIC GOTO.

The target line number is converted from character to LSB/MSB using the DECBIN routine, which places the output integer in location 20–21 ($14–15). The MSB *only* of the current line and the MSB of the target line are compared. If the target is higher, the routine FINLIN ($C613) is called to find the line from the line number, and TXTPTR is adjusted. Otherwise, TXTTAB is used as a parameter to FINLIN, and the scan for the line number starts at the beginning of the program. Once again TXTPTR is adjusted, and when CHRGET is later called, that line will be executed.

51410 $C8D2   RETURN
BASIC RETURN.

This routine calls SCNSTK 50058 ($C38A) to find the GOSUB entry, moves the CURLIN saved contents that are in the stack to CURLIN and the TXTPTR saved in the stack to TXTPTR, thereby

directing NEWSTT 51118 ($C7AE) back to the statement following
GOSUB.

The routine also puts ($FF) into FORPNT+1 (73–74, $49–4A)
which effectively cancels any FORs from within the subroutine.

51448 $C8F8   SKIPST
BASIC DATA.

A very simple routine, this calls FIND2 51462 ($C906) to find
the next statement and falls through to the next routine.

Quotes can be used to include commas and colons in a DATA
string. Outside of quotes, a comma delimits the DATA item and a
colon ends the DATA statement.

A null DATA item is created by double commas or an ending
comma.

51451 $C8FB   BUMPTP
Increment TXTPTR by amount in .Y.

This routine is called when several bytes of the current line/
statement need to be skipped. This is ended with an RTS that usu-
ally goes back to the NEWSTT routine at 51118 ($C7AE).

51462 $C906   FIND2
Scan the BASIC text buffer at 512 ($200) for delimiters.

FIND2 is usually used to find the next BASIC statement or line.
Locations 7 and 8 are used to contain the characters being searched
for. The routine automatically ends the search if an end-of-line zero
byte is found and ignores delimiters while inside quote marks.

51496 $C928   IF
BASIC IF.

This is a fairly simple routine in comparison with the program-
ming flexibility it gives you. It calls the FRMEVL routine at 52638
($CD9E) to do the hard part of reducing the expression to a single
term. IF then simply checks the exponent of Floating Point Accu-
mulator 1 at 97 ($61) for a 0, which indicates the whole accumulator
is zero and the text is *false*. If it was a 0, IF calls FIND2 to skip the
rest of the line, jumps to BUMPTP, and then goes on to NEWSTT
for the next line to be executed. If the statement was *true*
(FACEXP<>0), GONE ($C7E4) is branched to, unless a following
THEN was followed by numerics, in which case GOTO is branched
to. If a GOTO is encountered (GOTO *must* be followed with
numerics), the GOTO routine is also branched to.

GO TO (with a space between the words) is not valid after IF
unless it is preceded by THEN. Also, IF X THEN 80 is the equivalent

of IF X <> 0 THEN 80. Once more, IF X GOTO 80 is perfectly valid.

The REM routine below is actually part of this IF routine, and IF actually ends at 51530 ($C94A).

51515   **$C93B**   **REM**
BASIC REM.

The REM routine calls FIND2 51462 ($C906) to skip the rest of the line, jumps to BUMPTP, and then goes on to NEWSTT to execute the next line.

The REM routine is actually part of the previous IF routine.

51531   **$C94B**   **ON**
BASIC ON.

GOSUB or GOTO must follow the variable name and GO TO (with an embedded space) is invalid. This routine decrements the LSB of FAC 101 ($65), until it reaches 0, skipping numbers between commas until then. It then passes the GOTO or GOSUB token and the target line number to the routine GONE at 51172 ($C7E4) to execute.

If the list of line numbers is shorter than the value of the variable, the next statement is executed. For example:

ON X GOTO 100,200:PRINT "WHOOPS"

would print WHOOPS if X was 4.

ON SGN(X)+2 GOTO 10,20,30

will go to 10 if X is negative, 20 if 0, and 30 if positive.

If the variable could be 0 and you wish to GOTO/GOSUB on that value, add at least 1 to the variable.

51563   **$C96B**   **DECBIN**
Convert decimal line number to LSB/MSB format.

This routine is called by the BASIC commands GOTO, LIST, and ON. It's also called by the routine NEWLIN when adding, replacing, or deleting a BASIC line.

This is used to convert and range-check (0–63999) a line number, placing the output LSB/MSB format in location 20–21 ($14–15).

51621   **$C9A5**   **LET**
BASIC LET.

This routine controls the reassignment or creation and initialization of scalar and array variables: strings, floating point, integer, TI$, and TI.

When this routine is finished, the variable or descriptor has been created or modified in the variable pool.

206

This is a rather long routine, with 17 JSRs. Not all are used for the same variable types, though.

The following routines are always called:
EVLVAR ($D08B) evaluate variable
TYPCHK ($CD8A) type-match checking
FRMEVL ($CD9E) evaluate expression
with others called depending on the contents of locations 13 ($D) (type of variable: 255 ($FF)=string; 00=numeric) and 14 ($E) (numeric variable type: 128 ($80)=integer; 00=floating point), which are used to determine the type of variable processed. These flags are set by the initial call to the EVLVAR routine ($CF28).

Floating point variable assignment is passed to routine FACTFP at 56272 ($DBD0) which stores the Floating Point Accumulator in memory as a variable.

51650 $C9C2   LET2
LET: Assign integer variable.

51674 $C9DA   LET5
LET: Assign TI$.

51756 $CA2C   LET9
LET: Assign string variable.

51840 $CA80   PRINTN
BASIC PRINT#.

This routine calls CMD, the following routine, and then jumps to the CLRCHN 65484 ($FFCC) routine to close the output channel.

51846 $CA86   CMD
BASIC CMD.

This routine calls OUTCHN 65481 ($FFC9) to open the output channel, stores the file number in location 19 ($13), stores the current channel number for BASIC I/O routines, and goes to the routine PRINT at 51872 ($CAA0) for the processing of any PRINT# style parameters included. Since PRINT# calls this routine, there is a very good chance that there *will* be parameters.

Unlike PRINT#4, CMD4 leaves the device in listen mode so that future output still is directed to the CMD device.

See location 19 ($13) for more information on CMD.

51866 $CA9A   PRT1

The instructions from here to location 51871 ($CA9F) are part of the PRINT routine.

51872 $CAA0   PRINT
BASIC PRINT.

A long routine, PRINT includes instructions to handle the various forms of output parameters possible, such as floating point variable, TAB, SPC, semicolon, comma, strings, null strings, carriage return/linefeed, PI, ST, TI$, and TI.

All variables are converted to strings and eventually printed by a call to the CCHROUT vector at 65490 ($FFD2) for each character.

51944 $CAE8   PRT6

Part of PRINT, this tabs to the correct column for comma delimiter.

51960 $CAF8   PRT7
BASIC TAB, BASIC SPC.

This TAB and SPC processing routine is part of the routine PRINT.

TAB and SPC are not for the printer, since they are based on the current cursor position.

51998 $CB1E   PRTSTR

Another part of the PRINT routine, this prints a string ended by a carriage return or when the length is decremented to 0.

This routine is called by several other routines to display messages. It can also be called by machine language programs by setting the .Y register to the MSB of the message address, and the .A register to the LSB. The message should be ended with a carriage return, followed by a byte consisting of 0's.

52027 $CB3B   PRTOS

This section of the PRINT routine prints format characters of space, cursor right, or question marks. The latter is for the INPUT routine.

52045 $CB4D   IGRERR

Error message formatting routine for BASIC keywords GET, INPUT, and READ.

52091 $CB7B   GET
BASIC GET.

This routine disallows direct mode entry, opens the input channel, if GET# was specified, by calling INPCHN ($FFC6), and storing the channel number the BASIC I/O channel at location 19 ($13). It

calls READ ($CC06) to perform the I/O, and closes the input channel if necessary by calling the CLRCHN routine at ($FFCC).

You can tell that a single character is being requested by GET, rather than multiple characters by INPUT, when the blinking cursor is not present. If you add a blinking cursor to your GET routine, avoid the character used by the system. Otherwise, the program's user can become confused.

GET (without a file number) retrieves one byte from the keyboard buffer at 631 ($277), which is filled by the IRQ driven routine. For tape, GET retrieves a single character from the BASIC tape buffer at location 829–1019 ($33D–$3FB). When these characters are exhausted, which is determined by testing location 166 ($A6), an additional tape block is read into the buffer. For disk, ST must be checked for a 64, meaning this is the last byte of data, because a GET# beyond that point will return a carriage return.

52133 $CBA5   INPUTN
BASIC INPUT#.

This routine opens the input channel by calling the vector INPCHN at 65478 ($FFC6) and stores the channel number in location 19 ($13). A call to INPUT ($CCBF) is made, then the input channel is closed by calling the CLRCHN routine at ($FFCC).

Rather than EXTRA IGNORED, the extra data is simply discarded. A FILE DATA ERROR message is issued when the data type is different from the variable type.

52159 $CBBF   INPUT
BASIC INPUT.

This routine disallows direct mode entry.

The PRTSTR routine 51998 ($CB1E) is called to print any prompting message specified and the PRTOS routine at 52027 ($CB3B) for the question mark. The latter is printed only if 19 ($13) CHANNL indicates the keyboard.

GETLIN ($C560) is called to receive the input from the device and fill the BASIC text buffer, then READ ($CC06) is jumped to. That routine validates the input and assigns it to the proper variables named.

An active CMD causes an INPUT prompt to be displayed on the CMD device.

By placing two quote marks and a delete character into the keyboard buffer at 631 ($277) and setting the number of characters in the buffer to 3 at location 198 ($C6), you can allow commas and colons to be entered in response to INPUT. You can do this by entering the following line:

POKE 198,3:POKE 631,34:POKE 632,34:POKE 633,20:INPUT
"ENTER MLC OPCODE AND OPERANDS";ML$

The first quote mark puts the INPUT routine into a quoted-string
subroutine; the second cancels that mode for the screen editor so
that the INST, DEL, and cursor keys perform normally.

    The input diversion subject, discussed at locations 153 ($99) and
19 ($13), may be of interest to you. Also see the related discussion at
512 ($200). Another reference is the article "Perfect Commodore
INPUT," by Blaine Standage, in the January 1983 issue of
COMPUTE!.

| **52230** | **$CC06** | **READ** |

    This routine locates the next DATA item for READ, scans the
BASIC text buffer with CHRGET and modifications to TXTPTR,
and assigns incoming information to numeric or string variables,
producing error messages as needed.

    Location 17 ($11) is used as a flag to indicate which of READ,
INPUT, or GET is active. Values in that location mean: 0=INPUT,
64 ($40)=GET, 152 ($98)=READ. See that location for additional
information.

| **52476** | **$CCFC** | **EXTRA** |

INPUT error messages.

    EXTRA IGNORED and REDO FROM START messages, each
followed by carriage return, linefeed, and a zero byte are created by
this routine.

| **52510** | **$CD1E** | **NEXT** |

BASIC NEXT.

    This routine determines if a FOR loop is needed, based on the
presence of a variable name with NEXT. See location 73–74 ($49–
4A) for additional information. This routine calls SCNSTK ($C38A) to
find the FOR entry, and if not found, issues the NEXT WITHOUT
FOR error message.

    The stack entries for FOR are used to apply the STOP value to
the variable and the maximum value specified by TO is checked. If
the loop is done, the stack entries for FOR are purged. Otherwise,
CURLIN and TXTPTR are overlaid from the stack values and
NEWSTT ($C7AE) is branched to. This executes the statement *after*
the FOR statement.

| **52618** | **$CD8A** | **TYPCHK** |

Variable type checking.

    Different entry points to this routine provide four types of vari-
able type checking services for calling routines.

• The first entry point at 52618 ($CD8A) calls FRMEVL ($CD9E) to evaluate the expression, then falls through to the next entry point.

• The second entry point at 52621 ($CD8D) clears the carry flag to indicate that a numeric check is to be performed on a variable.

• The third entry point at 52623 ($CD8F) sets the carry flag to indicate that a string check is to be performed.

• The final entry point at 52624 ($CD90) actually performs the test by comparing the carry flag indicator to the variable type flag that is stored in location 13 ($D), which is the type of variable flag with settings of: 255 ($FF)=string, 00=numeric. If the type match fails, then the message TYPE MISMATCH is indicated and the routine ERROR ($C437) is branched to.

Numerous routines call this routine at any of the four entry points whenever data needs to be checked before placing it in a variable or mixing it with other data.

52638 $CD9E   FRMEVL
Formula/expression evaluation.

This is another powerful routine that BASIC provides. This is the master routine that drives subroutines which extend to location 53222 ($CFE6). However, other routines can call subroutines through that location, too.

The function of these routines is to obtain, parse (break apart syntactically), error check, combine by performing the indicated operations, and resolve to the final answer any expression that the BASIC program contains. This is done for both scalar and array variables, including string or numeric expressions, as well as for constant information and combinations of all.

These routines can call themselves (*recursion*) for inner levels of expressions to be evaluated.

The type-of-variable flag at location 13 ($D) is set by the final result, as is the numeric variable type flag at 14 ($E), if appropriate.

A numeric result is in location 97–102 ($61–66) when these routines finish their work, while a string result is indicated by a pointer in location 100–101 ($64–65). The length of the string will be in location 97 ($61).

Math operation precedence was explored at location 49280 ($C080), the math operation dispatch vector table.

Because of the usage of the stack to contain intermediate results, an OUT OF MEMORY condition may occur if the expression is exceedingly complex and stack space is minimal.

An example of an expression to be evaluated could be:

STR$(72/(X*A%(3)+(SQR(VAL(D$)))*COS(EXP(Z−INT(SY))
+LEN(Q$))))+"%"

which is nonsensical, but you can appreciate the power of these

routines by working out just the *sequence* of the operations to be performed.

The math operation table, the stack, the CHRGET routine, string routines, floating point routines, stack manipulation routines, variable type check,and function routines are used when they can aid this master routine.

52867 $CE83   EVAL
Evaluate a single term of an expression.

This routine has a vector at location 778–779 ($30A–30B).

It performs reduction of a single expression term to its next-level form. The PI symbol is replaced by its numeric floating point equivalent, a number in the program is converted to floating point, and negation is performed.

52904 $CEA8   PIVAL
The floating point number PI=$82 $49 $0F $DA $A1.

52909 $CEAD
Factoring is continued.

NOT is processed, the FN performer at $D3F4 may be called, SGN may be called, and so on.

52948 $CED4
BASIC NOT.

NOT is further processed. Part of the EVAL routine.
Some examples of NOT use:
NOT X= −(X+1), so *NOT 1= −2, NOT 0=1*
NOT −1= 0 so *NOT* true=false
IF NOT TF THEN 840. If TF is 0 (false), then the program goes to line 840.

52977 $CEF1   PAREXP
Evaluation within parentheses is performed.

This is accomplished by calling the following syntax check routines, then calling routine FRMEVL at 52638 ($CD9E) since inner parentheses may be encountered.

52983 $CEF7   RPACHK
Syntax check for)

This falls through to SYNCHR at location 52991 ($CEFF).

52986   **$CEFA**   **LPACHK**

Syntax check for (

     This is actually called before RPACHK ($CEF7), even though it's placed after that routine. This also falls through to SYNCHR ($CEFF).

52989   **$CEFD**   **COMCHK**

Syntax check for ,

     This routine falls through to SYNCHR at location 52991 ($CEFF).

52991   **$CEFF**   **SYNCHR**

Syntax check for a specific character in .A from CHRGET.

     This routine is called from many routines in BASIC.

53000   **$CF08**   **SYNERR**

Cause SYNTAX ERROR message via jump to ERROR ($C437).

53005   **$CF0D**   **FACT10**

Set up index for − (monadic minus).

53012   **$CF14**   **VARRANGE***

Check range of variable ?

53032   **$CF28**   **FACT12**

Obtain variable name and type from EVLVAR ($D08B), check for null string, and handle TI$, TI, and ST references.

53159   **$CFA7**   **FACT17**

Invoke function.

     This routine uses the function dispatch vector table at FUNDSP ($C052) to set the address of the needed function in location 84–86 ($54–56). A JSR ($0054) is then done to the routine.

     The resolution of the function may need the repeated calling of the FRMEVL ($CD9E) routines to perform inner expression evaluation.

53222   **$CFE6**   **ORR**

BASIC OR.

     Sets .Y to ($FF) and falls through to the next routine. See the next routine for an OR *truth* table.

53225 $CFE9   ANDD

BASIC AND.

This routine performs both AND and OR functions, depending on the value in .Y. AND is indicated by 00 in .Y while OR is 255 ($FF). The parameters are converted into two-byte integer values (−32768 to 32767) before processing. Strings, obviously, are not valid arguments. The result is floating point in FAC.

Remember that AND and OR are the lowest in the order of precedence table. Many program logic bugs concerning these statements can be traced if you remember this.

Table 8—5. AND/OR Truth Table

AND T F	OR T F	Resultant values
----+----	----+---	----------------
T T F	T T T	−1 = TRUE
F F F	F T F	0 = FALSE

53270 $D016   COMPAR

Compare numerics or strings.

Also used for BASIC <, =, >.

This routine compares floating point numbers by calling CMPFAC at 56411 ($DC5B), and compares strings using the next routine. The floating point result is left in FAC and in .A : 0=not equal; −1=equal.

String comparisons also set .X as 0, 1, or 2, indicating that the left-hand operand is greater, equal, or less.

53294 $D02E   CMPST

Compare strings.

Strings are compared and the floating point result is left in FAC: 0=not equal; −1=equal.

53377 $D081   DIM

BASIC DIM.

This calls the next routine to create each dimension specified. For example: DIM A(3),L$(8),G(9) would call the EVLVAR routine three times, once for each dimension.

Remember that the zero element of a dimension exists, takes up space, and may be used like any other element.

Also consider the push-up of arrays that must be done each time a new scalar variable, including strings, is newly defined. This may be avoided by defining all scalar variables before defining

arrays. See the discussion at location 45–46 ($2D–2E) for details.

An array will automatically be DIMed to 11 elements (DIM X(10)) if an element is referenced before the DIM statement is encountered. This can cause a REDIM'D ARRAY message if the DIM is later encountered. Also keep in mind that a DIM statement should not be put within a loop.

A BAD SUBSCRIPT message is issued if the subscript exceeds the DIM size.

Because of the unique way that arrays are stored in BASIC, you can cause BASIC to reclaim the space taken by *all* arrays by including the following instructions:

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

This program line causes the array storage pool to be ignored by instructing BASIC that the end of arrays is the same location as the end of scalar variables.

See Appendix B for a description of the internal format of BASIC storage of variables.

53387 $D08B   EVLVAR
Locate or create variable.

The variable name is checked for proper syntax. Locations 13 ($D; type of variable) and 14 ($E; numeric variable type) are set to indicate the qualities of the variable. Location 16 ($10) is set for an FN or subscripted variable. Next, the name of the variable is saved in location 69–70 ($45–46) with the appropriate high-order bit settings to denote the variable type.

If the variable is an array, the routine ARY is called. Otherwise, the next routine is called to locate the variable.

Appendix B has a description of the variable formats in storage.

53479 $D0E7   FNDVAR
Locate the variable.

The variable specified in location 69–70 ($45–46) is searched for, beginning where the pointer to the start of variables specifies. This pointer is at 45–46 ($2D–2E). The variable is searched for up to the location indicated by the arrays start pointer, which is at 47–48 ($2F–30).

If the variable cannot be found, the routine MAKVAR at 53533 ($D11D) is called to create the variable.

If the variable *is* found, the routine RETVP 53637 ($D185) is called to create a pointer to the variable.

53523 $D113   CHRTST
Check if ASCII character is alphabetic.

This routine is used when checking the variable name for proper syntax to insure that the first character is alphabetic. It's also used by many other routines to perform an alphabetic test.

53533 $D11D   MAKVAR
Create new variable.

This tests the variable name for TI, TI$, or ST, and issues a SYNTAX ERROR message if the user is trying to create a variable with any of those names.

It calls for the relocation of any arrays to a location seven bytes upward to allow the creation of a seven-byte variable descriptor. The routine that moves the arrays upward is MAKSPC, at 50104 ($C3B8). ARYTAB (array starting pointer) at 47–48 ($2F–30) is then reset, and the seven-byte variable descriptor is created. This routine then falls through.

53637 $D185   RETVP
Return the address of the found or created variable.

The address of the variable is stored in location 71–72 ($47–48), the pointer to variable. This points to the byte just after the two-character name in the variable descriptor, but location 95–96 ($5F–60) points to the start of the variable descriptor.

53652 $D194   ARYHED
Calculate the length of an array descriptor.

This routine adds five (two-byte name, two-byte total size, one-byte number of dimensions) to the number of dimensions specified, multiplied by two, to obtain the length of the needed array descriptor.

53669 $D1A5   MAXINT
Maximum integer value of 32768 in floating point.

Expressed as $90 80 00 00 00.

53674 $D1AA   INTIDX
Convert floating point to two-byte fixed point in .A and .Y.

This is used for subscript conversion as well as other tasks.

It calls the routine MAKINT at location 53695 (D1BF) to convert floating point to an integer, and returns the value in .A and .Y.

53682 $D1B2   GETSUB
Convert an expression to integer number.

Used for subscripts and other fixed expressions, this routine calls FRMEVL at 52638 ($CD9E) to evaluate the expression, displays an

ILLEGAL QUANTITY message if the number is negative, and falls through to the next routine.

53695 $D1BF   MAKINT
Convert floating point to signed integer.

This routine calls FPINT 56475 ($DC9B) to convert floating point to integer.

53713 $D1D1   ARY
Find an array item or create an array.

The stack is loaded with the description of the array variable, GETSUB 53682 ($D1B2) is called to resolve subscript expressions, and the array is searched for in the area bounded by the pointers to the start of arrays at 47–48 ($2F–30) and to the end of arrays at 49–50 ($31–32).

If the array is found, this routine jumps to ARY2 at 53837 ($D24D) for testing of the subscript value. Otherwise, a jump is made to ARY6 at 53857 ($D261) to create the array.

53829 $D245   BADSUB
Display BAD SUBSCRIPT message.

53832 $D248   ILQUAN
Display ILLEGAL QUANTITY message.

53837 $D24D   ARY2
Found the array, check the subscript range.

This checks to see whether the subscripts are within the size of the array; it branches to one of the above message routines if not. It also checks for redimensioning of an array.

If another DIM is specified with a different number of subscripts, this routine issues a REDIM'D ARRAY error message.

This then jumps to ARY14 at 53994 ($D2EA) to retrieve a particular element of an array.

53857 $D261   ARY6
Create an array.

This routine calls two routines: ARYHED at 53652 ($D194) to calculate the size of the array descriptor needed and RAMSPC at 50184 ($C408) to insure the availability of enough memory for the array. It also creates the array descriptor and calls M16 54092 ($D34C) to calculate the array size when creating a multidimension array. The pointer to the end of arrays at 49–50 ($31–32) is adjusted. Finally, the whole of the new array is initialized to zeros.

53994 $D2EA   ARY14
Locate a particular array element.

This checks the range of the subscript and the number of sub-scripts specified. It calculates the address within the array descriptor of the needed element and returns the address of the element in location 71–72 ($47–48). The routine M16 at 54092 ($D34C) is called to calculate the dimension size when locating elements in a multidimension array.

54092 $D34C   M16
Compute multidimension array size.

M16 multiplies the size of the current dimension by the size of the next dimension in an array.

54141 $D37D   FRE
BASIC FRE.

This routine calls for garbage collection and calculates the free area size.

See the garbage collection routine at 54566 ($D526). Garbage collection can also be triggered by any request for scalar variable space, string space, or array space when there's not enough room to satisfy that request. The routine RAMSPC 50184 ($C408) is usually responsible for calling for garbage collection when it detects that.

An interesting fact is that a string argument (FRE(XYZ)) causes the temporary string of XYZ to be purged before garbage collection is done. Garbage collection always processes the string storage stacked pointers.

The FRE calculation is done by subtracting the pointer to the end of arrays at 49–50 ($31–32) from the pointer to the bottom of arrays at 51–52 ($33–34), with the result placed in .Y and .A. The routine falls through to the next routine.

54161 $D391   MAKFP
Convert .Y (LSB) and .A (MSB) integer to floating point.

This routine actually sets up the conversion and calls INTFP1 56388 ($DC44) to do the actual work.

The reverse conversion is done by FPINT at 56475 ($DC9B).

54174 $D39E   POS
BASIC POS.

Calls the CPLOT* vector at 65520 ($FFF0) to get the position of the cursor, then calls the routine MAKFP 54161 ($D391) to convert it to a floating point number.

The expression within parentheses is ignored.

The contents of location 211 ($D3), the cursor position on logical screen line, are retrieved and converted to floating point. You could do the same with PS=PEEK(211).

POS is not for the printer, since it is based upon the current cursor position.

54182 $D3A6   NODIRM
Check if a statement is entered in direct mode.

The routine MAIN sets location 58 ($3A) to 255 ($FF) when input from the keyboard is received without a line number.

This routine checks that location and displays an ILLEGAL DIRECT message if that is the case.

This is called by any other routine that prohibits direct mode commands.

54190 $D3AE   UNDEF
Issue an UNDEF'D FUNCTION message for EVALFN ($D3F4).

The FN that was specified has not been defined by a previous DEF FN statement.

54195 $D3B3   DEF
BASIC DEF.

This routine calls FN at 54241 ($D3E1) to check the syntax and create the descriptor for the function. The routine NODIRM at 54182 ($D3A6) is called to eliminate direct mode. More syntax checking is done, and this routine pushes the following onto the stack: garbage byte, the dependent variable's address (in DEF FNXX(A)=2*B, A is the dependent variable), and the address of the DEF in the line.

The definition is skipped (not to be syntax checked till it is actually used in an expression) by calling BUMPTP at 51448 ($C8F8), and a jump is made to EVFN3 54351 ($D44F) to build the descriptor for the function.

When a syntax error is discovered in a DEF FN statement, it will be flagged as an error within the calling FN statement.

Even though the DEF statement must fit on one line of BASIC, you can *chain* them in the following manner:

DEF FN A1(X)=6250/(M*(68+LL))
DEF FN A0(X)=COS(FNA1(−X))+(FI/FN)

The above example illustrates how one function can be extended by including another. The program could issue Y=FN A0(312) to access the combined function definition.

DEF FN must be encountered before any use of the function by FN statements. Otherwise, an UNDEF'D FUNCTION error message is issued.

The dependent variable's contents will not be changed by the execution of the function; see the EVALFN ($D3F4) routine description.

When a program issues LOAD for a shorter program, the DEF FN variables need to be redefined since the variable points to the program line containing the DEF FN statement in the old program.

See Appendix B for a description of the internal format of BASIC statements and DEF FN variables.

Also refer to "User-Defined Functions: Defined," by Myron Miller, in the September 1982 issue of *COMPUTE!*.

54241 $D3E1   FN
Check DEF FN and FN syntax.

This routine insures that an FN token follows the word DEF and evaluates the syntax of the name of the function, which must follow floating point variable naming conventions. It calls the variable location/creation routine EVLVAR at 53387 ($D08B) to cause the variable descriptor to be found or created for the function.

54260 $D3F4   EVALFN
BASIC FN.

This routine calls the FN routine at 54241 ($D3E1) to check the syntax of the FN statement and to obtain the address of the function descriptor. It also causes the expression within parentheses (FNXX(AB*C)) to be evaluated by PAREXP 52977 ($CEF1), and stacks the contents of the dependent variable to preserve the current value. In DEF FN NA(X)=3*X, X is the dependent variable. The routine then uses the dependent variable descriptor area in the variable pool as a work area to hold the floating point number obtained by evaluating the expression on the right side of the = sign in the DEF statement. The final result is placed in FAC, and the dependent variable is restored from the saved contents previously stacked. The routine then falls through to the next routine.

Expression evaluation determines when to call this routine. You'll notice that FN is not in the function or keyword dispatch vector table.

54351 $D44F   EVFN3
Store DEF FN values into the function descriptor from stack.

54373 $D465   STR
BASIC STR$.

This routine insures that the parameter given is numeric, calls FLTASC 56797 ($DDDD) to convert FAC to ASCII, and then calls MAKSTR 54407 ($D487) to create the string.

54389 $D475   ALC1
Calculate new string length and vector.

    Passing a pointer to the new string, this routine calls ALCSPC 54516 ($D4F4) to allocate memory space for a string.

    When finished, FAC contains the string length in location 97 ($61) and the next two bytes contain a pointer to the string.

54407 $D487   MAKSTR
Scan and set up string.

    This routine finds the end of the string, calculates the length of the string, and calls ALC1 54389 ($D475) to allocate memory space for the string. It then calls XFRST1 54920 ($D688) to save the string in memory or checks the temporary string stack at location 22 ($16) to insure that there is room for another descriptor. If not, it gives the message FORMULA TOO COMPLEX. If the string originated from the BASIC text buffer at 512 ($200), a temporary string stack descriptor is used to point to the string in the BASIC program.

    Many routines, such as PRINT, INPUT, READ, STR$, and expression evaluation, call this routine to perform string setup.

54516 $D4F4   ALCSPC
Allocate memory space for a string.

    Passing the amount of space needed for a string, this routine checks that there is space available in free memory and adjusts the pointer to the bottom of active strings at location 51–52 ($33–34) to accommodate the string. If space is not available, GRBCOL is called, and the memory allocation is tried again. If still not available, the OUT OF MEMORY message is issued.

54566 $D526   GRBCOL
Garbage collection.

    Each time a string is redefined (changed in any way, including concatenating with +), the string is actually rebuilt in the string area at the high-end memory. Garbage collection is the process of scanning all the string descriptors in the variable pool, temporary string stack, and arrays; finding the string that is still in use that is at the highest location in memory; and moving it as high as possible in the string storage area. Then *all* the descriptors are scanned again for the next highest in-use string to be moved to the highest unused area of the string pool. This continues through all the string descriptors, until all in-use strings have been pushed upwards, overlaying any discarded strings.

    Any strings that have a length of zero in the descriptor are ignored and will be recreated if referenced again. Then the pointer at

51–52 ($33–34) is adjusted to reflect the new bottom of active
strings.

During all this collection time, the STOP key is not checked.

In the article "Learning about Garbage Collection," by Jim
Butterfield, in the March 1981 issue of *COMPUTE!*, an important set
of conclusions was demonstrated: Garbage collection will scan all the
string descriptors and repack the strings whether any have been
abandoned or not. A collection immediately after a collection will
still require the same activity twice. Strings that are part of a pro-
gram line or DATA line don't affect the collection time. Abandoned
old strings don't take much time during collection—only those that
are kept cost time. Collection time is proportional to the square of
the number of strings manufactured using concatenation, RIGHT$,
MID$, LEFT$, and so on.

54717 $D5BD   GCOLI3
Check if most eligible string to collect.

This routine is called by the garbage collect routine described
above to determine if the current string is at the highest memory
location.

54790 $D606   COLECT
Garbage collect a string.

This is called by the garbage collect routine GRBCOL to move
the string to high string memory and update the descriptor to point
to its new location. This routine then calls the routine MOVEBL
50111 ($C3BF) to actually move the string.

54845 $D63D   ADDSTR
BASIC +, concatenate string.

The ADDSTR routine checks the length of the combined string
and issues a STRING TOO LONG message if needed. It then calls
ALCSPC 54516 ($D4F4) to allocate space for the combined length,
calls XFERSTR at 54906 ($D67A) to build the new string in the new
area, and calls for the deletion of the old temporary or permanent
strings.

54906 $D67A   XFERSTR
Move string in memory.

This is a utility subroutine called by several other routines to
move a string from one point to another.

Location 53–54 ($35–36) is used as a pointer to the target loca-
tion and location 34–35 ($22–23) as a pointer to the source string.

54947   **$D6A3**   **DELST**

Discard a temporary string.

The pointer to the string descriptor is passed to this routine in location 100–101 ($64–65).

This routine calls the DELTSD 55003 ($D6DB) routine.

If the string descriptor is on the temporary string stack, the routine is not interested in reclaiming it, but saves a pointer to the actual string in location 34–35 ($22–23). If the string is *not* on the temporary stack and is the very last string in the string storage area, the pointer to the bottom of active strings at location 51–52 ($33–34) is moved up the length of the string to deallocate it.

This routine is called by several other routines. One reason it's called by so many others is that it conveniently takes a pointer to a string descriptor and returns a pointer to the actual string in location 34–35 ($22–23), with the length of the string left in .A.

55003   **$D6DB**   **DELTSD**

Clean up the temporary string descriptor stack.

If the pointer to the string descriptor that the routine receives is pointing into the temporary string descriptor stack, the descriptor is deleted. See location 22–24 ($16–18).

55020   **$D6EC**   **CHR**

BASIC CHR$.

Creates a descriptor on the temporary string stack for the newly created one-byte string with the value specified in the argument.

55040   **$D700**   **LEFT**

BASIC LEFT$.

This routine calls the routine FINLMR at 55137 ($D761) for parameters and creates a temporary string descriptor with the left-hand amount of the string specified. It will not extend the string.

55084   **$D72C**   **RIGHT**

BASIC RIGHT$.

Calls FINLMR 55137 ($D761) for parameters and uses a complemented position parameter to allow the routine LEFT at 55040 ($D700) to perform its task.

55095   **$D737**   **MID**

BASIC MID$.

This routine calls FINLMR at 55137 ($D761) for parameters, checks for a zero length string (ILLEGAL QUANTITY), and the

begin and end points are calculated, with the default of the end of the string if the length is not specified. LEFT 55040 ($D700) is given the task of actually doing the extraction and building of the new string.

55137 $D761   FINLMR
Obtain string parameters for LEFT$, MID$, and RIGHT$.

The first two parameters for these string functions are pulled from the stack and stored in work areas.

55164 $D77C   LEN
BASIC LEN$.

This calls GSINFO 55170 ($D782) to obtain the length of the string, then calls MAKFP at 54161 ($D391) to convert it to a floating point number.

The string contents are not counted directly, and the length parameter of the descriptor is used.

55170 $D782   GSINFO
Get string information.

GSINFO is called by the BASIC commands ASC, LEN, and VAL to obtain the length of the string and a pointer to the string in location 34–35 ($22–23). This routine calls DELST 54947 ($D6A3) to obtain the information.

55179 $D78B   ASC
BASIC ASC.

This calls GSINFO 55170 ($D782) to obtain the pointer to the string; only the first character is converted to floating point by a call to MAKFP 54161 ($D391).

An ILLEGAL QUANTITY message is issued if the length of the string is 0, as in X$=""; you'll see recommendations for using X=ASC(X$+CHR$(0)) to overcome this problem.

Readability of the program can be increased by using this function. For instance, POKE 7,58 is not as clear as POKE 7,ASC(:).

The reverse of ASC is CHR$, *not* STR$.

55195 $D79B   GETBYT
Obtain number 0–255.

A one-byte parameter is obtained by evaluating the expression and checking that it reduced to the range 0–255.

This routine calls GETSUB 53682 ($D1B2) to convert the result to a positive integer. This type of value is used by POKE and WAIT where the value cannot exceed the storage range of a single byte.

55213 $D7AD   VAL
BASIC VAL.

String information is obtained from GSINFO at 55170 ($D782), CHRGET is used to scan the string after the previous TXTPTR is saved, and ASCFLT 56563 ($DCF3) is called to turn the ASCII numeric values into a floating point number. Sign characters, decimal points, exponentiation, and spaces are all valid. The first illogical character, including a second decimal point, terminates the VAL function, but the expression may contain other function calls (for example, VAL(MID$(B$,6,8))).

When the length of the argument is 0, the returned value will also be 0.

55275 $D7EB   GETAD
Get two parameters for POKE and WAIT.

A two-byte address is made into an integer in location 20–21 ($14–15) by a call to the routine MAKADR at location 55287 ($D7F7), and a one-byte (0–255) parameter is obtained by a call to the GETBYT routine at 55195 ($D79B) and left in .X.

55287 $D7F7   MAKADR
Convert floating point FAC to two-byte positive integer.

The range and sign of FAC are checked and an ILLEGAL QUANTITY message is issued if negative, or greater than 65535. This routine calls the routine FPINT 56475 ($DC9B) for the actual conversion, and stores the result in location 20–21 ($14–15).

This routine is used by the SYSTEM (SYS), PEEK, and GETAD routines.

55309 $D80D   PEEK
BASIC PEEK.

This routine uses the address developed by MAKADR, picks up the byte of data at that address, and calls the routine MAKFP at location 54161 ($D391) to convert it to a floating point number.

55332 $D824   POKE
BASIC POKE.

The target address is developed by the GETAD routine; it also hands back the value in .Y for POKE to place at that address. Thus POKE is rather short.

55341 $D82D   WAIT
BASIC WAIT.

Two parameters are obtained from calling the GETADR routine 55275 ($D7EB), and a third is obtained or defaulted to zero by this routine. The second parameter is stored in location 73 ($49) and the third in 74 ($4A). The contents of the address are exclusive-ORed with the third parameter and ANDed with the second; it then loops until the result is not zero.

See locations 197 ($C5), 653 ($28D), and 37137 ($9111) for examples of the use of the WAIT instruction.

Another reference is "All About Commodore's WAIT Instruction," in the January 1983 issue of *COMPUTE!*, by Louis Sander.

55369　　　　$D849　　　　ADD05
Round FAC by .5.

This routine adds .5 to FAC by setting .A and .Y with .5 and calling part of the routine PLUS at location 55399 ($D867). This routine, in turn, is called by FLTASC 56797 ($DDDD) when converting a floating point number to TI$ or to an ASCII string. This is also called by the routine SIN, location 57960 ($E268).

55376　　　　$D850　　　　LAMIN
Subtract memory contents from FAC.

This loads a floating point number in memory to FAC2 and falls through to the next routine.

55379　　　　$D853　　　　SUB
BASIC − (subtract).

This routine subtracts FAC from FAC2, the result placed in FAC. This is accomplished by complementing the sign and jumping to the routine PLUS 55402 ($D86A).

55394　　　　$D862　　　　PLUS1
Perform exponent preshifting (?) and fall through.

This routine apparently requests that the exponent be denormalized prior to the mathematical function until both numbers have equal exponents.

The routine called to perform this is ASRRES at location 55683 ($D983).

See Appendix B for an explanation of normalization of floating point numbers.

55399　　　　$D867　　　　LAPLUS
Add memory contents to FAC.

This loads a floating point number in memory to FAC2 and falls through to the next routine.

55402 $D86A   PLUS
BASIC + (add).

The PLUS routine adds the contents of FAC2 to FAC, falling through to the next routine if the result is negative, or skipping to the routine NORMLZ 55550 ($D8FE) if it is not negative.

55463 $D8A7   PLUS6
Make the result negative if a borrow was done.

This routine readjusts the exponent so that the resulting FAC is negative. Entry points at $D8D2 and $D8D7 are used by many routines to convert simple numbers that have plugged into FAC by these routines into floating point.

55543 $D8F7   ZERFAC
Zero out FAC and make sign positive since result was zero.

55550 $D8FE   NORMLZ
Renormalize the FAC result.

Add any fractions and normalize the result in FAC.

55623 $D947   COMFAC
Complement FAC entirely.

This routine changes FAC into the two's complement form by reversing the bits with EOR ($FF) commands, and adding one.

55678 $D97E   OVERFL
Issue OVERFLOW message and exit.

55683 $D983   ASRRES
Perform exponent preshifting (?) and fall through.

This routine apparently performs exponent denormalization prior to mathematical function until both numbers have equal exponents.

This is called by the following routines:

TIMES3 ($DA59)
FPINT ($DC9B)
PLUS ($D86A)

55740 $D9BC   FPC1
Constant of one for a floating point accumulator.

$81, 00, 00, 00, 00

This constant is also used by FOR as a default STEP stack entry. This value is ORed in the second byte with 128 ($80) at 56251 ($DBBB) when loaded into the accumulator.

55745 $D9C1   LOGCON
Constants for LOG function.

Table 8—6. LOG Constants

Constants	Floating Point Representation
3.0	$03
0.434255942	$7F,5E,56,CB,79
0.576584541	$80,13,9B,0B,64
0.961800759	$80,76,38,93,16
2.885390070	$82,38,AA,3B,20
0.5 * SQR(2)	$80,35,04,F3,34
SQR(2)	$81,35,04,F3,34
−0.5	$80,80,00,00,00
LOG(2)	$80,31,72,17,F8

55786 $D9EA   LOG
BASIC LOG.

Calculation of LOG to base e of FAC to FAC, using the values at LOGCON at location 55745 ($D9C1).

55848 $DA28   TIMES
BASIC * multiplies FAC2 by FAC, leaving the result in FAC. The routine TIMES3 55897 ($DA59) is called once for each mantissa byte-pair.

55897 $DA59   TIMES3
Multiply-a-byte subroutine.

This adds a mantissa byte the number of times specified in .A.

55948 $DA8C   LODARG
Move floating point memory locations to FAC2.

.A and .Y point to the four bytes of memory that contain a three-byte mantissa, plus a sign byte.

55991 $DAB7   MULDIV
Add exponents of FAC and FAC2.

This routine stores the sum of FAC and FAC2 exponents in FAC exponent, testing for an OVERFLOW error.

56034 $DAE2   MULTEN
Multiply FAC by 10.

MULTEN is a subroutine called when converting floating point

to an ASCII value, for example TI$. The PLUS routine 55402 ($D86A) is called to do the task.

56057 $DAF9   FPCTEN
+10 floating point constant: $84,20,00,00,00.

56062 $DAFE   DIVTEN
Divide FAC by 10.

The routine DIVIDE at location 56082 ($DB12) is called to do the work.

56079 $DB0F   LADIV
Move floating point in memory to FAC2.

In preparation for division, this loads FAC2 from memory location.

56082 $DB12   DIVIDE
BASIC / (Divide FAC2 by FAC resulting in FAC).

Before performing the divide operation, a check for DIVISION BY ZERO is made.

56226 $DBA2   LODFAC
Move floating point memory into FAC.

This is called by other routines to load a floating point number pointed to by .A and .Y into FAC.

56263 $DBC7   FACTF2
Move FAC to memory.

This routine is called to store a floating point number into memory location 92–96 ($5C–60), part of TEMPF3.

56266 $DBCA   FACTF1
Move FAC to memory.

FACTF1 is called to store a floating point number into memory location 87–92 ($57–5B), part of TEMPF3. The second byte of the value to be loaded is ORed with 128 ($80) when loading the Floating Point Accumulator.

56272 $DBD0   FACTFP
Move FAC to memory.

This is called to store a floating point number into memory, at the location pointed to by 73–74 ($49–4A).

56276 $DBD4   STORFAC
Perform move of FAC to memory.

The above routines fall through to this routine to actually accomplish the storing of FAC to memory, now pointed to by .X and .Y. This routine uses the first two bytes of location 34–37 ($22–25) as a pointer to the target location.

56316 $DBFC   ATOF
Transfer FAC2 to FAC.

This is a loop that transfers five bytes from FAC2 to FAC; the sixth byte (sign) is forced to be positive by storing a zero in it.

56332 $DC0C   RFTOA
Move FAC to FAC2, with rounding.

This calls the routine ROUND 56347 ($DC1B) and then falls through.

56335 $DC0F   FTOA
Move FAC to FAC2, without rounding.

FTOA moves six bytes (the full exponent, mantissa, and sign) of FAC to FAC2.

56347 $DC1B   ROUND
Round FAC by adjusting the rounding byte.

Location 112 ($70) is doubled, and if now greater than 128 ($80), a one is added to the FAC.

56363 $DC2B   SGNFAC
Test the sign of FAC.

On exit, .A=0 if zero; 1 if positive; or 255 ($FF) if negative.

56377 $DC39   SGN
BASIC SGN.

This calls the routine SGNFAC 56363 ($DC2B) then falls through.

56380 $DC3C   INTFP
Convert the sign obtained above to 0 or −1 in FAC.

If entered without falling through from SGN 56377 ($DC39), this routine converts .A to floating point in FAC.

56388 $DC44   INTFP1

Convert a two-byte integer to floating point in FAC.

The number in location 98–99 ($62–63) is converted into FAC.

56408 $DC58   ABS

BASIC ABS.

The FAC sign byte at 102 ($66) is shifted right one bit to remove any negative sign. The high order bit equals 1 when negative; 0 when positive.

56411 $DC5B   CMPFAC

Compare FAC to memory.

.A and .Y point to the five-byte memory location to be compared with FAC. Afterward, .A=0 if equal; 1 if the memory location is less than FAC; and 255 ($FF) if the memory location contains a number greater than FAC.

The routine SGNFAC 56363 ($DC2B) may be called.

56475 $DC9B   FPINT

Convert FAC floating point to signed integer.

The resulting four-byte integer is left in 98–101 ($62–65). Routine INTFP1 56388 ($DC44) converts a two-byte integer into floating point.

56524 $DCCC   INT

BASIC INT.

FAC is rounded down to an integer in floating point format. The result is four bytes in locations 98–101 ($62–65). This routine is called by many functions requiring an integer in floating point format.

Sometimes INT will round a number downward. This could be avoided by INT(X+.5); however, if X is a negative number, this would round it down, making it a *greater* value. To correctly round negative or positive numbers, use SGN(X)*INT(ABS(X)+.5).

56553 $DCE9   FILFAC

Store the contents of .A in locations 98–101 ($62–65).

Used to zero out the locations.

56563 $DCF3   ASCFLT

Convert an ASCII string to a floating point number in FAC.

This is called by VAL to evaluate and convert the string, allowing +, −, E, spaces, and a decimal point in the string. Note that

location 95 ($5F) is used as a switch to indicate that a decimal point has already been processed.

56702   **$DD7E**   **ASCI8**
Add .A to FAC.

Part of the ASCFLT routine at 56563 ($DCF3), this adds .A to FAC by calling other routines.

56755   **$DDB3**   **FPC12**
String to floating point conversion constants.

Table 8–7. String to Floating Point Constants

Constants	Floating Point Representation
99,999,999.9	$9B,3E,BC,1F,FD
999,999,999.25	$9E,6E,6B,27,FD
1,000,000,000.0	$9E,6E,6B,28,00

56770   **$DDC2**   **PRTIN**
Issue message IN.

.A and .X are loaded with the contents of location 57–58 ($39–3A), then fall through to the next routine.

56781   **$DDCD**   **PRTFIX**
Decimal number display routine.

An integer number (.A*256+.X) is converted to floating point in FAC. The routine FLTASC 56797 ($DDDD) is called to convert the number to a string, and the routine PRTSTR 51998 ($CB1E) is called to print the number. This routine can be called from a machine language program.

56797   **$DDDD**   **FLTASC**
Convert FAC to TI$ or an ASCII string.

A fairly long and involved routine. A work area at 256–270 ($100–10E) is used and the result is left in that area.
This routine is called by several others:

PRINT	51872	$CAA0
PRTFIX	56781	$DDCD
STR	54373	$D465
FACT12	53032	$CF28

57105   **$DF11**   **FLP05**
0.5 constant for rounding and SQR.

$80,00,00,00,00

57110 $DF16   FLTCON

Powers of 10 table, in four-byte fixed integer format.

This is used for converting strings

Constant	Fixed Integer Representation
−100,000,000	$FA,0A,1F,00
+10,000,000	$00,98,96,80
−1,000,000	$FF,F0,BD,C0
+100,000	$00,01,86,A0
−10,000	$FF,FF,D8,F0
+1,000	$00,00,03,E8
−100	$FF,FF,FF,9C
+10	$00,00,00,0A
−1	$FF,FF,FF,FF

57146 $DF3A   HMSCON

Constants for TI$ division conversion, in four-byte fixed integer format.

Constant	Fixed Integer Representation
−60*60*60*10	$FF,DF,0A,80
+60*60*60	$00,03,4B,C0
−60*60*10	$FF,FF,73,60
+60*60	$00,00,0E,10
−60*10	$FF,FF,FD,A8
+60	$00,00,00,3C

57170 $DF52

Unused area, filled with $BF,AA,AA,AA,AA,AA,AA,AA,AA,...

57201 $DF71   SQR

BASIC SQR.

This moves FAC to FAC2, loads FAC with 0.5, and falls through.

57211 $DF7B   EXPONT

BASIC ↑ (up arrow/power).

This routine calculates FAC2 to the FAC power, resulting in FAC. It calls LOG 55786 ($D9EA) for FAC2, which is multiplied by FAC and calls EXP 57325 ($DFED).

57268 $DFB4   NEGFAC
BASIC monadic —

Negate FAC by exclusive ORing the sign byte with a constant of 255 ($FF). Zero is left unchanged.

57279 $DFBF   EXPCON
Table for EXP, in floating point format.

Used to calculate 2 to the N power.

Table 8–8. LOG and EXP Constants

Constant	Floating Point Representation
1/LOG(2)	$81,38,AA,3B,29
7	$07 count of values
.0000214987637	$71,34,58,3E,56
.000143523140	$74,16,7E,B3,1B
.00134226348	$77,2F,EE,E3,85
.00961401701	$7A,1D,84,1C,2A
.0555051269	$7C,63,59,58,0A
.240226385	$7E,75,FD,E7,C6
.693147186	$80,31,72,18,10
1.0	$81,00,00,00,00

57325 $DFED   EXP
BASIC EXP.

The value in FAC is multiplied by 1/LOG(2), converted to an integer, and the table at 57279 ($DFBF) is applied to calculate e (2.718281828) to the power of FAC, and left in FAC. EXP is the reverse function of LOG. For EXP(X), the same results can be achieved by 2.7182818 ↑ X.

Excessive negative numbers will be set to 0 by this routine, while arguments over about 88 will receive the OVERFLOW error message.

Location Range: 57344–58527 ($E000–$E49F)
BASIC Spillover into Kernal ROM

BASIC spills over into this 8K ROM area up to location 58527 ($E49F)—occupying a total of 1183 bytes. RND, SYS, OPEN, CLOSE, LOAD, SAVE, VERIFY, additional trigonometic functions,

and BASIC initialization routines take up the bulk of this 1K+ space.

The remainder of the 8K area contains the routines of the VIC's Kernal operating system. The Kernal handles all the I/O devices, including the screen and keyboard, and provides routines that BASIC and machine language can use to communicate with these devices.

57408 $E040   SEREVL
Series evaluation subroutine.

This calls the next routine to accomplish the evaluation of complex trigonometric functions. A pointer is passed to this routine. This pointer is stored in location 113–114 ($71–72) and is known as the series evaluation pointer. It points to the table of constants for the trigonometric function being evaluated by the evaluation routines. This pointer will specify a location within the tables at 58171 ($E33B), 55745 ($D9C1), 57284 ($DFC4), or 58092 ($E2EC). See those constant tables and the routines around these locations for further information.

57430 $E056   SER2
Math series evaluation routine.

This routine computes polynomials based upon the table pointer passed to it. This table pointer is stored in location 113–114 ($71–72). The initial pointer passed to this routine points at the number of table entries to be processed, which is followed by the table values.

SER2 multiplies and adds the coefficients to FAC to complete the calculation.

57482 $E08A   RNDCI
Table of constants for RND.

This contains two floating point format numbers to use when the argument passed to the BASIC command RND is a positive number.

The first number is used to multiply the last seed by 11,879,546.4 ($98,35,44,7A,00); the second number, .0000000392767778 ($68,28,B1,46,00), is added to the result.

57492 $E094   RND
BASIC RND.

See the discussions at location 139 ($8B) and 57482 ($E08A) for details of the RND function.

57590 $E0F6   PATCHBAS*
BASIC patch routines.

These routines are apparently inserted here to invoke CLR when RS-232 is opened, and to display an error message if a called Kernal routine returns with the carry bit set. BREAK is assumed if the carry bit is set, but the error message number is zero.

A BASIC routine will call these routines rather than calling the Kernal routine directly or with indirect vectors. This adds yet another level of calls for Kernal routines, but also facilitates further changes; only *one* routine in BASIC needs to know the proper address for these Kernal calls.

The patch routines handle calls to the following Kernal routines:

Table 8—9. Calls to the Kernal

Routine	Action	Jump Calls from		Vector Default
$E109	Output a character	$FFD2	$0326	$F27A
$E10F	Input a character	$FFCF	$0324	$F20E
$E115	Set output device	$FFC9	$0320	$F309
$E11B	Set input device	$FFC6	$031E	$F2C7
$E121	Get a character	$FFE4	$032A	$F1F5

A routine at $E0F6 is provided to handle the Kernal's returned carry bit being set. BASIC routines that call the Kernal directly for functions other than the ones provided in these routines commonly branch to that entry point to issue the appropriate error message.

57639 $E127   SYSTEM
BASIC SYS.

The argument expression is evaluated, then type checked and converted to a two-byte integer format by calling routine MAKADR 55287 ($D7F7), which stores the result in LSB/MSB format at location 20–21 ($14–15). A return address within the SYS routine is pushed onto the stack. Then the registers at locations 780–783 ($30C–30F) are loaded and an indirect JMP is performed off the vector stored in location 20–21 ($14–15). When SYS is reentered by the RTS of the target routine, the registers are stored in the SAVE area of 780–783 ($30C–30F) for the next SYS to use.

You *may* modify these saved registers prior to issuing SYS so that the correct parameters are passed. FAC, FAC2, and other locations may need to be set, depending on the requirements of the target routine(s).

57683 $E153   SAVE
BASIC SAVE.

The routine calls PARSL 57809 ($E1D1) to set the filename, device, and secondary address parameters. .X and .Y are loaded with the contents of locations 45–46 ($2D–2E), which is the pointer to the end of the BASIC program, and .A is loaded with the constant of 43 ($2B), the zero page location of the pointer to the start of the BASIC program. A JMP is then made to the Kernal SAVE routine via the vector at $FFD8.

57698 $E162   BVERIF
BASIC VERIFY.

This sets .A to 1 to indicate a VERIFY is in progress and falls through to the next routine, skipping one instruction.

57701 $E165   BLOAD
BASIC LOAD.

The first instruction of this routine loads .A with a zero to indicate that a LOAD was requested. Note that this is skipped by BVERIF.

The routine calls PARSL 57809 ($E1D1) to set the filename, device number, and secondary address parameters. .X and .Y are loaded with the contents of locations 43–44 ($2B–2C), the pointer to the start of the BASIC program area, and .A is left as 1 or 0 to indicate LOAD or VERIFY. A JMP is then made to the Kernal LOAD routine via the vector at $FFD5. See that routine for more information regarding the LOAD sequence of events.

When the Kernal is finished, VERIFY will issue OK if in direct mode, or VERIFY ERROR, depending on the ST setting.

LOAD will issue LOAD ERROR regardless of whether program mode or direct mode is active. Otherwise, direct mode causes the pointer to the end of the BASIC program at 45–46 ($2D–2E) to be reset from the returned values in .X and .Y. The program lines are then rechained by calling the routine LNKPRG at 50483 ($C533).

When LOAD is issued from a program, the current program is ended and the LOADed program begins execution. Only the message PRESS PLAY ON TAPE will be issued. No CLR is issued, so the existing variables are not reset. Modified or constructed string variables will be available, but functions must be redefined. The program being LOADed must not be larger than the LOADing program. The commands 0 LOAD *filename*,8 can be used to load/run a disk or tape program if the device number is omitted or if it's ,1.

57787 $E1BB   FOPEN
BASIC OPEN.

The routine PAROC 57878 ($E216) is called to set the filename, device, secondary address, and argument expression that may be specified with OPEN. See that routine for more details of the OPEN command.

This JMPs off the COPEN* vector at $FFC0 to $031A and finally to the Kernal routine OPEN at 62474 ($F40A).

57796 $E1C4   FCLOSE
BASIC CLOSE.

The routine PAROC at 57878 ($E216) is called to set the file number.

This then JMPs off the CCLOS vector at 65475 ($FFC3) to $031C, and finally to the Kernal routine CLOSE 62282 ($F34A).

If closing tape, an end of file byte (0) is written, and perhaps an I.D. 5 header, if opened with a secondary address of 2, to indicate end of tape.

Disks are marked with an end-of-file marker in the last block (sector) when the file is closed.

Failing to close an output file to disk can cause the file to be irretrievable. If FILE NOT OPEN is received when attempting to close it in direct mode, try OPEN 15,8,15:CLOSE 15.

57809 $E1D1   PARSL
Set LOAD, VERIFY, and SAVE parameters.

This is a rather curious routine that first calls the Kernal routine SETNAM 65097 ($FE49), assuming that there is no filename, and causes the device number and secondary address to default to 1 and 0 by calling the Kernal routine SETLFS 65104 ($FE50) with these parameters set.

Then the routine IFCHRG 57859 ($E203) is called and returns if any parameters were specified on the OPEN, VERIFY, or SAVE. Otherwise, the IFCHRG returns to the routine calling this routine.

If this routine is reentered, it then calls the above mentioned routines to set the specified parameters, calling IFCHRG for each one, in order to exit with the default parameters as soon as there are no more specified.

57859 $E203   IFCHRG
Check whether more characters are in the current statement.

If a call to CHRGET indicates that more characters are in this statement, then return. Otherwise, return to the routine that called the calling routine.

238

57867 $E20B   SKPCOM
Skip any comma in parameters being scanned.

This routine calls the routine COMCHK 52989 ($CEFD) to do its work, then falls through to the next routine.

57870 $E20E   CHRERR
Insure that a parameter is present after a delimiting comma.

If a call to CHRGET indicates that more characters are in this statement, then return. Otherwise, issue a SYNTAX ERROR message.

57878 $E216   PAROC
Handle parameters for OPEN and CLOSE.

This routine sets the filename, device, secondary address, and argument expression that may be specified with OPEN or CLOSE.

The routine first calls the Kernal routine SETNAM 65097 ($FE49), assuming that there is no filename, and causes the device number and secondary address to default to 1 and 0 by calling the Kernal routine SETFLS 65104 ($FE50) with these parameters set.

Then the routine IFCHRG 57859 ($E203) is called and returns if any parameters were specified on the OPEN or CLOSE. Otherwise, it returns to the routine calling this routine.

If this routine is reentered, it then changes the secondary address (also called the command) for serial devices to 255 ($FF) and passes it and the device number retrieved to the routine SETLFS at 65104 ($FE50).

Once again, the routine IFCHRG is called and returns if any parameters specified on the OPEN or CLOSE. Otherwise, it returns to the routine calling this routine.

If this routine is reentered again, it calls SETLFS once again to pass the specified device number and secondary address.

Finally, any specified filename is passed to SETNAM at 65097 ($FE49).

All of this is done in this order to allow all parameters except the file number to be optional.

57953 $E261   COS
BASIC COS.

The cosine of FAC in radians is placed in FAC. This function can also be performed with SIN(X+(?/2)). This adds a constant value of ?/2 to the value and falls through to the next routine.

57960 $E268   SIN
BASIC SIN.

Calculates the sine of FAC in radians. This is also used, with

different constant values, for COS and TAN calculations.

This routine performs repetitive subtraction and division, using the table of trigonometric constant values at 58077 ($E2DD) to achieve the desired results.

To convert radians to degrees, the formula is: DEG=RAD*(180/?). DEG will have a possible range of −90 to +90 degrees.

58033 $E2B1   TAN
BASIC TAN.

Determines the tangent of FAC in radians.

This routine calls SIN 57960 ($E268) and COS 57953 ($E261) to help evaluate the expression.

58077 $E2DD   FPC20
Trigonometric evaluation constant values used for COS, SIN, and TAN.

These table values are in five-byte floating point format.

Table 8–10. Trigonometric Constants

Constant	Floating Point Representation
?/2	$81,49,0F,DA,A2
?*2	$83,49,0F,DA,A2
.25	$7F,00,00,00,00
5	counter of following values
−14.38139	$84,E6,1A,2D,1B
42.007797	$86,28,07,FB,F8
−76.70417	$87,99,68,89,01
81.605223	$87,23,35,DF,E1
−41.34170	$86,A5,5D,E7,28
6.2831853	$83,49,0F,DA,A2
165	$A5 (one-byte)

58123 $E30B   ATN
BASIC ATN.

Determines the arctangent of FAC in radians.

Simpler than other trigonometric functions, this applies the rather long table of constant values at ATNCON 58171 ($E33B).

58171 $E33B   ATNCON
Table of constant values for ATN evaluation.

The values are in five-byte, floating point format.

Table 8—11. ATN Constants

Constant	Floating Point Representation
12 ($B) count of following values	
−6.84793912 E-4	$76,B3,83,BD,D3
+4.85094216 E-3	$79,1E,F4,A6,F5
−.0161117018	$7B,83,FC,B0,10
+.0342096380	$7C,0C,1F,67,CA
−.0542791328	$7C,DE,53,CB,C1
+.0724571965	$7D,14,64,70,4C
−.0898023954	$7D,B7,EA,51,7A
+.1109324133	$7D,63,30,88,7E
−.1428398080	$7E,92,44,99,3A
+.1999991200	$7E,4C,CC,91,C7
+.3333333160	$7F,AA,AA,AA,13
+1.000000000	$81,00,00,00,00

58232　　　　　$E378　　　　　COLDBA*

Perform a cold start of BASIC.

This routine is pointed to by the vector stored in location 49152 ($C000) COLDST*, that is branched upon as the last act of the Kernal during its power-on/reset routine START 64802 ($FD22), which the 6502 automatically jumps to via the vector stored in location 65532 ($FFFC).

This routine calls three routines, resets the stack pointer to 507 ($1FB), and then jumps to READY 50292 ($C474) to display the READY. message.

The three routines that are called are:

● INITVCTRS* 58459 ($E45B), which initializes the vectors starting at 768 ($300).

● INITBA 58276 ($E3A4), which initializes CHRGET and the page zero pointers.

● FREMSG 58372 ($E404), which displays the messages CBM BASIC and BYTES FREE, then jumps to the NEW routine at 50754 ($C642).

For more details of the functions performed, see the listed routines.

A SYS 58232 will restart BASIC.

58247　　　　　$E387　　　　　CGIMAG

CHRGET routine and RND seed to be copied to page zero RAM.

The CHRGET routine stored here is copied to locations 115–138 ($73–$8A) and the seed value for RND is copied to locations 139–143 ($8B–8F). This seed value is .811635157 ($80,4F,C7,52,58).

The copying of this area to zero page is done by the routine INITBA 58276 ($E3A4).

58276 $E3A4 INITBA
Initialize BASIC: Restore CHRGET and page zero pointers.

This routine is called by COLDBA* 58232 ($E378) during the cold start of BASIC.

The CHRGET routine is copied from CGIMG 58247 ($E387) to 115–138 ($73–$8A); locations 0–6 are initialized as described at those locations; various other locations in page zero are initialized; and the address of the bottom of RAM is obtained by calling MEMBOT 65154 ($FE82), and stored in 43–44 ($2B–2C). The top of contiguous RAM is determined by calling MEMTOP 65139 ($FE73) and stored in 51–52 ($33–34) and 55–56 ($37–38); a leading zero is placed at the start of the BASIC program area; and one is added to the pointer specifying that area. The pointer is at location 43–44 ($2B–2C).

58372 $E404 FREMSG
Display cold start of BASIC messages.

This is called by COLDBA* 58232 ($E378) during the cold start of BASIC.

This calculates the amount of free space in the BASIC area by subtracting the start of BASIC program pointer at location 43–44 ($2B–2C) from the pointer to the end of BASIC memory at location 55–56 ($37–38); it also displays the **** CBM BASIC V2 **** message, and the BYTES FREE message. These messages are located at CBMMSG 58409 ($E429). Finally, a jump to the NEW routine at 50754 ($C642) is performed.

58409 $E429 CBMMSG
BASIC cold start messages.

The **** CBM BASIC V2 **** and BYTES FREE messages.

58447 $E44F BASVCTRS*
Six BASIC vectors to be copied to location 768 ($300).

The INITVCTRS* routine at 58459 ($E45B) copies these vectors to RAM. See the description of these vectors at their RAM locations 768–778 ($300–$30A).

58459 $E45B INITVCTRS*
Copy BASIC vectors from ROM to RAM.

This is called by COLDBA* 58232 ($E378) during the cold start of BASIC. Six BASIC vectors at location 58447 ($E44F) are copied to

RAM. See the description of these vectors at their RAM locations
768–778 ($300–$30A).

58471 $E467 WARMBAS*
Perform a warm start of BASIC.

When both the RUN/STOP and RESTORE keys are pressed, the
BREAK* routine at 65234 ($FED2) uses the vector at ($C002) to
branch here after it has completed its duties.

See the BREAK* routine for details of the RUN/STOP–
RESTORE and ML BRK instruction functions performed.

The BASIC current I/O channel at 19 ($13) is reset to 0, indicat-
ing the keyboard. Calls are made to the following routines:
CLRCHN 62451 ($F3F3) to reset all I/O channels to their defaults
(keyboard in, screen out); part of the CLR routine at 50810 ($C67A)
is called to reset the temporary string stack at 22 ($16); and the 6502
stack pointer (.S) is also reset.

Finally, this jumps to the routine READY 50292 ($C474) to dis-
play the READY. message.

For more details of the function performed, see the listed
routines.

Notice that the BASIC program and variables have been pre-
served and that only the stack entries have been lost.

58486 $E476 PATCHER*
Program patch area.

This contains two instructions that are used by the routine
BLOAD 57701 ($E165) to add calls which rechain the program lines
by calling routine LNKPRG 50483 ($C533), to call the routine
RESTOR at 51229 ($C81D), and to reset the temporary string stack
pointer and the 6502 stack pointer.

The remainder of this area is composed of bytes containing the
value 255 ($FF).

Chapter 9

Kernal ROM

Kernal ROM

Location Range: 58528–65535 ($E4A0–$FFFF)

Kernal ROM

The Kernal handles all the I/O devices, including the screen and keyboard, and provides routines that both BASIC and machine language routines can use to communicate with devices.

The Kernal is started when power-on/reset occurs. The 6502 chip automatically jumps to the location specified in location 65532 ($FFFC), which is the Kernal start-up routine. See that location for a description of the start-up activities.

Any Kernal routine can be entered or jumped out of at virtually any point. The descriptions in this chapter are individualized by the entry and exit points most often used. A routine to perform a given function can, in fact, be entered at a point that causes a different function to be processed.

The Kernal likes to use an existing routine if possible, rather than forcing the computer to contain the same instructions at two locations. Many levels of JSRs, stack manipulation of the return address, and other techniques may be confusing when you're first examining a routine. It's typical to use one routine to set up parameters for an existing routine to process. The descriptions below attempt to identify the fact that another routine is called to perform the task attributed to a particular routine, and to indicate places where the Kernal *falls through* (doesn't exit at the end of a routine, but continues into the next sequential routine).

58528 $E4A0   SEROUTI*
Serial: output a 1 on the serial data line.

This routine sets VIA2PCR* ($912C, bit 5) to 0 to indicate the manual output mode; CB2 to be held low. This sends a 1 on the serial line.

This is called by all routines that send serial data.

58537 $E4A9   SEROUTO*
Serial: output a 0 on the serial data line.

This routine sets VIA2PCR* ($912C, bit 5) to 1 to signal the manual output mode; CB2 to be held high. A 0 is thus sent on the serial line.

It's called by all routines that send serial data.

58546   **$E4B2**   **SERGET***
Serial: get an input bit from VIA1 and stabilize.

This retrieves a serial bit from VIA1PA2* at 37151 ($911F) and stabilizes it by testing until it remains constant. The bit is then placed in .A.

58556   **$E4BC**   **PATCHES***
Program patch area.

Preamble to LOADRAM 62786 ($F542) to display a SEARCH-ING message.

58561   **$E4C1**   **entry point:**

Transfer 195–196 ($C3–C4) to 174–175 ($AE–AF) and display LOADING or VERIFYING message.

58575   **$E4CF**   **entry point:**

Addendum to close tape (write header I.D. 5).

58624   **$E500**   **IOBASE**
Retrieve the address of the I/O memory page.

In the VIC-20, a call to this routine returns the address of the first 6522 VIA chip in .X and .Y. The address of that chip is 37136–37151 ($9110–$911F).

This routine is provided to give programs which use the VIA type device to perform I/O operations the ability to locate the address of that device for future compatibility on other similar devices. There are PIAs and CIAs that are of the same type as the VIA.

The routine RND 57492 ($E094) calls this routine to access the VIA timers for the expression RND(0).

To utilize the vectored compatibility feature, always call this routine through the jumping vector at 65523 ($FFF3).

58629   **$E505**   **SCRN**
Retrieve the maximum number of screen columns and lines.

For compatibility purposes, this routine returns the size of the screen in columns and lines, and sets .X to indicate 22 columns and .Y to signify 23 lines.

Be aware that these are constant values and are not obtained by examining the 6560 VIC chip registers.

To utilize the vectored compatibility feature, always call this routine through the jumping vector at 65517 ($FFED).

You may use this routine to determine if your program is running on a VIC-20:

SYS 65517:IF PEEK(781)<>22 OR
PEEK(782)<>23 GOTO xxx
The routine will branch to xxx if not a VIC-20.

58634 $E50A   PLOT
Read or set the current cursor column and line.

The numbers used and returned correspond to the physical lines
and columns, not the logical linked lines.

If the carry flag is clear, .X is saved in location 214 ($D6), the
cursor line number; .Y is saved in 211 ($D3), the cursor row number;
and the cursor is moved to that position on the screen.

If the carry flag is set, .X is loaded from location 214 ($D6) and
.Y is loaded from 211 ($D3). These values are then returned to the
caller.

See location 217–241 ($D9–F1) for an example of using this rou-
tine from BASIC.

The BASIC keywords POS, TAB, SPC, and a comma included in
PRINT parameters cause this routine to return the current cursor
position.

To utilize the vectored compatibility feature, always call this
routine through the jumping vector at 65520 ($FFF0).

58648 $E518   INITSK*
Initialize the 6550 VIC chip, screen, and related pointers.

This is called by the power-on/reset and RUN/STOP–
RESTORE keys routines. It restores the keyboard and screen as I/O
defaults by calling SETIODEF* at location 58811 ($E5BB).

It also resets the 6560 VIC chip registers 36866 ($9002) and
36869 ($9005) from the screen page number stored in 648 ($288).
These are the screen address registers.

The cursor blink counts, keyboard table scan address, keyboard
buffer size, and current foreground color nybble in low memory
work areas are also reset. This routine then falls through to the
following routines.

58719 $E55F   CLSR
Clear the screen.

See location 217–241 ($D9–F1) for an example of using this
routine.

This routine falls through.

58753 $E581   HOME*
Move the cursor to the screen home position.

This falls through to the following routine.

58759 $E587   SETSLINK*
Reset the screen line link table pointers.

See location 217–241 ($D9–F1) for an example of using this routine and an explanation of the screen line link table.

58805 $E5B5   UNUSDNMI*
NMI entry for restore key (no entries to this routine found).

A short routine, this calls SETIODEF* 58811 ($E5BB) and then JuMPs to the HOME* routine at 58753 ($E581). There appears to be no reference to this routine.

58811 $E5BB   SETIODEF*
Reset the default device numbers.

This routine is called by INITSK* 58648 ($E518).

Locations 153–154 ($99–9A) are reset to indicate that the input device number is now the keyboard (device 0) and the output device is the screen (device 3). This then falls through to the next routine.

58819 $E5C3   INITVIC*
Reset the VIC chip registers.

This routine sets the VIC chip registers from a table at VICINIT* 60900 ($EDE4) to 36864–36879 ($9000–900F) during power-on/reset or RUN/STOP–RESTORE processing.

58831 $E5CF   LP2
Get a character from the keyboard queue and shift it down.

.A is loaded with the first character in the keyboard buffer at location 631 ($277). The count of characters in the buffer (at location 198, $C6) is decremented, and the characters remaining in the buffer are shifted down to the beginning.

This is called by routines GETQUE* 58853 ($E5E5) and GETIN 61941 ($F1F5).

58853 $E5E5   GETQUE*
Wait for character to appear in the keyboard buffer.

This routine is called by GET2RTN* at 58905 ($E619) as it is obtaining characters up to a carriage return.

The routine calls for the display of the current character, turns off cursor flashes if there are characters in the keyboard buffer at 631 ($277), calls LP2 58831 ($E5CF) to obtain a character, and loops in the calling instructions until it's returned a character. If SHIFT/RUN has been pressed, it places LOAD and RUN in the keyboard buffer from location 60916 ($EDF4), and returns to GET2RTN* with a

character for it to process. It will return if it's not a carriage return—forcing any previous keyboard buffer characters to be displayed.

The loop in this routine is the reason why INPUT# or calling the routine CHRIN 61966 ($F20E) is not recommended for device 2, RS-232. This loop will cause the other necessary protocol checking instructions not to be performed until a carriage return is received. This can be disastrous for the handshaking RS-232 routines, since they could get out of synchronization with the sending device.

58905 $E619   GET2RTN*
Empty and display the keyboard buffer up to a carriage return.

This is called by routine GETSCRN* 58959 ($E64F) to display the keyboard buffer at 631 ($277) up to a carriage return.

In turn, it calls routine GETQUE* 58853 ($E5E5) to loop waiting for another keyboard buffer character. It recalls that routine until it returns with a carriage return.

This routine sets various low memory pointers to the beginning and end of the character string obtained and displayed.

58959 $E64F   GETSCRN*
Obtain INPUT from screen.

Because this routine calls GET2RTN* 58905 ($E619), any keyboard buffer contents will have been displayed on the screen. This routine differentiates between keyboard and screen input, but reads the screen for both. It's called by the routine CHRIN 61966 ($F20E).

59064 $E6B8   QUOTECK*
Test for quotes and set flag.

This sets the quote mode flag at location 212 ($D4) if the ASCII value for a quote character (34) is in .A.

This quote mark check is performed when reading from the keyboard or screen, and when writing to the screen.

59077 $E6C5   SETCHAR*
Set up display of a character on the screen.

The routine PUTSCRN* at location 60074 ($EAAA) actually stores a character on the screen, but this and other routines are needed to check and set the environment before that action takes place.

The state of insert and reverse mode is tested for and accommodated, the current color nybble is retrieved from location 646 ($286), the routine at SYNPRT* 60065 ($EAA1) is called to cause the color for the screen location to be saved in the color map and the character to be displayed, and the routine calls the next routine to advance the cursor. It then exits.

59114

See the routine SCRNOUT* at location 59202 ($E742); it calls this routine.

59114 $E6EA   SCROLL
Advance the cursor on the screen, adds lines, and scroll.

After the cursor is advanced one position, the need for screen scrolling and insertion of blank lines is determined and satisfied.
See the routine SCRNOUT*, which calls this routine.

59181 $E72D   RETREAT*
Back up cursor into the previous logical screen line from the first column of the current logical line.

Refer to the routine SCRNOUT* at 59202 ($E742). SCRNOUT* calls this routine. Contrast this routine with BACKUP* at 59624 ($E8E8).

59202 $E742   SCRNOUT*
Handle characters going to the screen.

This routine is the main driver routine (in other words, calls other routines to accomplish the needed functions) for displaying characters on the screen and handling control characters. In essence, this is the heart of the screen editor.

The other screen-related routines are called as needed by this routine to perform their tasks.

The actions of this routine vary, depending on whether direct or program mode is active.

The following keys are given their meaning and screen function by this routine:

INST	CLR	RETURN
DEL	HOME	SHIFT
cursor keys	RVSON	Commodore key
color keys	RVSOFF	

59587 $E8C3   NXTLINE*
Advance cursor to the next logical screen line.

59608 $E8D8   RTRN*
Handle the carriage return key.

This routine turns off quote mode, zeros the number of outstanding inserts, turns off the reverse mode, and positions the cursor to the start of the next logical line.

59624 $E8E8   BACKUP*
Move the cursor to the end of the previous physical screen line
from the first column of a continuation of a logical line.

Contrast this routine with the routine RETREAT* at location
59181 ($E72D).

59642 $E8FA   FORWARD*
Move the cursor to the start of the next screen line if the cursor is
in the last column of the screen.

59666 $E912   COLORSET*
Set the current foreground color code.

The color code table at COLORTBL* 59681 ($E921) is scanned
for the character in .A. If found, the number of the table entry
becomes the color code at location 646 ($286).

59681 $E921   COLORTBL*
Color code key table.

The ASCII values of the keys corresponding to each color code
are:

Table 9—1. ASCII Values for Color Codes

Dec	Hex	Color	Code
144	$90	BLK	0
5	$05	WHT	1
28	$1C	RED	2
159	$9F	CYN	3
156	$9C	PUR	4
30	$1E	GRN	5
31	$1F	BLU	6
158	$9E	YEL	7

59689 $E929   CNVRTCD*
Code conversion table.

This 76-byte table has so far defied all my attempts to determine
its use. My guess is that it's a keyboard decoding table, but I haven't
found any references to it in the Kernal or BASIC.

59765 $E975   SCRL
Scroll the screen.

A lot of work needs to be done to scroll the screen. An entire
logical line needs to be scrolled off the top and a complete logical
line scrolled in from the bottom while in direct mode.

The screen line link table at 217 ($D9) must be adjusted, the screen map and color map lines must be moved, lines may need to be erased, and low memory pointers need to be updated to reflect the current cursor position. The STOP key is tested during this routine.

59886 $E9EE   OPENLIN*
Open up a blank physical line on the screen for inserts.

Inserting a blank line on the screen is another complex task for the screen editor. Scrolling downward may need to be done, in addition to all the tasks described for routine SCRL 59765 ($E975).

59990 $EA56   MOVLINE*
Move screen line.

Twenty-two bytes are moved from one screen line to another, including the corresponding color map bytes.

60014 $EA6E   SETADDR*
The address of the screen line and color line is set in memory.

A call to the routine COLORSYN* at 60082 ($EAB2) returns the color map address for the screen line. The screen line link table address is stripped of continuation flags. Both of these addresses are stored in work areas for other screen routines.

60030 $EA7E   LINPTR*
Set a pointer to the address of the start of a screen line.

This determines the screen line address from both the screen line link table and from the screen line LSB table at location 60925 ($EDFD), given the number of the screen line to obtain the address for. The pointer being set is located at 209–210 ($D1–D2).

60045 $EA8D   CLRALINE*
Blank out a physical screen line.

This moves spaces to 22 bytes of the screen map, pointed to by a temporary pointer, and sets the corresponding color map bytes to the color code representing white.

60065 $EAA1   SYNPRT*
Synchronize color to byte and store character on screen.

The cursor blink countdown at location 205 ($CD) is set to 02, and the routine COLORSYN* (60082 $EAB2) is called. This routine then falls through.

60074 $EAAA   PUTSCRN*
Store a character on the screen.

PUTSCRN* isn't an involved routine, since the routines called by SETCHAR* have done all the hard work.

On entry, .A=screen POKE code; .X=foreground color. These are placed in the appropriate screen map and color map positions pointed to by temporary pointers.

60082 $EAB2   COLORSYN*
The address of the color map byte for screen map byte is found.

The screen map byte pointer in location 209–210 ($D1–D2) is converted to a color map byte pointer in 243–244 ($F3–F4).

See location 217–241 ($D9–F1) for an example of using this routine.

60095 $EABF   IRQ
IRQ interrupt handler.

When the 6502 encounters an IRQ interrupt, it jumps off the vector at 65534 ($FFFE) which points to the routine IRQROUT* at location 65394 ($FF72). That routine will jump off the RAM vector at location 788–789 ($314–315) to reach this routine if the interrupt was not caused by a RUN/STOP–RESTORE key.

Normally, this routine is entered every 1/60 second because of the interrupts generated by timer 1 of VIA2. The routine calls UDTIM 63284 ($F734) to update the jiffy clock at location 160–162 ($A0–A2) and to obtain the current keypress from 37167 ($912F) and store it in location 145 ($91). The cursor is blinked and the tape motor is turned off if appropriate. The routine SCNKEY 60190 ($EB1E) is called to do the keyboard scan. Refer to that location for more details.

Tape I/O interferes with accurate clocking and the RUN/STOP key, since the vector at 788–789 ($314–315) is replaced during the tape operation.

See the notes at location 788–789 ($314–315).

60190 $EB1E   SCNKEY
Scan the keyboard for keypresses using 6522 VIA2.

The A and B ports of VIA2 (VIA2PB* ($9120) and VIA2PA1* ($9121)) are examined for keypresses using the techniques outlined at those locations. Normally the table ASCII values of keys located at NORMKEY* 60510 ($EC5E) are used to decode the key. Location 245–246 ($F5–F6) points to this table or any alternate table used. Location 653 ($28D) is set to indicate any SHIFT, Commodore, or CTRL key pressed. Location 203 ($CB) is set to the matrix coordinate of the current key pressed; a value of 64 is set if no keys are pressed.

The vector at 655 ($28F) is used to call SETKEYS* 60380 ($EBDC). Then the routine checks the repeater flag at 650 ($28A) and repeats the key if the time intervals set in 651 and 652 ($28C) allow.

If the count of characters in the keyboard buffer (location 198, $C6) is less than ten, the ASCII value of the key is placed in the keyboard buffer at location 631 ($277) and the count is incremented.

At exit, VIA2 is left in a mode to sense the RUN/STOP key quickly.

60380 $EBDC   SETKEYS*
Set keyboard decode table address in 245–246 ($F5–F6)

This routine is called by SCNKEY at 60190 ($EB1E) to determine the correct keyboard decoding table to be used, based upon the SHIFT key flags at location 653 ($28D). Location 657 ($291), the flag to enable/disable the SHIFT/Commodore key switch, is checked when selecting the proper table. The VIC chip register at 36869 ($9005) is set to reflect any change in character sets. This routine has many NOP instructions which indicate that it evidently had additional functions or tests at one time. The Commodore 64 version has been cleaned up.

The pointers in KEYVCTRS* 60486 ($EC46) are used as the current table address to be stored in location 245–246 ($F5–F6). See the following routine for the possible table values.

60486 $EC46   KEYVCTRS*
Keyboard decode table addresses.

The routine SETKEYS* 60380 ($EBDC) uses this table to store the current keyboard decode table address in locations 245–246 ($F5–F6), based upon the shift flags in 653 ($28D).

Table 9–2. Keyboard Decoding Table

Shift flags in Location 653 ($28D)				Decode table address set in 245–246	
Dec	Binary	Keys being pressed			
0	00000000	none		60510	$EC5E
1	00000001	SHIFT		60575	$EC9F
2	00000010	Commodore key		60640	$ECE0
3	00000011	SHIFT + Commodore		60510	$EC5E
				or 60575	$EC9F
				(until pressed again)	
4	00000100	CTRL		60835	$EDA3
5	00000101	SHIFT + CTRL		60835	$EDA3
6	00000110	Commodore + CTRL		60835	$EDA3
7	00000111	SHIFT + CTRL + Commodore		60835	$EDA3

60510 $EC5E   NORMKEYS*
Table used for decoding unshifted keys into ASCII.

Also used for CTRL plus SHIFT key decoding of keys.

Refer to Table 1-1, Keycode Values at Location 197 ($C5), for the ASCII value for each key.

60575 $EC9F   SHFTKEYS*
Table used for decoding SHIFTed keys into ASCII.

Also used for CTRL plus Commodore key decoding of keys.

60640 $ECE0   LOGOKEYS*
Table used for decoding Commodore SHIFTed keys into ASCII.

60705 $ED21   CHARSET*
Used to set uppercase/graphics character set.

The VIC chip control register at location 36869 ($9005) is set to reference the uppercase/graphics character set.

60720 $ED30   GRAPHMODE*
Set the environment specified by graphics control characters.

This is called by the routine SCRNOUT* 59202 ($E742) to implement the graphic modes specified by certain ASCII control characters.

Table 9—3. Graphics Controls

CHR$	Effect
14	Select lowercase mode
142	Select uppercase mode
8	Disable shift to another case
9	Enable shift to another case

60763 $ED5B   WRAPLINE*
This routine is called by SCROLL 59114 ($E6EA) to mark the next physical screen line in the screen line link table as *no continuation* of the current line.

60777 $ED69   WHATKEYS*
Apparently unused keyboard decoding table.

From the values that are stored here, the table was perhaps to be used as a CTRL and Commodore key decoding table, but it's now unable to be referenced. The Commodore 64 does not have this table.

60835 $EDA3   CTRLKEYS*
Table used for decoding CTRL SHIFTed keys into ASCII.

This is also used when any other SHIFT key (SHIFT and/or Commodore key) is pressed while the CTRL key is pressed.

60900 $EDE4   VICINIT*
Initial values for VIC chip registers.

The routine INITVIC* 58819 ($E5C3) loads these values into the VIC chip registers at 36864–36879 ($9000–900F) when power-on/reset or RUN/STOP–RESTORE is being processed.

60916 $EDF4   RUNTB
LOAD and RUN words for SHIFT and RUN keys.

The characters LOAD and RUN (each followed by a carriage return) are placed into the keyboard buffer at location 631 ($277) by the routine GETQUE* when the left SHIFT key and RUN key are pressed at the same time.

60925 $EDFD   LDTB2
Screen line link table LSB of lines in screen map.

This table is used by the routine LINPTR* 60030 ($EA7E) to obtain the address of any given screen line and place it in location 209–210 ($D1–D2). The table contains 23 entries; the first contains 0, and each succeeding entry is incremented by 22.

See location 217–241 ($D9–F1) for an example of using this table in conjunction with the RAM located screen line link table.

60948 $EEI4   TALK
Serial: send talk with attention.

Bit 6 of the device number is turned on to instruct the serial device to talk to the VIC-20. The routine then falls through to the next routine, skipping the first instruction.

The use of this routine is discussed in the *VIC-20 Programmer's Reference Guide*.

60951 $EEI7   LISTEN
Serial: send listen with attention.

Bit 5 of the device number is turned on to instruct the serial device to listen to the VIC-20. The routine then calls RSPAUSE* 61792 ($F160) to disable any RS-232 NMIs and falls through to the next routine.

The use of this routine is also discussed in the *VIC-20 Programmer's Reference Guide*.

60956 $EE1C   LIST1
Serial: prepare to send serial command with attention.

The character to be sent is saved in location 149 ($95), and a handshaking protocol sequence is sent to the device so that it will listen to the command about to be sent. This routine then falls through.

An alternate entry point to this routine is at 60992 ($EE40). Entry at this point causes a send *without* the attention bit.

61001 $EE49   SRSEND*
Serial: send command or data to serial devices.

This sends the data or command to the addressed serial device, checking to see that it acknowledges receipt of the transmission. Location 165 ($A5) is used to count down the bits to be sent, which are in location 149 ($95).

61108 $EEB4   SRBAD*
Serial: set ST for timeout or DEVICE NOT PRESENT.

Depending on which instruction the routine is entered, it sets the appropriate ST bits at location 144 ($90) by calling READIOST* 65130 ($FE6A).

61120 $EEC0   SECOND
Serial: send secondary address after listen command.

Once the serial device has been commanded to listen with the LISTEN routine 60951 ($EE17), the secondary address is sent to it after ORing with 96 ($60). The routine LIST1 60956 ($EE1C) is called to send the secondary address, then this routine falls through.

The use of this routine is discussed in the *VIC-20 Programmer's Reference Guide*.

61125 $EEC5   SCATN
Serial: clear attention.

This routine removes the serial attention bit from VIA1 port A, bit 7. This is found in the VIA1PA2* routine at 37151 ($911F).

61134 $EECE   TKSA
Serial: send secondary address after talk command.

Once the serial device has been told to talk (the routine TALK does this), the secondary address is sent to it. The acknowledgment from the device is awaited. Routines LIST1 60956 ($EE1C), SEROUT0* 58537 ($E4A9), SCATN 61125 ($EEC5), SRCLKHI* 61316 ($EF84), and SERGET* 58546 ($E4B2) are called to perform the task.

You can refer to the *VIC-20 Programmer's Reference Guide* for details on how to use this routine.

61156 $EEE4   CIOUT
Serial: send a byte on the serial line.

This routine is called after the serial device has been instructed to listen (by the routine LISTEN) and after any needed secondary address has been sent with SECOND 61120 ($EEC0).

The character to be sent into location 149 ($95) is set by this routine, which sends it via a call to SRSEND* 61001 ($EE49) when this routine is next called, or when UNLSN 61188 ($EF04) is called.

Look at the *VIC-20 Programmer's Reference Guide* for details on how to use this routine.

61174 $EEF6   UNTLK
Serial: send untalk command to serial devices.

All talking devices are instructed to stop talking when this is issued. The routine actually falls through to the next routine to perform much of the task.

The use of this routine is discussed in the *VIC-20 Programmer's Reference Guide*.

61188 $EF04   UNLSN
Serial: send unlisten command to serial devices.

All listening devices are told to stop listening.

Refer to the *VIC-20 Programmer's Reference Guide* for a discussion on how to use this routine.

61209 $EF19   ACPTR
Serial: receive byte from serial device.

The routines TALK 60993 ($EE41) and TKSA 61134 ($EECE) are called prior to this routine to initiate and send the dialog.

This routine is discussed in the *VIC-20 Programmer's Reference Guide*.

61316 $EF84   SRCLKHI*
Serial: set clock line high.

This turns off bit 1 of VIA2PCR* 37164 ($912C), causing CA2 to be set to a low value. It's inverted to high by the circuitry.

This routine is called by several other routines to assist in the serial handshaking sequences.

61325 $EF8D   SRCLKLO*
Serial: set clock line low.

Bit 1 of VIA2PCR* 37164 ($912C) is turned on by this routine, causing CA2 to be set to a high value, which is inverted to low by the circuitry.

This is called by other routines to assist in the serial handshaking sequences.

61334 $EF96   WAITABIT*
Serial: delay one millisecond.

Called by the routine LIST1 60956 ($EE1C), this routine uses the second timer in VIA2 to time a one-millisecond delay, looping in instructions that test for its expiration.

61347 $EFA3   RSNXTBIT*
RS-232: send the next bit (NMI continuation routine).

This checks location 180 ($B4) to determine if all the bits for the byte in 182 ($B6) have been sent. It calls RSPRTY* 61375 ($EFBF) to calculate the parity when the entire byte has been sent. Location 181 ($B5) is set to 1 or 0 for the next bit to be sent. Location 189 ($BD) is used as a parity work byte. When an RS-232 interrupt occurs, VIA1 CB2 will be set high or low, depending on the value in location 181 ($B5).

61375 $EFBF   RSPRTY*
RS-232: calculate parity and stop bits value.

This is called by the above routine RSNXTBIT* to calculate the parity and number of stop bits for a byte. Location 189 ($BD) is used to determine which state the parity bit should have (1 or 0), depending on the parity option chosen. Even or odd parity specifications send the corresponding parity bit. For instance, *even* will send a one parity only if the number of bits in the byte is *odd*, therefore making it *even*. Mark parity always transmits parity bit of 1. Space parity always transmits a 0 parity bit.

61416 $EFE8   RSSTOPS*
RS-232: transmit stop bits.

Depending on the number of stop bits requested by the user, this routine is called once or twice to generate and send either one or two stop bits. This is accomplished by calling an alternate entry point of RSNXTBIT* 61347 ($EFA3). Location 180 ($B4) is used to calculate the needed stop bit configuration.

61422 $EFEE   RSNXTBYT*

RS-232: prepare the next byte to be sent from send buffer.

Called by routines RSNXTBIT* 61347 ($EFA3) and RSNMISET* 61710 ($F10E), this routine obtains the next byte to be sent from the transmit buffer at the top of RAM created when the RS-232 device was opened.

Because of an incorrect address in the code, this routine always believes that *Clear To Send* and *Data Set Ready* are present.

The number of bits to send is placed in 180 ($B4), while locations 181 ($B5) and 189 ($BD) are zeroed. The byte to be sent is placed in 182 ($B6). The transmit buffer pointer at 669 ($29D) is kept current.

61462 $F016   RSMISSNG*

RS-232: set *Clear To Send* or *Data Set Ready Missing* status.

Meant to set the RS-232 status byte at 663 ($297) when either of these two conditions is met, it seems that only *Data Set Ready Missing* will be set because of the coding errors at OPENRS 62738 ($F512) and RSNXTBYT* 61428 ($EFF4).

This is called when an error is detected when opening RS-232 or preparing the next byte to be sent.

61479 $F027   RSCPTBIT*

RS-232: compute desired word length bit count.

When opening the RS-232 device, this routine is called to calculate the correct number of data bits, based on the number specified in the control register at location 659 ($293).

61494 $F036   RSINBIT*

RS-232: receive an input bit (NMI driven).

Called by the routine RSNMI* 65246 ($FEDE) when a timer 2 interrupt is sensed, this routine calls RSSTRBIT* to determine if a start bit was received. If so, then bit 0 of 167 ($A7) will contain the inbound bit from the RS-232 device. Location 170 ($AA) is built bit by bit, and then RSINBYTE* 61551 ($F06F) is called to store the new byte in the receive buffer.

61515 $F04B   RSSTPBIT*

RS-232: determine if all the stop bits have been received yet.

If the proper number of stop bits has not yet been received, the routine exits. Otherwise, it falls through to the next routine. Locations 167–168 ($A7–A8) are used to determine the number of stop bits received.

61531 $F05B   RSPREPIN*
RS-232: prepare to receive the next input byte.

    The routine enables VIA 1 CB1 interrupts, disables timer 2 interrupts, and sets 169 ($A9) to indicate that a start bit is being sought.

61544 $F068   RSSTRBIT*
RS-232: check for start bit in receive mode.

    Location 169 ($A9) is tested to see if a start bit has been received. If not, this goes to the routine RSPREPIN* 61531 ($F05B).

61551 $F06F   RSINBYTE*
RS-232: put constructed byte into receive buffer.

    If the receive buffer pointed to by location 668 ($29C) has room for a character, this stores the byte in the buffer and falls through to the next routine. Otherwise, it forgets the just-received character and goes to RSOVERUN at location 61602 ($F0A2).

61579 $F08B   RSINPRTY*
RS-232: parity checking of the input byte.

    The parity option selected, if any, that was stored in 167 ($A7) is compared to the input byte parity stored in location 171 ($AB). If correct, the routine RSSTPBIT* 61515 ($F04B) is branched to. Otherwise, a branch is made to the RSPRTYER* routine at 61597 ($F09D).

61597 $F09D   RSPRTYER*
RS-232: parity error on input byte.

    This loads .A with the error code ($01) and goes to routine RSINERR* at location 61610 ($F0AA).

61602 $F0A2   RSOVERUN*
RS-232: buffer overrun on input byte.

    This routine loads .A with the error code ($04) and goes to routine RSINERR* 61610 ($F0AA).

61605 $F0A5   RSBREAK*.
RS-232: break detected on input.

    .A is loaded with the error code ($80) and this routine goes to RSINERR* at 61610 ($F0AA).

61608 $F0A8   RSFRAMER*
RS-232: framing error on input.

    This loads .A with the error code ($02) and goes to routine RSINERR* 61610 ($F0AA).

263

61610 $F0AA   RSINERR*
RS-232: set input error status and continue.

    After the RS-232 error status is posted in 663 ($297) for the user
to correct, the next input byte is prepared for by jumping to
RSPREPIN* 61531 ($F05B).

61625 $F0B9   RSDVCERR*
RS-232: ILLEGAL DEVICE message for LOAD or SAVE.

    An ILLEGAL DEVICE NUMBER message is issued if a LOAD or
SAVE is attempted on an RS-232 device. The routine FILEMSG* at
location 63358 ($F77E) is called to issue the message.

61628 $F0BC   RSOPNOUT*
RS-232: open an RS-232 channel for output.

    This routine is called by CHKOUT 62217 ($F309) to prepare an
RS-232 output channel. The routine performs the necessary x-line
handshake and simply returns for three-line handshaking.

61677 $F0ED   RSOUTSAV*
RS-232: store a character in the transmit buffer.

    If there is no room in the transmit buffer, the routine loops until
the character can be stored in the buffer. Interrupts cause sending
routines to remove bytes from the buffer and send them along. The
transmit buffer at location 670 ($29E) is kept current.

    This routine falls through to the next routine if transmission has
not been started; otherwise,it returns.

61698 $F102   RSPREPOT*
RS-232: set up NMI for transmission.

    This prepares timer 1 of VIA 1 to interrupt at the needed time to
implement the selected baud rate. Location 665–666 ($299–29A)
contains the bit timing value for the baud rate.

    The routine RSNXTBYT* 61422 ($EFEE) is called to begin
transmission.

61718 $F116   RSOPNIN*
RS-232: open an RS-232 channel for input.

    Called by routine CHKIN 62151 ($F2C7), this routine enables
timer 2 and CB1 interrupts on VIA1 and exits for three-line mode or
full duplex x-line handshaking. Half duplex x-line handshaking
causes a check for dataset-ready and clear-to-send to turn off, turns
off request-to-send, and waits for data-terminal-ready to come on
before proceeding.

61775 $F14F   RSNXTIN*
RS-232: retrieve the next character from the receive buffer.

GETIN at 61941 ($F1F5) calls this routine for the next RS-232 input byte. The receive buffer is emptied by this routine, returning a 0 in .A if no characters are in the buffer.

61792 $F160   RSPAUSE*
RS-232: check that serial and tape are idle, to protect from RS-232.

Tape and serial read/write routines call this routine to insure that RS-232 NMI sequences are not currently active, which would hinder tape and serial timing. If RS-232 is active, the routine loops until the RS-232 sequence is finished.

61812 $F174   KMSGTBL*
Table of Kernal messages.

The last letter of each message has the high order bit on. To cause a particular message to be printed, .Y is loaded with the index of the message within this table, then the routine KMSGSHOW* at 61926 ($F1E6) is branched to. If an I/O ERROR message is to be displayed, the FILEMSG routine at 63358 ($F77E) is branched to at the appropriate entry point to cause the desired error number to follow the message.

Table 9—4. Kernal Error Messages

Index Dec	Hex	Message
00	$00	(CR) I/O ERROR
		This message is followed by a code number:
		1 TOO MANY FILES
		2 FILE OPEN
		3 FILE NOT OPEN
		4 FILE NOT FOUND
		5 DEVICE NOT PRESENT
		6 NOT INPUT FILE
		7 NOT OUTPUT FILE
		8 MISSING filename
		9 ILLEGAL DEVICE NUMBER
12	$0C	(CR) SEARCHING FOR
27	$1B	(CR) PRESS PLAY ON TAPE
46	$2E	PRESS RECORD & PLAY ON TAPE
73	$49	(CR) LOADING
81	$51	(CR) SAVING
89	$59	(CR) VERIFYING
99	$63	(CR) FOUND
106	$6A	(CR) OK

BASIC preempts the Kernal error message I/O ERROR in program mode. BASIC has its own error messages and prefers them over the Kernal message of I/O ERROR followed by an error number. See the list of BASIC messages at location 49566 ($C19E).

See location 157 ($9D) for the Kernal control/error message selection indicator that enables or disables Kernal messages. The vector at 65424 ($FF90) causes SETMSG at 65126 ($FE66) to reset location 157 ($9D) to the value in .A. You could set this yourself with a POKE statement.

61922　　　　$F1E2　　　　SPMSG
Display LOADING or VERIFYING if control messages wanted.

This routine is called by 63082 ($F66A) to display the appropriate message. It checks the flag at 157 ($9D) and skips the message if Kernal control messages are turned off. These would be turned off by BASIC in program mode; these messages are issued only in direct mode.

This falls through to the next routine if the message is to be displayed.

61926　　　　$F1E6　　　　KMSGSHOW*
Print Kernal control messages.

This is called by any routine wanting to display a Kernal control message. It leaves checking of location 157 ($9D), the Kernal message control, to the calling routine. .Y received is used as an index into the message table at address 61812 ($F174).

61941　　　　$F1F5　　　　GETIN
Routing routine for obtaining a character of input data.

The next input device character (or 0) will be placed in .A when this routine is called. Contrast this with the routine CHRIN 61966 ($F20E).

This routine routes the actual processing of this function to several different routines, based on the device currently being used as indicated in location 153 ($99).

The vector at 65508 ($FFE4) should be used to access this routine so that the 810 ($32A) RAM vector will be used.

If the input device is not the keyboard or RS-232, this goes to CHRIN. If the input device *is* for the keyboard and location 198 ($C6) indicates that the keyboard buffer is empty, then this simply returns. If there are characters in the buffer, the LP2 at 58831 ($E5CF) is called.

For RS-232 devices, routine RSNXTIN* 61775 ($F14F) is called. BASIC READ, GET, and INPUT call this routine via the vector at 810 ($32A).

The use of this routine is discussed in the *VIC-20 Programmer's Reference Guide*.

61966 $F20E   CHRIN
Input characters from current input device.

The result of calling this routine is that one byte of data is obtained from the input device and placed into .A. The keyboard or screen can return up to 88 bytes into the BASIC input buffer at 512 ($200), one byte being returned by each call to this routine. Contrast this with the routine GETIN 61941 ($F1F5).

This routine routes the actual processing of this function to several different routines, based on the device that is currently being used as indicated in location 153 ($99).

The vector at 65487 ($FFCF) should be used to access this routine so that the 804 ($324) RAM vector will be used.

For the keyboard or screen, the GETSCRN* routine at 59124 ($E64F) is used.

For serial devices, the routine ACPTR 61209 ($EF19) is used; refer to the interesting feature before the call in routine CHRINSR* 62052 ($F264).

For the RS-232 device, CHRINRS* 62063 ($F26F) is called.

Tape device handling falls through to the next routine.

Refer to the *VIC-20 Programmer's Reference Guide* for a short discussion on the use of this routine.

62000 $F230   CHRINTP*
Obtain a byte from the tape buffer.

This calls CHRINTP2* 62032 ($F250) to point to the byte in the tape buffer or request the next tape block. It tests for end of data and sets location 144 ($90) to 64 ($40) if so.

62032 $F250   CHRINTP2*
Load .A with next tape character, getting block when needed.

This routine calls JTP20 63626 ($F88A) to increment the pointer in 166 ($A6) to the next byte, calling JTP20 63680 ($F8C0) to get the next tape block if needed.

62052 $F264   CHRINSR*
Obtain a byte from the serial line.

If ST in location 144 ($90) is other than 0, this routine returns a carriage return as a data character in case you ignored EOI on serial. Otherwise, this jumps to the routine ACPTR 61209 ($EF19).

62063

62063 $F26F   CHRINRS*

Obtain a byte from the RS-232 device.

    This loops in a call to RSNXTIN* 61775 ($F14F) until a nonzero is returned.

    Because of the possibility of an endless loop here, CHRIN is not recommended for RS-232 devices. See page 255 of the *VIC-20 Programmer's Reference Guide*. GET# or the routine GETIN via the vector at 65508 ($FFE4) should be used instead.

62074 $F27A   CHROUT

Output character to current output device.

    Calling this routine sends one byte of data from .A to the output device.

    This routine routes the actual processing of this function to several different routines, based on the device that is currently being used as indicated in location 154 ($9A).

    The vector at 65490 ($FFD2) should be used to access this routine so that the 806 ($326) RAM vector will be used.

    For screen output, the routine SCRNOUT* 59202 ($E742) is used; the routine CIOUT at 61156 ($EEE4) is used for serial output; and for RS-232 output, the routine RSOUTSAV* 61677 ($F0ED) is used.

    Tape output is handled by falling through to the next routine.

62096 $F290   CHROUTTP*

Output a character to tape.

    The character to be sent on an open channel is stored in location 158 ($9E).

    This routine calls JTP20 63680 ($F8C0) to update the tape buffer index in location 166 ($A6) to the next byte. When the buffer is full, the routine WBLK 63715 ($F8E3) is called to write the block.

    Check the *VIC-20 Programmer's Reference Guide* for details on the use of this routine.

62151 $F2C7   CHKIN

Open .X file number channel for input.

    Once a file has been opened, this routine sets location 153 ($99), the input device, to the associated (.X) file device number. It also checks that the file is opened, that it was opened as an input file, and that the device is present.

    For serial devices, both an instruction to talk and the secondary address are sent to the device.

    For RS-232, this routine calls RSOPNIN* 61718 ($F116) to open the channel.

The jump at 65478 ($FFC6) should be used to access this routine so that the 798 ($31E) RAM vector will be used.

62217 $F309   CHKOUT
Open .X file number channel for output.

Once a file has been opened, this routine sets 154 ($9A), the output device, to the associated (.X) file device number. It checks that the file is opened, that it was opened as an output file, and that the device is present.

For serial devices, a command to listen, as well as the secondary address, is sent to the device.

The routine RSOPNOUT* 62628 ($F0BC) is called to open the channel for RS-232.

The jump at 65481 ($FFC9) should be used to access this routine so that the 800 ($320) RAM vector will be used.

62282 $F34A   CLOSE
Close logical file number in .A.

The FNDFLNO* routine at 62415 ($F3CF) is called by this routine to find the file information; SETFLCH* 62431 ($F3DF) is called to get the index into the file tables.

If RS-232, the buffers are deallocated and the routine MEMTOP 65139 ($FE73) is called to return the two 256-byte areas to availability.

If writing to tape, an end of file 0 is put into the end-of-tape buffer 828 ($33C) and the buffer is written. If the secondary address has bit 1 on, an end-of-tape header with tape I.D. 5 is also written.

Locations 152 ($98), the number of open files; 601 ($259), file number table; 611 ($263), the device number table; and 621 ($26D), secondary address table, are updated.

The jump at 65475 ($FFC3) should be used to access this routine so that the 796 ($31C) RAM vector will be used.

62415 $F3CF   FNDFLNO*
Find file number (.X) in file table at 601 ($259).

If the file number is found, .P bit 1 is set so that BEQ is true. This routine is called by the routines CHKIN, CHKOUT, CLOSE, and OPEN.

62431 $F3DF   SETFLCH*
Set file characteristics of file (.X) into 184–186 ($B8–$BA).

File, secondary address, and device number are stored in the indexes at 184–186 ($B8–BA) from the entries in the tables:

601 ($259) LAT File number table
611 ($263) FAT Device number table
621 ($26D) SAT Secondary address table

62447 $F3EF   CLALL
Abort all open files (with no close).

This sets location 152 ($98) to indicate no open files and falls
through to the next routine.

The jump at 65511 ($FFE7) should be used to access this routine
so that the 812 ($32C) RAM vector is used.

62451 $F3F3   CLRCHN
Abort all open channels.

Locations 153 ($99), the input device, and 154 ($9A), the output
device, are reset to keyboard (0) and screen (3).

If serial devices were on open channels, this routine calls
UNLSN 61188 ($EF04) and/or UNTLK 61174 ($EEF6).

Pressing the STOP key calls this routine.

The jump at 65484 ($FFCC) should be used to access this rou-
tine so that the 802 ($322) RAM vector is used.

62474 $F40A   OPEN
Open a logical file, file number in 184 ($B8).

If file number 0 is used, this gets a FILE NOT AN INPUT FILE
error. It also calls FNDFLNO* 62415 ($F3CF) to insure that the file is
not already open and checks that this open doesn't exceed ten open
files.

File number, secondary address, and device number are stored
in the tables:
601 ($259) LAT File number table
611 ($263) FAT Device number table
621 ($26D) SAT Secondary address table

If a serial device is being opened, the routine SERNAME* at
62613 ($F495) is called; if an RS-232 device is being opened,
OPENRS* 62663 ($F4C7) is called.

When opening a file to be read from tape, this calls FNDHDR*
at 63591 ($F867) to find the specified file, or calls 63407 FAH
($F7AF) for the next file on the tape.

When opening an output file to tape, the routine TAPEH 63463
($F7E7) is called to write the new file header (I.D. 4).

The jump at 65472 ($FFC0) should be used to access this rou-
tine so that the 794 ($31A) RAM vector will be used.

62613 $F495   SERNAME*

Send secondary address and filename to serial device.

The secondary address stored in location 185 ($B9) is sent to the serial device stored in 186 ($BA) by calling LISTEN 60951 ($EE17) and SECOND 61120 ($EEC0).

DEVICE NOT PRESENT is indicated if the return status 144 ($90) has the high order bit on.

The filename pointed to by 187 ($BB) is sent by calling CIOUT 61156 ($EEE4) for each character.

62663 $F4C7   OPENRS*

RS-232: open RS-232 device.

Up to four characters of the specified filename are stored in:

659 ($293)	RS-232 control register
660 ($294)	RS-232 command register
661–662 ($295–296)	RS-232 nonstandard bit timing

The routine RSCPTBIT* 61479 (F027) is called to compute the desired word length bit count, which is stored in location 664 ($298).

The clock time per bit is computed and stored in 665 ($299) using the table at 65372 ($FF5C). The resulting values are copied to 37140–1 ($9114–5) and 37144–5 ($9118–9) when needed. These are VIA 1's timers.

For x-line handshaking, an erroneous (?) test for dataset ready at 37152 ($9120) is performed, which will always be true.

Two RS-232 buffers are allocated from the top-of-user-RAM space by calling MEMTOP 65139 ($FE73) to read and later set the pointers for top of RAM, after the top has been decremented by two pages. RS-232 pointers at 247–248 ($F7–F8), 249–250 ($F9–FA), and 667–669 ($29B–29D) are initialized.

62786 $F542   LOAD

Load (or verify) to RAM from device number specified in 186 ($BA).

The .X and .Y input parameters are the LSB/MSB address of the starting RAM to be loaded or compared. These are stored in 195–196 ($C3–C4) and a branch is performed off the vector in 816 ($330), normally returning to the next instruction of the routine. .A input parameter is 0 for LOAD or 1 for VERIFY. Keyboard, screen, and RS-232 are illegal devices.

Serial device activity is routed to the LOADSER* routine at 62812 ($F55C), and tape functions are routed to the routine LOADTP* 62929 ($F5D1).

The jump at 65493 ($FFD5) should be used to access this routine to use the RAM vector at 816 ($330).

62812 $F55C   LOADSER*
Load or verify RAM from a serial device.

Serial devices are sent the filename and a secondary address of 0, with some command bits on, and respond with a FILE NOT FOUND condition or a pointer of the location of where the data was saved from. This pointer at location 174–175 ($AE–AF) is overlaid by 195–196 ($C3–C4) if a relocatable load was specified. Otherwise, it is used as the address for the start of the load. The file from disk is then loaded to RAM.

62929 $F5D1   LOADTP*
Load or verify RAM from tape.

The address to load the data to must be greater than 512 ($200) or the ILLEGAL DEVICE message will display.

PRESS PLAY is requested, if needed. Either the next filename is searched for by FAH 63407 ($F7AF) or the specified file is searched for by the routine FNDHDR 63591 ($F867). See the explanation of the tape header format at location 828 ($33C). The first byte of the tape header is tested (the tape I.D.). If the tape I.D. is 3 (a nonrelocatable tape) or if the I.D. is 1 and the secondary address in 185 ($B9) is nonzero, the starting address for the LOAD comes from the second and third bytes in the tape header. However, if the tape I.D. is 1 and the secondary address in 185 ($B9) is 0, it uses the starting address specified in .X and .Y passed to the load routine.

The ending address of the LOAD is calculated from the starting address plus the length saved in the tape header.

The message LOADING or VERIFYING and the filename are displayed. LDBLK* 63689 ($F8C9) is called to read in the two program blocks.

The routine TSTOP at 63819 ($F94B) will be called to test for the STOP key.

63047 $F647   SRCHING*
Display SEARCHING message for tape device.

If location 157 ($9D) allows Kernal control messages, the message is displayed and falls through to the next routine.

63065 $F659   FILENAME*
Display the filename.

The filename pointed to by location 187–188 ($BB–BC) is displayed by calling the CHROUT routine at 62074 ($F27A) for each character.

63082 $F66A   LDVRMSG*
Display LOADING or VERIFYING message.

The contents of location 147 ($93) are used to determine which message is appropriate. This routine calls SPMSG at 61922 ($F1E2) to do the actual displaying.

63093 $F675   SAVE
Save RAM to device number specified in 186 ($BA).

.X and .Y are stored in 174–175 ($AE–AF) as the ending address. The zero page index in .A is used to move the starting address to location 193–194 ($C1–C2). The vector at 818 ($332) is jumped from, normally returning to the next routine.

The jump at 65496 ($FFD8) should be used to access this routine or the RAM vector at 818 ($332) for the next routine.

63122 $F692   SAVESER*
Save RAM to serial device.

An RS-232, screen, or keyboard SAVE displays ILLEGAL DEVICE message.

A tape SAVE is routed to the routine SAVETP* 63217 ($F6F1).

The rest of this routine handles serial devices. A check is made to insure that a filename is present, SAVING along with the filename is displayed, and the starting address of the SAVE is sent to the disk for recording with the data or program. The RAM between the start and end address is sent to disk by calling the CIOUT routine at 61156 ($EEE4), with a test for the STOP key after every character sent.

63217 $F6F1   SAVETP*
Save RAM to tape.

The tape buffer pointed to by 177–178 ($B1–B2) is checked to insure that it is not located below 512 ($200); otherwise, an ILLEGAL DEVICE error is displayed. PRESS RECORD AND PLAY is displayed if needed. SAVING and any filename is also displayed. The tape header I.D. is determined from the secondary address in 185 ($B9) and placed in the tape header. An even secondary address results in a tape I.D. of 1 for a relocatable program, while an odd secondary address results in a tape I.D. of 3 for a nonrelocatable program. The routine TAPEH 63463 ($F7E7) is called to write the tape header with the start and end address of the area saved, as well as the filename.

Routine WBLK 63715 ($F8E3) is called to write a three-second leader onto the tape, followed by the save area in the format of two identical blocks separated by a short interblock leader. Finally, if the

secondary address has bit 1 on, an I.D. 5 (end of tape) header is written by calling TAPEH.

Routine TSTOP 63819 ($F94B) is called to test for the STOP key.

Figure 3-1 at location 833–1019 ($341–3FD) shows a typical tape format. Refer to that diagram for details on each block written to tape.

63272 $F728   SAVING*
Display SAVING message.

If location 157 ($9D) allows Kernal control messages, this is displayed by calling KMSGSHOW* 61926 ($F1E6); the routine FILENAME* 63065 ($F659) is called to display a filename.

63284 $F734   UDTIM
Increment the jiffy clock at 160–162 ($A0–A2).

This updates the clock and resets it to 0 after 24 hours. Location 162 ($A2) is incremented every .01667 second (one jiffy; 1/60 second).

Location 37167 ($912F) is stored in 145 ($91) for STOP key testing purposes.

The IRQ routine 60095 ($EABF) calls this routine every jiffy.

Tape I/O interferes with accurate clocking and STOP key testing, although the routine TAPE periodically calls this routine.

The NMI* 65193 ($FEA9) routine also calls this routine.

The jump at 65514 ($FFEA) should be used to access this routine.

63328 $F760   RDTIM
Put jiffy clock from 160–162 ($A0–A2) into .Y, .X, and .A.

This routine is called from LET 51621 ($C9A5) when TI$ is set in BASIC.

The jump at 65502 ($FFDE) should be used to access this routine.

63335 $F767   SETTIM
Set time into jiffy clock 160–162 ($A0–A2) from .Y, .X, and .A.

This is called from LET 51621 ($C9A5) when TI$ is set in BASIC and should be accessed with the jump at 65499 ($FFDB).

63344 $F770   STOP
Check for STOP key in ($91), purge keyboard queue and channels if so.

Location 145 ($91) is checked for a value of 254 ($FE). The

UDTIM routine at 63284 ($F734) stores the current keypress in that location.

Bit 1 of .P (or location 783 ($30F) if SYS 63344 is used) will be on if the STOP key was found; otherwise, .A contains the current keypress value from the STOP key keyboard row.

Tape I/O interferes with accurate clocking and the STOP key test, although the routine TAPE periodically calls this routine.

The jump at 65505 ($FFE1) should be used to access this routine to use the RAM vector at 808 ($328).

63358 $F77E FILEMSG*

I/O error file error message handler.

Multiple entry points allow any particular error number to be displayed, depending on the point of entry.

If location 157 ($9D) allows Kernal error messages, the message is displayed by calling KMSGSHOW* 61926 ($F1E6) and then CHROUT 62074 ($F27A) to display the error number.

See location 61812 ($F174) for the assigned error numbers.

63407 $F7AF FAH

Tape: find next tape header, .X back contains header I.D. number.

This finds the next tape header by calling RDTPBLKS* 63680 ($F8C0) to load the next header into the tape buffer. If the first byte in the buffer is not equal to 1, 3, 4, or 5, it keeps reading tape blocks until a header is found.

If the header is not an I.D. 5 header, and location 157 ($9D) allows Kernal control messages, the Found message and the filename are displayed.

See location 828 ($33C) for an explanation of the tape header I.D. meanings.

Routine TSTOP 63819 ($F94B) is called to test for the STOP key.

63463 $F7E7 TAPEH

Tape: build an output tape header in the tape buffer area.

.A into this routine contains the tape header I.D.

The tape buffer is first cleared to all spaces. This routine then fills the tape buffer (pointed to by 178–179 ($B2–B3)) with the header I.D., the starting and ending address of the area saved, and the filename. The latter is pointed to by location 187 ($BB) for the length specified in address 183 ($B7).

This routine calls LDAD1 63572 ($F854) to temporarily (for this routine only) reset both 193–94 ($C1–C2) to the start of the tape buffer and 174–175 ($AE–AF) to the end-of-tape buffer. It also calls the routine WBLK 63715 ($F8E3) to write a ten-second leader and

the tape header in the buffer onto the tape in the format of two identical blocks, separated by a short interblock leader.

TSTOP 63819 ($F94B) is called to test for the STOP key.

63565 $F84D   TPBUFA*

Tape: load tape buffer address from 178-179 ($B2-B3) into .X and .Y.

This is called by tape routines whenever the tape buffer address is needed.

63572 $F854   LDAD1

Tape: set load/save starting and ending pointers to the tape buffer.

TPBUFA* at 63565 ($F84D) is called to obtain the address of the tape buffer, which is then saved into the start-of-save pointer at 193-194 ($C1-C2); 192 is added and saved in the end-of-save pointer at 174-175 ($AE-AF).

LDAD1 is called by tape routines whenever the tape buffer address range is needed.

63591 $F867   FNDHDR*

Tape: find the tape header for a specified filename (or next).

The routine FAH 63399 ($F7AF) is called by this to find the next tape header. If no filename was specified, then it returns to the caller since the next tape file has been found. If a filename *was* specified, the filename in the header to the desired name is compared to the length specified in 183 ($B7). If the two don't match, this routine loops until a match is found.

TSTOP at 63819 ($F94B) is called for the STOP key test.

63626 $F88A   JTP20

Tape: increment the tape buffer character counter.

The count of the characters in the tape buffer counter at 166 ($A6) is incremented. It's then compared to location 192 ($C0) to determine if the tape buffer is full.

63636 $F894   CSTEL

Tape: display PRESS PLAY ON TAPE message.

The routine CS10 63659 ($F8AB) is called to determine the tape button status and if necessary, KMSGSHOW* 61926 (F1E6) is called to display the message. This routine loops while waiting for a tape button to be pressed.

Location 63819 ($F94B) is referenced to test for the STOP key.

63659 $F8AB   CS10
Tape: check tape's *play/rewind/forward* button status.

Location 37151 ($911F), bit 6 is tested. A BEQ instruction (bit 1 in .P is on) in the calling routine is taken if a tape button is pressed.

63671 $F8B7   CSTE2
Tape: display PRESS RECORD & PLAY ON TAPE message.

This routine calls CS10 63659 ($F8AB) to determine the tape button status and if needed, calls part of the CSTEL routine at 63642 ($F89A) to display the message and to loop while waiting for a button to be pressed.

63680 $F8C0   RDTPBLKS*
Tape: initiate tape header read.

The LDAD1 routine at 63572 ($F854) is called to set the load/save starting and ending pointers to the tape buffer. Location 147 ($93) is also set, so that it indicates a load operation; it then falls through to the next routine.

63689 $F8C9   RBLK
Tape: read blocks from tape.

This calls CSTEL 63636 ($F894) to display a PRESS PLAY ON TAPE message. IRQ interrupts are disabled to allow the changing of the IRQ vector so that tape routines are interrupt-proof. This also prepares parameters for and calls TAPE at 63732 ($F8F4) to enable an alternate IRQ vector to point to the routine READT 63886 ($F98E).

63715 $F8E3   WBLK
Tape: write blocks to tape.

LDAD1 63572 ($F854) is called to set the load/save starting and ending pointers to the tape buffer. Location 171 ($AB) is set by this to cause a short leader to be written, tape buffer to tape.

The CSTE routine at 63671 ($F8B7) is called, which displays the message PRESS RECORD & PLAY ON TAPE; the IRQ interrupts are disabled to allow a changing of the IRQ vector so tape routines are interrupt-proof; parameters are prepared for; and TAPE 63732 ($F8F4) is called to enable an alternate IRQ vector to point to WRTZ 64680 ($FCA8). This routine then falls through.

63732 $F8F4   TAPE
Tape: common tape read/write, start tape operations.

TAPE disables all IRQ interrupts from VIA 2, the normal source. It calls RSPAUSE* 61792 ($F160) to disable any RS-232 NMI

interrupts and saves the current IRQ vector from location 788–789 ($314–315) to address 671–672 ($29F–2A0). See the latter location for additional information.

The proper IRQ vector is set by a call to BSIV 64758 ($FCF6):
● If called for a save to tape, enables timer 2 VIA2 interrupts and resets the IRQ vector to WRTZ 64680 ($FCA8).
● For a load from tape, enables CA1 VIA2 interrupts and resets the IRQ vector to 63886 ($F98E).

This routine sets location 190 ($BE) to 2, the number of blocks to be saved or loaded, and calls NEWCH 64475 ($FBDB) to set up a new tape character. It turns on the tape motor by setting the proper bits in 37148 ($911C) and delays one-third second for the motor to gain speed.

The IRQ interrupt is enabled so that the next IRQ interrupt calls WRTZ or READT.

During tape processing, this routine loops while testing for the STOP key and updating the jiffy clock between IRQ interrupts. When the IRQ vector is finally reset to the default IRQ vector, the routine exits.

63819 $F94B   TSTOP
Tape: check for the STOP key.

This routine calls the STOP key test routine STOP 63344 ($F770) and exits if the key has not been pressed. If the key has been pressed, this jumps to TNIF 64719 ($FCCF) to deactivate the tape IRQ vector and restore the normal vector.

63837 $F95D   STTI
Tape: set time limit for tape dipole.

Timer 1 is set to a value which limits the amount of time the tape can be read before an IRQ occurs. This seems to be used to handle random errors, such as tape dropouts, preventing total disruption of a tape load as well as possibly allowing recovery.

63886 $F98E   READT
Tape: read tape data bits into location 191 ($BF) (IRQ driven).

The time between CA1 interrupt occurrences is determined for the current dipole. This dipole time is later used to determine whether the dipole just read is noise, 0, 1, or a word marker.

When a word marker, which signifies the end of a byte, is found, this routine jumps to TPSTORE* 64173 ($FAAD) to handle the byte just received.

If the bit just received is in error (for instance, a parity error), an error flag in 168 ($A8) is set.

The dipole timebase in 176 ($B0) is adjusted, based on the calculated value in 146 ($92); location 164 ($A4) is set to indicate which dipole was read.

The value of the first dipole, which is also the bit value, is saved in location 215 ($D7). If both dipoles were 0, a leader bit has been read and location 150 ($96) is set to indicate the tape is either before or between blocks.

If all eight data bits have not yet been received, the bit just received is rotated into the high order bit of address 191 ($BF).

Sets location 182 ($B6) if any tape errors, such as parity or mismatched dipoles, occurred during the reading of this byte.

If the byte has been fully processed, the routine RTI* 65366 ($FF56) is branched to.

Refer to location 146 ($92) for a description of a dipole; the location also includes Figure 1-2, which illustrates a square wave cycle.

64173 $FAAD   TPSTORE*

Tape: determine if to store the input character from tape.

This routine does a number of things. It starts looking for data if enough leader has been read, checks for a *long block* condition, and looks for a block countdown character, when one is expected. When a block countdown is found, is it the first or second block? Is the countdown character 1, indicating data is next? This routine also checks for a *short block* condition present. Has the end of the LOAD area been reached?

TPSTORE* also determines whether currently reading block one or block two.

If verifying, it compares the tape byte with RAM. If not equal, it sets location 182 ($B6) to indicate a verify error.

If loading, the routine checks 182 ($B6) for error flags. If there are none, it stores the byte in RAM and proceeds. If an error is indicated, it stores the address in the error log at 256–318 ($100–13E) for attempted correction. The error index in 158 ($9E) is incremented. If there are 31 errors, the read is aborted.

Error correction is attempted during the read of the second copy of the tape block, using the erroneous byte pointers.

When both blocks have been processed, the IRQ vector is reset to its normal default setting and a checksum (or parity) of the bytes that were just read is computed. This checksum should be equal to the one that was read as the last byte of block two. If they're not equal, ST in 144 ($90) is set to indicate a checksum error.

64466 $FBD2   RD300

Tape: called to reset the tape read pointer.

The tape read pointer in 172–173 ($AC–AD) is loaded from 193–194 ($C1–C2).

64475 $FBDB   NEWCH

Tape: new tape character setup.

This is called to prepare for a new character for tape input or output. The bit count in location 163 ($A3) is reset to 8 and the following related fields are zeroed: 164 ($A4), 168 ($A8), 169 ($A9), and 155 ($9B).

64490 $FBEA   TPTOGLE*

Tape: toggle the tape write line to invert the output signal.

The tape write line is flipped, reversing the signal polarity being written to the tape. Depending on the entry point used to enter the routine, various values are used for timer 2 for the time of the next cycle.

64518 $FC06   BLKEND*

Tape: end of block write processing.

This is called by the routine WRITE* 64523 (FC0B) once the memory area and the checksum byte have been written for a block, in order to set the high order bit of 173 ($AD) on, writing an interblock leader.

This then jumps to RTI* 65366 ($FF56) to end the interrupt exit.

64523 $FC0B   WRITE*

Tape: data write (IRQ driven).

This is the routine called with each VIA2 timer 2 interrupt to actually write the cycles onto tape. This makes up two dipoles for each bit to be saved or for writing a word marker. A word marker is two long cycles, followed by two medium cycles.

TPTOGLE* 64490 ($FBEA) is called for each cycle written.

This also checks location 173 ($AD) to see if the high order bit is on, indicating the block SAVE is complete. If it is on, the routine goes to WRTN1 64661 ($FC95) to write a leader.

The value of the bit written in the first dipole is inverted so that the second dipole will write the opposite value. For example, if the first dipole wrote cycles for values of 1–1, the second dipole will be 0–0. Similarly, if the first half was 0–0, the second half is 1–1. Thus, a bit value of 1 is represented by four cycles of 1–1—0–0, while a 0 is represented by four cycles of 0–0—1–1. Bits are written with four cycles, while they are read during tape load as two dipoles.

A parity work bit based on the bit being written is updated by this routine. Location 189 ($BD) shifts right one bit, moving the next bit to be sent into bit 0. When all eight bits have been written (indicated by 163 ($A3) containing zero), the parity bit to be written is prepared, as is the next byte to be sent. If all bytes from the save area have been written, this writes the checksum byte and branches to the BLKEND* routine at 64518 (FC06).

This routine jumps to RTI* 65366 ($FF56) to end the interrupt exit.

64661 $FC95   WRTN1
Tape: block leader write (IRQ driven).

This handles the end-of-block processing. If the second block was just saved, the tape motor is turned off by calling TNOFF 64776 ($FD08).

If the first block was just saved, a long dipole is written, followed by 80 leader cycles.

64680 $FCA8   WRTZ
Tape: leader write (IRQ driven).

WRTZ writes leader cycles to tape before blocks and between blocks. Leader cycles are very close to the length of time of the values written for 0 data cycles, and are indeed considered 0's when reading back the tape during tape load.

This routine resets the IRQ vector to 64523 ($FC0B) and enables interrupts so the next IRQ caused by timer 2 goes to the WRITE* routine to write a data block.

It also sets location 165 ($A5) to indicate that block countdown characters are to be written before actually writing the data from the save area to tape; it then jumps into the middle of the WRITE* routine to write the block countdown characters.

Address 190 ($BE) is tested for the completion of writing both blocks. If both are completed, this routine falls through.

64719 $FCCF   TNIF
Tape: restore IRQ vector.

The tape IRQ vector is deactivated and the normal vector is restored. This routine then calls TNOFF 64776 ($FD08) to stop the tape motor, changes location 37152 ($9120) back to scan for the STOP key, and restores the saved IRQ vector to 788–789 ($314–315) from location 671–672 ($29F–2A0).

64758 $FCF6   BSIV
Tape: reset the current IRQ vector.

This calls TNIF 64719 ($FCCF), then depending on the

parameter passed in .X, sets the IRQ vector in 788–789 ($314–315) to one of the following addresses in the table at 65009 ($FDF1):

63886 ($F98E) Read tape block
64680 ($FCA8) Write tape leader
64523 ($FC0B) Write tape

64776 $FD08   TNOFF

Tape: kill motor.

Bits 3–1 of VIA1 routine VIA1PCR* 37148 ($911C) are set to 1's to stop the tape motor.

64785 $FD11   VPRTY

Compare current to end of load/save pointers (tape and serial).

This routine compares the pointer to the current load/save byte (172–173, $AC–AD) to the pointer to the end of the area (174–175, $AE–AF). The carry flag is cleared if the end pointer is greater than the current pointer.

This is called to determine if the load/save action has completed.

64795 $FD1B   WRT62

Increment current load/save pointer (tape and serial).

One is added to the pointer which specifies the current load/save byte in location 172–173 ($AC–AD).

64802 $FD22   START

Power-on/reset routine (checks for autostart cartridge).

The routine sets the stack pointer to an effective address of 511 ($1FF) and clears the 6502 decimal mode; calls CHKAUTO* 64831 ($FD3F) to check for an autostarting cartridge; branches to the address contained in 40960 ($A000) if found; calls INITEM* 64909 ($FD8D) to initialize memory and system contents; calls RESTOR 64850 ($FD52) to set the system vectors; calls INITVIA* 65017 ($FDF9) to initialize the 6522 VIA register; calls INITSK* 58648 ($E518) to initialize the 6560 VIC chip registers and screen; enables interrupts; and branches to the address contained in 49152 ($C000) to start BASIC.

Location 65532 ($FFFC) contains the address of this routine for the 6502 to branch on when a 6502 RESET is detected.

You can use SYS 64802 to perform the power-on/reset routine (a cold restart) and SYS 64812 to skip the autostart cartridge test.

64831 $FD3F   CHKAUTO*

Check for an autostarting program at 40960 ($A000).

This routine is called by the routines START 64802 ($FD22) and NMI* 65193 ($FEA9).

Starting at address 40964 ($A004), this checks for the five characters *A0CBM* that are stored in location 64845 ($FD4D), setting the zero flag if found. See location 40960 ($A000) for more information.

64845 $FD4D   A0CBM*

A0CBM characters with the high order bit on in the last three characters. $41,30,C3,C2,CD

64850 $FD52   RESTOR

Cause the RAM system vectors to be reset to provided defaults.

This routine sets .X and .Y to the address of the provided system vector table at 64877 ($FD6D) and falls through to the next routine after clearing the carry flag in .P.

A jump to this routine is provided at location 65418 ($FF8A).

START 64802 ($FD22) and BREAK* 65234 ($FED2) both call this routine for power-on/reset and RUN/STOP–RESTORE keys processing.

64855 $FD57   VECTOR

Read or set system RAM vectors.

Normally, this stores the 16 vectors from 64877–64908 ($FD6D–$FD8C) into the RAM vectors in locations 788–802 ($314–322).

Entering this routine at location 64855 ($FD57) enables the carry flag of .P, which determines whether a read or set of the vectors is to be done. If carry is set, .X and .Y are used as the address where the vectors are copied to from the current contents of 788–802 ($314–322). If carry is clear, .X and .Y are used as the address of a user vector table that should be stored into 788–802 ($314–322).

A jump to this routine is provided at location 65421 ($FF8D). To modify one or more vectors, copy the RAM vectors to your user area, modify the vectors required, and then call for the user vectors to be copied to 788–802 ($314–322).

64877 $FD6D   VECTORS*

Default system vector address storage table.

Routine RESTOR 64850 ($FD52) copies these 16 vectors into the RAM vectors in location 788–802 ($314–322). See that area for details of the contents of this table.

64909 $FD8D   INITMEM*

Initialize system memory.

This zeros 0–255 ($0–FF) and 512–1023 ($200–$3FF). It sets 178–179 ($B2–B3) to point to the tape buffer at 828 ($33C) and starts looking for RAM from 1024 ($400) by calling the routine TSTMEM* 65169 ($FE91).

This sets the screen map memory page into 648 ($288). See that location for additional details.

The start of RAM pointer is set into 641 ($281) by calling the MEMTOP 65139 ($FE73) routine. See location 641 ($281) for further details of the possible values in that location.

If RAM ends before 8191 ($1FFF), this goes into an error loop.

START at 64802 ($FD22) calls this routine.

65009 $FDF1   IRQVCTRS*

IRQ vectors table.

64680 ($FCA8) Write tape leader
64523 ($FC0B) Write tape
60095 ($EABF) Normal default IRQ vector
63886 ($F98E) Read tape block

This table is used by the routine BSIV 64758 ($FCF6) to set or restore the IRQ vector in 788–789 ($314–315) to the needed address.

65017 $FDF9   INITVIA*

Initialize the 6522 VIA registers.

37136–37151 ($9110–$911F) 6522 VIA Chip 1
37152–37167 ($9120–$912F) 6522 VIA Chip 2

See those locations for additional information.

This routine is called by START 64802 ($FD22) and BREAK* 65234 ($FED2) for power-on/reset and RUN/STOP–RESTORE keys processing.

Initialization values are:

Auxiliary control registers:
timer 1: (free-running mode); output on PB7 disabled
timer 2: set to an interval timer in one-shot mode
shift register: disabled
port B latch enable: to reflect the data on the pins
port A latch enable: to reflect the data on the pins
VIA1 peripheral control register:
CB2: manual output mode (CB2 held high)
CB1: interrupt flag on a high-to-low transition
CA2: manual output mode (CA2 held high)
CA1: interrupt flag on a low-to-high transition
VIA2 peripheral control register:

CB2: manual output mode (CB2 held low)
CB1: interrupt flag on a high-to-low transition
CA2: manual output mode (CA2 held high)
CA1: interrupt flag on a low-to-high transition
Data direction VIA1 port B: input
Data direction VIA1 port A: input, except PA7 output
Data direction VIA2 port B: output
Data direction VIA2 port A: input
VIA1 interrupt enable register: transition on CA1
VIA2 interrupt enable register: timer 1
VIA2 timer 1: 17033 ($4289)

65097 $FE49   SETNAM
The filename pointer and length are stored from .X, .Y, and .A.

Prior to a LOAD or SAVE, this routine is used to set the desired filename information.

The length is stored from .A into location 183 ($B7). A 0 indicates that no filename is needed for tape.

The pointer to the filename is stored from .X and .Y into 187–188 ($BB–BC).

See all of these locations for additional information.

The jump at 65469 ($FFBD) should be used to access this routine.

65104 $FE50   SETLFS
Set the current file number, device, and secondary address.

.A (file number) is stored into location 184 ($B8), .X (device number) is stored into location 186 ($BA), and .Y (secondary address) is stored into location 185 ($B9). A value of 255 ($FF) is used in the latter to indicate no secondary address.

See these locations for additional information.

Prior to a LOAD or SAVE, this routine is used to set the desired file and device information.

The jump at 65466 ($FFBA) should be used to access this routine.

65111 $FE57   READST
Reset RS-232 status, branch to 65128 ($FE68) for non-RS-232 status.

If the current device number in location 186 ($BA) is not a 2 (signifying RS-232), this routine branches to location 65128 ($FE68). Otherwise, this zeros the RS-232 status in address 663 ($297). (The Commodore 64 Kernal returns the status in .A as well as zeroing the byte, but this was overlooked in the VIC and the .A returned is zeroed.)

Use RS=PEEK(663):POKE 663,0 to obtain the RS-232 status, not ST.

The jump at 65463 ($FFB7) should be used to access this routine.

65126 $FE66   SETMSG
Set the byte used to enable/disable Kernal message display.

This stores .A into 157 ($9D) and falls through. See location 157 ($9D) for more details.

The jump at 65424 ($FF90) should be used to access this routine.

65128 $FE68   READIOST*
Load .A with the non-RS-232 I/O status ST.

.A is loaded with the current contents of 144 ($90) and this routine then falls through. See location 144 ($90) for more details.

The jump at 65463 ($FFB7) should be used to access this routine.

65130 $FE6A   ORIOST*
OR .A with the contents of 144 ($90) ST and store there.

When reading the status information, the contents of 144 ($9A) have already been loaded into .A, so there is no effect.

This routine can be used to add a status bit by loading .A with a value to be ORed with 144 ($90).

The new value (or unchanged value) in .A is stored in 144 ($9A).

See the device number, secondary address, and status code table in Appendix D.

65135 $FE6F   SETTMO
Set timeout value for IEEE-488.

This stores .A in 645 ($285). It's used only with an IEEE-488 add-on card. A 0 in bit 7 enables a 64 millisecond timeout value, while a 1 in that bit disables the timeout altogether. See location 645 ($245) for additional information.

The jump at 65442 ($FFA2) should be used to access this routine.

65139 $FE73   MEMTOP
Read or set the top of memory pointer.

The carry flag in .P determines the action performed:
Carry clear: 643–644 ($283–284) set to .X .Y contents.
Carry set: .X .Y set to the contents of 643–644 ($283–284).

The jump at 65433 ($FF99) should be used to access this routine.

65154 $FE82   MEMBOT
Read or set the bottom of memory pointer.

The carry flag in .P determines the action performed:
Carry clear: 641–642 ($281–282) set to .X .Y contents.
Carry set: .X .Y set to the contents of 641–642 ($281–282).
The jump at 65436 ($FF9C) should be used to access this routine.

65169 $FE91   TSTMEM*
Test a memory location.

TSTMEM* stores values in bits 6–0 of the byte pointed to by 193–194 ($C1–C2) + .Y, after saving the contents of the byte. The original contents of the byte are restored after the testing is done.

On exit, the carry flag in .P is clear if the memory location proved to be non-RAM and set if it *is* a RAM location.

This routine is called by INITMEM* 64909 ($FD8D) when testing the limits of RAM.

65193 $FEA9   NMI*
NMI handler routine.

This routine processes RUN/STOP–RESTORE keys and VIA1 timer interrupts.

The 6502 chip jumps off the vector at 65530 ($FFFA) when an NMI is sensed. The vector points to this routine.

After disabling the IRQ interrupts, a jump off the link at 792–793 ($318–319) is performed, normally continuing with the next instruction in the routine.

The .A, .X, and .Y registers are saved on the stack.

CHKAUTO* 64831 ($FD3F) is called and branches to the address contained in 40962 ($A002) if an autostart cartridge is found, whether the RUN/STOP key was pressed or not.

If the VIA1 interrupt enable register at 37149 ($911D) indicates that the interrupt is disabled, or the RESTORE key was pressed without the RUN/STOP key, then this branches to RTI* 65366 ($FF56).

If the RESTORE key was *not* pressed, this goes instead to RSNMI* 65246 ($FEDE).

UDTIM 63284 ($F734) is called to update the jiffy clock, then calls the test STOP key routine STOP at 63344 ($F770). If the RUN/STOP key was pressed with the RESTORE key, this falls through to the following BREAK* routine. Otherwise, it branches to RTI* 65366 ($FF56).

The vector at 65530 ($FFFA) should be used to access this routine.

65234 $FED2   BREAK*
BREAK interrupt entry.

The BREAK* routine handles the BRK ML instruction ($00) and RUN/STOP–RESTORE keys. You can cause a warm restart by SYSing 65234.

RESTOR 64850 ($FD52) is called to reset the system vectors in RAM, INITVIA* 65017 ($FDF9) to reinitialize the 6522 VIAs, and INITSK* 58648 ($E518) to initialize the 6560 VIC chip registers and screen. This then branches to the address contained in 49154 ($C002) to perform a warm start of BASIC.

A vector to this routine is provided at location 790–791 ($316–317).

65246 $FEDE   RSNMI*
RS-232: NMI sequences.

An interrupt on VIA1 timer 1 is used for RS-232 bit transmission scheduling, VIA1 timer 2 is used for RS-232 bit reception, and VIA1 CB1 is used for RS-232 reception of a new byte.

This calls RSNXTBIT* 61347 ($EFA3) to send the next bit, or RSINBIT* 61494 ($F036) to receive an input bit. It performs baud rate calculations using 659 ($293) and the table at 65372 ($FF5C) in order to set VIA1 timer 2 for the next bit to be sent or received. This falls through to the next routine.

65366 $FF56   RTI*
Restore 6502 registers from the stack and return from interrupt.

65372 $FF5C   BAUDTBL*
RS-232: VIA timer 2 values for baud rate table.

If any of the NI (Not Implemented) values are chosen for the baud rate specification in location 659 ($293), the RS-232 control register, values are retrieved past the end of this table using the following routine's instructions as values. This results in the baud rate being set to less than 50.

Table 9—5. Baud Rates and VIA1 Timer 2 Values

Baud	VIA1 Timer 2 Value
50	9.135 millisecond
75	6.060 millisecond
110	4.103 millisecond
134.5	3.337 millisecond
150	2.985 millisecond
300	1.447 millisecond
600	0.679 millisecond
1200	0.294 millisecond
(1800) 2400	0.166 millisecond
2400	0.102 millisecond
3600 (NI)	
4800 (NI)	
7200 (NI)	
9600 (NI)	
19200 (NI)	

65394 $FF72   IRQROUT*
IRQ routine initial 6502 entry point.

The 6502 chip uses the vector at 65534 ($FFFE) to branch upon when an IRQ is sensed. That vector points to this routine. Further IRQs are disabled by the 6502.

The 6502 causes the program counter (MSB/LSB) and .P to be saved on the stack. This routine also places .A, .X, .Y, and the stack pointer on the stack.

If the Break flag in .P is set, the vector at 790–791 ($316–317) that points to the routine BREAK* at 65234 ($FED2) is branched upon. Otherwise, the vector at 788–789 ($314–315) is used to branch to the routine IRQ at 60095 ($EABF).

65413 $FF85   C4FFS*
Five unused bytes of 255 ($FF).

Location Range: 65418–65535 ($FF8A–$FFFF)
ROM Vectors

----- These contain a JMP opcode followed by the vector -----

65418 $FF8A   CRESTOR*
JuMP to 64850 ($FD52) RESTOR.

65421 $FF8D   CVECTOR*
JuMP to 64855 ($FD57) VECTOR.

65424   **$FF90**   **CSETMSG***
JuMP to 65126 ($FE66) SETMSG.

65427   **$FF93**   **CSECOND***
JuMP to 61120 ($EEC0) SECOND.

65430   **$FF96**   **CTKSA***
JuMP to 61134 ($EECE) TKSA.

65433   **$$FF99**   **CMEMTOP***
JuMP to 65139 ($FE73) MEMTOP.

65436   **$FF9C**   **CMEMBOT***
JuMP to 65154 ($FE82) MEMBOT.

65439   **$FF9E**   **CSCNKEY***
JuMP to 60190 ($EB1E) SCNKEY.

65442   **$FFA2**   **CSETTMO***
JuMP to 65135 ($FE6F) SETTMO.

65445   **$FFA5**   **CACPTR***
JuMP to 61209 ($EF19) ACPTR.

65448   **$FFA8**   **CCIOUT***
JuMP to 61156 ($EEE4) CIOUT.

65451   **$FFAB**   **CUNTLK***
JuMP to 61174 ($EEF6) UNTLK.

65454   **$FFAE**   **CUNLSN***
JuMP to 61188 ($EF04) UNLSN.

65457   **$FFB1**   **CLISTEN***
JuMP to 60951 ($EE17) LISTEN.

65460   **$FFB4**   **CTALK***
JuMP to 60948 ($EE14) TALK.

65463   **$FFB7**   **CRDST***
JuMP to 65111 ($FE57) READST.

65466   **$FFBA**   **CSETLFS***
JuMP to 65104 ($FE50) SETLFS.

65469   **$FFBD**   **CSETNAM***
JuMP to 65097 ($FE49) SETNAM.

----- The following are indirect JMPs off a RAM vector -----
----- The RAM vectors can be set to go to your code -----

65472   **$FFC0**   **COPEN**
JuMP off 794-795 ($31A-31B) IOPEN.

65475   **$FFC3**   **CCLOS**
JuMP off 796-797 ($31C-31D) ICLOSE.

65478   **$FFC6**   **INPCHN**
JuMP off 798-799 ($31E-31F) ICHKIN.

65481   **$FFC9**   **OUTCHN**
JuMP off 800-801 ($320-321) ICKOUT.

65484   **$FFCC**   **CCLRCHN**
JuMP off 802-803 ($322-323) ICLRCH.

65487   **$FFCF**   **CINCH**
JuMP off 804-805 ($324-325) IBASIN.

----- The following contain a JMP opcode followed by the vector -----

65493   **$FFD5**   **CLOAD**
JuMP to 62786 ($F542) LOAD.

65496   **$FFD8**   **CSAVE**
JuMP to 63093 ($F675) SAVE.

65499   **$FFDB**   **CSETTIM***
JuMP to 63335 ($F767) SETTIM.

65502   **$FFDE**   **CRDTIM***
JuMP to 63328 ($F760) RDTIM.

----- The following are indirect JMPs off a RAM vector -----
----- The RAM vectors can be set to go to your code -----

65505   **$FFE1**   **ISCNTC**
JuMP off 808-809 ($328-329) ISTOP.

65508   **$FFE4**   **CGETL**
JuMP off 810-811 ($32A-32B) IGETIN.

65534

65511 **$FFE7** **CCALL**

JuMP off 812–813 ($32C–32D) ICLALL.

----- The following have a JMP opcode followed by the vector -----

65514 **$FFEA** **CUDTIM***

JuMP to 63284 ($F734) UDTIM.

65517 **$FFED** **CSCREEN***

JuMP to 58629 ($E505) SCRN.

65520 **$FFF0** **CPLOT***

JuMP to 58634 ($E50A) PLOT.

65523 **$FFF3** **CIOBASE***

JuMP to 58624 ($E500) IOBASE.

65526 **$FFF6**

Four unused bytes of 255 ($FF).

----- The following fixed vectors are used by the 6502 chip -----

65530 **$FFFA** **VCTRNMI***

6502 vector to 65193 ($FEA9) NMI*.

65532 **$FFFC** **VCTRRST***

6502 vector to 64802 ($FD22) START.

65534 **$FFFE** **VCTRIRQ***

6502 vector to 6

Appendices

Appendix A

Using the Binary and Hexadecimal Memory Contents of the VIC-20

This appendix will show you how to manipulate and use the information stored in the memory of the VIC-20 by demonstrating the typical, practical things you'll want to do with the memory contents. You'll also see some convenient tools to help you use and decipher the contents of memory. We'll concentrate on the practical aspects of pointer formats, setting and testing bits (BInary digiTS), and determining what an individual byte contains. The theory of why and how bytes contain information can best be addressed by other excellent references. If you are totally unfamiliar with the concepts of binary numbers and hexadecimal notation, you may want to consult an introductory work on those subjects. The *practical* application of the theories is the focus of this appendix; the tools provided, and your experience in using these tools, will help to show you the theories involved.

Memory Contents

Let's start our examples with the problem of determining what a memory location contains. PRINT PEEK(205) will display the decimal number equivalent of the binary contents of the eight-bit byte at location 205. When you look up that address in the memory map, you'll see that the content of memory location 205 is a cursor blink countdown. Let's suppose that the number 21 was displayed by the PEEK(205). (Since this memory location changes rapidly, you could see different numbers here at any point in time.) By consulting the code chart in Appendix C, you can see that 21 is stored in the byte as the binary number 00010101, expressed in hexadecimal notation as 15. You'll also notice that 21 could be the screen POKE code of the letter U, or the 6502 operation code of ORA-ZX, depending on the location of the byte containing that number. If you were PEEK-ing in the BASIC or Kernal routines, there would be a good chance that the 21 represents the ORA-ZX operation code, while a PEEK to the screen RAM map would indicate that the letter U was stored there for display purposes.

Type NEW on your VIC, then 10 GOTO10, and press the ENTER key. Next, type in PEEK(4613) on an 8K+ expanded VIC or PEEK(4101) on an unexpanded VIC. The number 137 you now see

displayed is within the BASIC program area, as you can tell by look-
ing up that memory location in the map. By consulting the section of
Appendix B that describes the internal storage format of BASIC pro-
grams, you'll see that the address you've PEEKed corresponds to the
first byte of the first BASIC line, after the link and line number
fields. The Appendix C code chart verifies that 137 is the token for
the GOTO keyword in BASIC. 137 could also represent the f2 key
being pressed, or the reversed letter *I* in the screen RAM map. The
chart also shows you the binary and hexadecimal representation of
137. Remember that the binary format is how the byte actually
appears in memory, and that BASIC displays it in decimal format for
your convenience.

Manipulating Bits

Now let's examine and manipulate bits within a byte. The most
common bits that you'll want to change are in the 6560 VIC register
at location 36879. This byte contains the selected background color
in bits 7–4, the inverse color switch in bit 3, and the selected border
color in bits 2–0. You may want to turn to the description of this
byte in the map to familiarize yourself with its use.

A few terms need to be defined before you begin working with
bits. The term *high order bit* refers to bit 7 of a byte, and the term
high order three bits refers to bits 7–5. The *low order bits* are those
with lower bit numbers. The *low order bit* is bit 0, and the *low order
two bits* are bits 1–0. A *nybble* is a half byte (four bits); usually it's
clear which half byte is being discussed, as in the term *low order
nybble*. Bits are numbered from left to right within a byte, from 7 to
0.

Table A–1. Bit Positions, Decimal/Hex Values

Bit	7	6	5	4		3	2	1	0
Dec	128	64	32	16		8	4	2	1
Hex	$80	$40	$20	$10		$08	$04	$02	$01

If this is your first exposure to binary numbers, you may find
that it seems a bit confusing at first, but as you use the bits (as you
undoubtedly will), you'll find that using bit values becomes second
nature. Refer to the VIC-20 Code Chart in Appendix C for a com-
plete table of each byte's binary, hexadecimal, and decimal numbers
if you get confused.

As an example, you can check the background color location at
36879 to see what bits are set there. If you PEEK that location, you'll
receive a number such as 27, which doesn't match the possible color
codes for background colors. By looking up 27 in the code chart,

however, the binary column gives you the answer. Only one bit of the four high order bits is *on*, or set to 1 (0001); that's the fourth bit. As you can see from Table A-1, the fourth bit has a value of 16. To determine the 0–15 color code that this represents, simply divide it by the rightmost bit of the high nybble; that bit is again the fourth bit, which has a value of 16. 16/16 is 1, the color code for white. If the bit combination of the high nybble had been 0110, you could see from Table A-1 that bits 5 and 6 were on (indicated by a 1). You simply add the bit values of these two *on* bits together (64+32=96), and then to determine the color code, divide that result by 16 (96/16=6). Six is the color code for blue.

A BASIC program you're running doesn't have the charts and tables to refer to, so how would it determine the current background color? The program *could* have variables set to all the possible combinations and test for each one, or it could use a formula to determine the value. Here are some bit testing and setting formulas for your programs.

The variable BV is the Bit Value, BYTE is the target memory byte, and RSLT is the result of the formula.

● To put BYTE bits 7–4 to RSLT bits 3–0 (this is the same as the divide-by-the-rightmost-bit-value solution discussed):
RSLT=(PEEK(BYTE) AND 240)/16

● To put BYTE bits 3–0 to RSLT bits 7–4: RSLT=(PEEK(BYTE) AND 15)*16

● To load RSLT with only bits 2–0 (for example):
RSLT=PEEK(BYTE) AND 4+2+1

● To determine if a particular bit is *on* or *off*: RSLT=(PEEK(BYTE) AND BV)/BV. RSLT=1 if the bit is *on*, or RSLT=0 if it is *off*.

● To reverse the setting of a bit in BYTE: POKE BYTE,(PEEK(BYTE) AND 255−BV) OR (1−(PEEK(BYTE) AND BV)/BV)*BV

● To turn *off* a bit in BYTE: POKE BYTE,PEEK(BYTE) AND 255−BV

● To turn *on* a bit in BYTE: POKE BYTE,PEEK(BYTE) OR BV

● To put a 1 or 0 in RSLT back into a particular bit in BYTE: POKE BYTE,(PEEK(BYTE) AND 255−BV) OR (RSLT*BV)

The examples, formulas, the bit number/value chart, and the VIC-20 Code Chart should make your bit setting and testing a bit simpler.

LSB/MSB Format

Using addresses, vectors, links, and pointers is important when you're exploring the VIC computer. These are all stored in an LSB/MSB (Least Significant Byte/Most Significant Byte) format. Again, the reasons for the existence of this format are less important, for the purposes of this discussion, than the methods for using the format.

Appendix A

The LSB/MSB format of pointers and BASIC line numbers can be decoded by using PTR=PEEK (LSB)+PEEK(MSB)*256.

The VIC-20 Code Chart in Appendix C can be useful when decoding or composing this format of numbers. Look at the column on that chart labeled MSB ADDR. For any given contents of a byte, this column gives you the result of multiplying it by 256. If you were to PRINT PEEK(4), the number 209 would be displayed. Location 3–4 is a pointer to a BASIC routine. To find the address of that routine, look up 209 in the Code Chart and note the MSB ADDR result of 53504. Now by doing PRINT PEEK(3) you'll see that 170 is the LSB portion of the pointer. Adding 170 to 53504 gives us the address of the routine in decimal: 53674. Of course, in a program you would use PTR=PEEK(3)+PEEK(4)*256.

Going the other way with this conversion technique is necessary when you're setting up location 1–2 for an upcoming USR instruction, for instance. If the target routine just happened to be at location 53674 (it wouldn't be, but this allows a clearer example), you would look in the Code Chart for the last line in the MSB ADDR column that has a value less than 53674. You should find it to be 53504, which is on the line for the decimal number 209. Now subtract 53504 from 53674 and you have the LSB number of 170; the MSB number is the 209 you found on the VIC-20 Code Chart. In a program you would compute these LSB and MSB numbers by: POKE 2,INT(53674/256):POKE 1,53674−(PEEK(2)*256). Using these techniques, you should have no problems using and setting LSB/MSB format pointers and numbers.

The program below includes routines that can be used in your own programs to perform conversions between decimal, binary, and hexadecimal numbers up to two bytes in length (0–65535, $0–FFFF). These routines are part of a program to display the contents of memory in all three number systems, and in character format, as well as the address of the location in both hexadecimal and decimal. This program will be useful for performing number conversions and displaying memory contents of the VIC-20 as you explore and use the map information.

On an unexpanded VIC-20, this program should be entered in an abbreviated form. For the unexpanded VIC-20, eliminate lines 1000–1250, all REM statements, and lines containing only a colon mark. No GOTO, THEN, or GOSUB statements reference these lines. In addition, embedded spaces outside of quotes may be omitted. The number of statements per line has been kept to a minimum to improve the readability of the program.

The lines containing the subroutines for number conversions are clearly marked, and the variables used as input and output by each of the subroutines are indicated.

298

Program A—1. Number Conversion

```
1000 GO TO 1260{3 SPACES}[ BYPASS NON-REM'ED HEADE
     R ]
1010 :
1020 "{18 SPACES}{A}************
     {S}
1030 "{18 SPACES}-NUM.BASE.MAP-
1040 "{18 SPACES}{Z}************
     {X}
1050 :
1060 MAPPING THE VIC-20{7 SPACES}G.R.DAVIES
     {11 SPACES}8/8/83
1070 :
1080 CONVERTS DEC OR HEX TO DEC/HEX/BIN AND DISPLA
     YS MEMORY
1090 BYTES IN DECIMAL, HEXADECIMAL, BINARY, AND CH
     ARACTER.
1100 :
1110 "ENTER ? AT THE PROMPT FOR RESPONSE FORMATS A
     ND HELP.
1120 :
1130 :
1140 * * * * * NUMBER * CONVERSION * SUBROUTINES *
     * * * *
1150 *{51 SPACES}*
1160 *{2 SPACES}DECIMAL TO HEXADECIMAL:{3 SPACES}L
     INES{2 SPACES}2620 - 2730{5 SPACES}*
1170 *{51 SPACES}*
1180 *{2 SPACES}HEXADECIMAL TO DECIMAL:{3 SPACES}L
     INES{2 SPACES}2490 - 2590{5 SPACES}*
1190 *{51 SPACES}*
1200 *{2 SPACES}DECIMAL TO BINARY:{8 SPACES}LINES
     {2 SPACES}2760 - 2880{5 SPACES}*
1210 *{51 SPACES}*
1220 * * * * * NUMBER * CONVERSION * SUBROUTINES *
     * * * *
1230 :
1240 :
1250 REM ***** TITLE FRAME *****
1260 PRINT"{CLR}{RVS}{2 SPACES}DECIMAL-HEX-BINARY
     {6 SPACES}BASE CONVERTER{6 SPACES}AND MEMORY
     {SPACE}DISPLAY{2 SPACES}"
1270 :
1280 :
1290 REM ***** MAIN PROMPTER *****
1300 PRINT"{RVS}ENTER:{OFF} NUMBER{16 SPACES}NUMBE
     R{RVS}.{OFF}BYTES{6 SPACES}";
1310 PRINT"{RVS}?{OFF} FOR INSTRUCTIONS
1320 PRINT"{3 SPACES}{RVS}Q{OFF} TO QUIT."
```

```
1330 :
1340 :
1350 REM ***** GET RESPONSE AND CHECK *****
1360 INPUT N$
1370 N1$=LEFT$(N$,1)
1380 IF N1$="?" THENGOSUB2920 : GOTO1300
1390 IF N1$="Q" THEN END
1400 :
1410 REM * EXTRACT $ *
1420 IF N1$="$" OR (N1$>"/" AND N1$<":") GOTO 1460
1430 PRINT"{RVS}*ERROR*{OFF} "N1$" IN "N$" MUST BE
     A $ OR DIGIT" : GOTO1360
1440 :
1450 REM * N2$=DIGITS ONLY, NO $ OR DOT *
1460 N2$=N$
1470 IF N1$="$" THEN N2$=MID$(N$,2)
1480 :
1490 REM * FIND ANY DOT IN ENTRY *
1500 DOT=0
1510 FOR X=1 TO LEN(N2$)
1520 IF MID$(N2$,X,1)="." THEN DOT=X
1530 NEXT
1540 :
1550 REM * SPLIT OUT ANY BYTE COUNT *
1560 IF DOT=0 GOTO 1610
1570 N3$=MID$(N2$,DOT+1)
1580 N2$=LEFT$(N2$,DOT-1)
1590 :
1600 REM * ROUTE HEX/DEC CONVERSIONS *
1610 IFN1$="$"GOTO1920
1620 :
1630 :
1640 REM ***** DECIMAL TO HEX CONVERSION *****
1650 REM * VALIDATE DEC *
1660 FORX=1 TO LEN(N2$)
1670 X$=MID$(N2$,X,1)
1680 IF X$<"0" OR X$>"9" GOTO 1710
1690 NEXT : GOTO1730
1700 REM (RESET LOOP)
1710 FOR X=0 TO 0 : NEXT
1720 PRINT"{RVS}*ERROR*{OFF} "X$" IN "N2$" IS NOT
     {SPACE}A DEC DIGIT" : GOTO 1360
1730 N2=VAL(N2$)
1740 IF N2<65536 GOTO 1760
1750 PRINT"{RVS}*ERROR*{OFF} "N2$" IS ABOVE 65,535
     " : GOTO 1360
1760 DE=N2
1770 :
1780 REM * RE-ROUTE BYTE DISPLAY *
```

```
1790 IF DOT>0 GOTO 2210
1800 :
1810 REM * DO DEC->HEX->BIN *
1820 D=DE
1830 GOSUB2660
1840 HE$=H$
1850 D=DE
1860 GOSUB2800
1870 PRINT"{BLK}DEC="N2;"HEX=$"H$:PRINT"BIN"B$"
     {BLU}"
1880 GOTO 1360
1890 :
1900 :
1910 REM ***** HEXADECIMAL TO DECIMAL CONVERSION *
     ****
1920 REM * VALIDATE HEX *
1930 FOR X=1 TO LEN(N2$)
1940 X$=MID$(N2$,X,1)
1950 IF X$<"0" OR X$>"F" OR (X$>"9" AND X$<"A") GO
     TO 1980
1960 NEXT : GOTO2000
1970 REM (RESET LOOP)
1980 FOR X=0 TO 0 : NEXT
1990 PRINT"{RVS}*ERROR*{OFF} "X$" IN $"N2$" IS NOT
      A HEX DIGIT (0-9,A-F).": GOTO 1360
2000 IF LEN(N2$)<5 GOTO 2040
2010 PRINT"{RVS}*ERROR*{OFF} $"N2$" IS ABOVE $FFFF
     ": GOTO1360
2020 :
2030 REM * DO HEX->DEC *
2040 HE$=RIGHT$("000"+N2$,4)
2050 H$=HE$
2060 GOSUB 2530
2070 DE=D
2080 :
2090 REM * RE-ROUTE BYTE DISPLAY *
2100 IF DOT>0 GOTO 2210
2110 :
2120 REM * DEC TO BINARY *
2130 GOSUB 2800
2140 PRINT"{BLK}DEC="DE;"HEX=$"HE$:PRINT"BIN"B$"
     {BLU}"
2150 GOTO 1360
2160 :
2170 :
2180 REM ***** DISPLAY MEMORY CONTENTS *****
2190 REM * DISPLAY THE BYTE ADDRESS *
2200 REM * DECIMAL TO HEXADECIMAL *
2210 FOR CNT=0 TO VAL(N3$)-1
2220 D=DE
```

Appendix A

```
2230 GOSUB 2660
2240 HE$=H$
2250 PRINT"{BLK}{RVS}AT "DE"{LEFT} $"HE$"{BLU}
     {OFF}"
2260 :
2270 REM * GET AND CONVERT THE BYTE CONTENTS *
2280 BYTE=PEEK(DE)
2290 D=BYTE
2300 GOSUB 2660
2310 D=BYTE
2320 GOSUB 2800
2330 H$=RIGHT$(H$,2)
2340 B$=RIGHT$(B$,9)
2350 :
2360 REM * FORM CHR$() *
2370 X$="{SHIFT-SPACE}"
2380 IF (BYTE>31 AND BYTE<128) OR BYTE>160 THEN X$
     =X$+CHR$(BYTE)
2390 BY$=RIGHT$(" "+STR$(BYTE)+" ",4)
2400 :
2410 REM * SHOW THE CONTENTS *
2420 PRINT"{BLK}"BY$;"$"H$" ";B$;X$"{BLU}"
2430 DE=DE+1
2440 REM WRAP AROUND TO 0 AT 65535
2450 IF DE>65535 THEN DE=0
2460 NEXT : GOTO1360
2470 :
2480 :
2490 REM ##### HEXADECIMAL TO DECIMAL CONVERSION S
     UBROUTINE #####
2500 REM{3 SPACES}VARIABLES: IN= H$ (FOUR HEX DIGI
     TS, WITH LEADING ZEROS)
2510 REM{3 SPACES}VARIABLES: OUT=D (0-65535)
2520 REM{3 SPACES}THIS ROUTINE DESTROYS H$
2530 D=0
2540 FOR X=1 TO 4
2550 D%=ASC(H$)
2560 D%=D%-48+(D%>64)*7
2570 H$=MID$(H$,2)
2580 D=16*D+D%
2590 NEXT : RETURN
2600 :
2610 :
2620 REM ##### DECIMAL TO HEXADECIMAL CONVERSION S
     UBROUTINE #####
2630 REM{3 SPACES}VARIABLES: IN= D (<65536)
2640 REM{3 SPACES}VARIABLES: OUT= H$ (FOUR HEX DIG
     ITS)
2650 REM{3 SPACES}THIS ROUTINE DESTROYS D
```

```
2660 H$=""
2670 D=D/4096
2680 FOR X=1 TO 4
2690 D%=D
2700 X$=CHR$(48+D%- (D%>9) *7)
2710 H$=H$+X$
2720 D=16*(D-D%)
2730 NEXT : RETURN
2740 :
2750 :
2760 REM ##### DECIMAL TO BINARY CONVERSION ROUTINE
     #####
2770 REM{3 SPACES}VARIABLES: IN= D (0-65535)
2780 REM{3 SPACES}VARIABLES: OUT =B$ (0000.0000 00
     00.0000)
2790 REM{3 SPACES}THIS ROUTINE DESTROYS D
2800 X$="0"
2810 IF D>32767 THEN X$="1" : D=D-32768
2820 FOR X=14 TO 0 STEP-1
2830 Y=2↑X
2840 X$=X$+RIGHT$(STR$( (DANDY) >0),1)
2850 NEXT
2860 B$=LEFT$(X$,4)+"."+MID$(X$,5,4)+" "
2870 B$=B$+MID$(X$,9,4)+"."+RIGHT$(X$,4)
2880 RETURN
2890 :
2900 :
2910 REM ***** HELP FRAME *****
2920 PRINT"{CLR}{RVS}TO CONVERT:{OFF} ENTER THE DE
     CIMAL NUMBER OR {RVS}${OFF}"
2930 PRINT"AND THE HEXADECIMAL":PRINT"(EX. 1024 OR
      $400)"
2940 PRINT"{2 SPACES}WITH A MAXIMUM OF{5 SPACES}65
     535 AND $FFFF.";
2950 PRINT"{4 SPACES}LEADING ZEROS MAY BE
     {2 SPACES}OMITTED."
2960 PRINT"{RVS}TO DISPLAY:{OFF} ENTER DEC OR $HEX
      ADDRESS {RVS}.{OFF} AND"
2970 PRINT"THE NUMBER OF BYTES TODISPLAY";
2980 PRINT"(EX. $33C.10{2 SPACES}WILL DISPLAY 10 B
     YTES AT 828 $33C
2990 PRINT"IF NO NUMBER ENTERED AFTER {RVS}.{OFF}
     {SPACE}THEN 1 ASSUMED.
3000 RETURN
```

Appendix B

BASIC Area Pointers and Internal Storage Formats of Variables and Lines

BASIC Area Pointers

The area of memory the VIC-20 uses for storage of a BASIC program and its variables is located at varying location ranges, depending on the amount of expansion memory added to the VIC-20. (Appendix E explores the subject of relocation of VIC-20 memory by the system or the user.) Regardless of the actual memory location range used for the BASIC storage area, certain pointers in low memory are used by BASIC to point to the limits of the BASIC area and to divide the area into several sections. These consist of several variable storage areas, an area for the BASIC program itself, and an unused area that is available for use by BASIC during program execution. Each of these areas, or pools, can expand or contract, with a few exceptions, by simply altering the pointers to the pool and moving any contents to the proper new location. This may involve moving an entire variable pool.

After the Kernal has initialized the VIC, the range of the BASIC area is pointed to by:

- 43–44 ($2B–2C) start of the BASIC area. On an unexpanded VIC-20 this pointer contains the address of 4097 ($1001).
- 55–56 ($37–38) end of the BASIC area. The unexpanded VIC-20 has an address of 7680 ($1E00) in this location.

The Kernal uses a different set of pointers to keep track of the top and bottom of the RAM space that contains the BASIC areas 641–642 ($281–282) and 643–644 ($283–284). BASIC has no effect on these Kernal pointers.

Additional pointers are maintained by BASIC to point to the boundaries of each type of variable pool:

- 45–46 ($2D–2E) start of the scalar (nonarray) variable pool.
- 47–48 ($2F–30) start of the array variable pool.
- 49–50 ($31–32) start of the free area.
- 51–52 ($33–34) bottom of the string variable pool.

The BASIC area may be reduced at the top and/or bottom by adjusting the pointers at 43–44 and 55–56. This is useful when a machine language (ML) program or data such as screen maps and custom character sets is to be placed in the user RAM area without

304

BASIC destroying the information. After adjusting the top of BASIC pointer at 55–56, a CLR statement should be used, which will cause all the current BASIC variables to be lost. A NEW statement is needed after adjusting the pointer at 43–44, causing the current BASIC program and variables to be lost. You can refer to a program at location 55–56 that reserves space at the top or the bottom of the BASIC area by resetting the pointers. ·

Advanced users can manipulate the pointers at 43–44 and 55–56 to do such things as allow multiple BASIC programs in RAM at one time, place the BASIC area in the 40960 ($A000) ROM area, append routines from other programs, and a variety of other techniques. The actions that NEW and CLR perform are described in more detail at their respective routine addresses in the map.

Once a BASIC program has been loaded into the BASIC area, the pointer at 45–46 is set to point one byte past the end of the BASIC program. The scalar variable pool will be started at this address when the program is RUN. If additional lines are added to the program, this pointer is pushed up as the lines are stored in their proper line number sequence within the BASIC program area. Higher numbered BASIC statements are pushed up to create room for the new statements. Additionally, adding or changing BASIC lines causes any variables still in the variable pools from a previous RUN to be lost.

When RUN is entered, BASIC begins to allocate space from the various variable pools to accommodate the variables defined in the program.

As an example, Figure B-1 shows the pointer relationships after RUN was entered and the STOP key was pressed on an unexpanded VIC-20, with the location in parentheses showing the result of entering a direct mode FRE(0) that returned 235 bytes to the free area.

Figure B–1. Pointer Relationships

Pointer	Area Use	Location
55–56 →	+-------------+	7680
	String Pool	
51–52 →	+-------------+	7323 (7558)
	Free Area	
49–50 →	+-------------+	6201
	Array Pool	
47–48 →	+-------------+	6070
	Scalar Pool	
45–46 →	+-------------+	6000
	BASIC program	
43–44 →	+-------------+	4097

Appendix B

When variables such as scalar variables, function descriptors, or string descriptors that point to the string in the string pool or BASIC program are added to the scalar pool, all entries in the array pool must be pushed up seven bytes (the size of all variables or descriptors stored in the pool) into the free area. Added arrays decrease the amount of the free area from the bottom, while strings stored in the string pool decrease the free area from the top. Since adding or changing a BASIC program clears the variables, no pushing up of variables is involved.

A CLR statement causes the pointer at 51–52 to be the same as 55–56, eliminating the boundary for the string pool. Then the pointers at 47–48 and 49–50 are overlaid with the address that is in the pointer at 45–46, eliminating the other pool boundary pointers. The only pointer of the group that has gone untouched is the pointer at 45–46, which specifies the end of the BASIC program.

The variables in the pools do not need to be erased since they will be simply overlaid when the space they occupy is needed. A NEW statement causes the pointer at 45–46 to be overlaid by the pointer at 43–44. In addition, the end of program three-byte indicator is placed at the beginning of the BASIC program area. Location 43–44 in the memory map explains how to recover your program from an unintentional NEW statement.

BASIC Statement Storage

BASIC program lines are stored in the program area in a compressed format, called *tokenized*. Each line is preceded by a two-byte link field that points in LSB/MSB format to the next BASIC line's link field. (If you're unfamiliar with the LSB/MSB format, refer to Appendix A for a short explanation.) Following the two-byte link field is a two-byte line number field in LSB/MSB format, then the tokenized BASIC line, ended with a byte containing 0.

The last BASIC program line has a link field that points to a dummy link field of two bytes of zeros. This is the end-of-program indicator that the pointer in 45–46 points just beyond for the start of the scalar variable pool.

The tokenization of BASIC program lines replaces BASIC keywords with a one-byte shorthand. You can see what these tokens are by looking at the VIC-20 Code Chart in Appendix C. Variable names, character strings, and line number references (for instance, GOTO 2000) are not tokenized. The stored BASIC lines appear as shown in Figure B-2.

BASIC Scalar Variables

Integer scalar variables. The seven-byte field for an integer variable is stored in the scalar variable pool in the format illustrated in Figure B-3.

Figure B—2. Stored BASIC Lines

Link-field LSB MSB	Line-number LSB MSB	BASIC line....	0

↓

Link-field LSB MSB	Line-number LSB MSB	BASIC line....	0

↓

Link-field 0 0	End of program
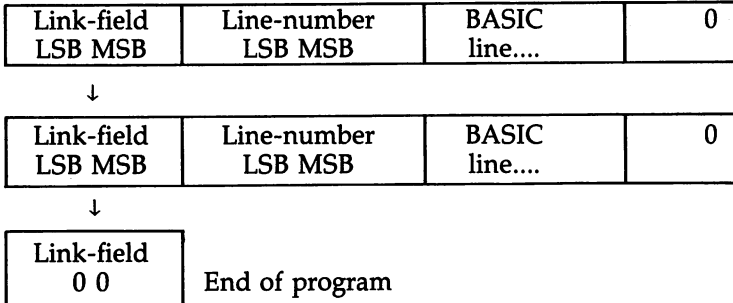
Figure B—3. Integer Scalar Variables

CHAR1 +128	CHAR2 +128	MSB	LSB	0	0	0

The characters occupying the first two bytes of the seven-byte variable contain the first two characters of the variable name, with 128 added to the ASCII value of both characters. If the variable has a name of one character (for example, Y%), the second character in the variable is simply the value of 128. The percent sign is not stored since the 128 added to each character identifies the variable as an integer type. The actual value that the integer variable is set to is in *MSB/LSB* format, a departure from line number and pointer LSB/MSB format. The value is stored in binary with a negative number indicated by the value having the high order bit on and the rest of the number in two's complement form (the individual bits are reversed and 1 is added). The valid range for an integer number is −32768 to 32767. For example, A%=2797 would be stored as detailed in Figure B-4.

Figure B—4. Integer Variable Example

A	%					
char1 193	char2 128	MSB 10	LSB 237	0 0	0 0	0 0

(10 * 256) + 237 = 2797

Floating point scalar variables. The seven-byte field for a floating point variable is stored in the scalar variable pool in the format shown in Figure B-5.

Appendix B

Figure B—5. Floating Point Scalar Variables

CHAR1 +0	CHAR2 +0	EXP	MANT1	MANT2	MANT3	MANT4

The first two bytes of the seven-byte variable contain the first two characters of the variable name. If the variable has a one-character name such as *R*, the second character in the variable is zero. The variable is in exponent and four-byte mantissa form. For example, the number 1.41421356 is stored as 129 53 04 243 52. You'll better understand this after we examine the parts of a floating point number.

Floating point exponent. The exponent of a floating point number specifies the direction and magnitude that the decimal point (actually the *binary* point) must be shifted (in number of bits) to obtain the actual value, much like the scientific notation of numbers. A right shift of one bit position of the binary point is expressed as 129 (1 + 128), while a left shift of one position would be 127 (-1 + 128). A right shift of 127 positions results in an exponent value of 255 (127 + 128), while a left shift of 128 is expressed as an exponent of 0 (-128 + 128). However, a 0 exponent is actually used to indicate a zero floating point number, since normalization of the exponent results in a 0 exponent only for a 0 floating point number.

Floating point mantissa. The high order bit of the first byte of the mantissa is used as a sign bit, 0 indicating a positive number and 1 a negative number. In scientific notation, 2,827,381.66 could be expressed as .282738166 E7. This means that the decimal point is to be shifted right seven positions or that ten to the seventh power (10,000,000) is to be multiplied by the number. The number before the E is called the mantissa.

Floating point numbers in the VIC-20 have a mantissa expressed as a binary number, so the number is multiplied by two to the exponent power. BASIC normalizes the mantissa so that the bit on the extreme left of the mantissa is the first significant bit (leading zero bits are discarded). The number five is 00000101 in binary, which is converted to 10100000 when normalized, and the exponent is set to record the number of significant bits. Three significant bits (101), for example, are stored in the exponent as 131 (3 + 128 = 131). The mantissa of the number 5 would be stored as 160 00 00 00 in decimal, $A0 00 00 00 in hexadecimal, or 10100000 00000000 00000000 00000000 in binary. The sign of the mantissa is placed in the high order bit of the first mantissa byte by storing a 0 if the number is positive, or a 1 for a negative number. The result of our example is then 32 00 00 00, $20 00 00 00, or 00100000 000000000 000000000

000000000. Remember that the exponent byte had been set to 131 ($83) after the mantissa was normalized.

When converting this format back to an unnormalized floating point number, the first byte of the mantissa is loaded into the floating point accumulator sign byte, where the 0-if-positive or 1-if-negative rule applies. The first byte of the mantissa is ORed with 128 ($80), setting the bit that was used to hold the sign. Then the complete mantissa is loaded into the floating point accumulator. The mantissa is positive since the sign bit contains a 0. Because the exponent is greater than 128 (meaning a positive number), the mantissa binary point is shifted 131−128=3 bits to the right. This results in 101.00000 00000000 which is the number 5 in decimal.

The range of values that a floating point variable can be set to is approximately +/− 1.701 E38 (exponent being positive or negative). A floating point variable will be displayed in scientific notation at approximately +/− 999,999,999.2. See Figure B-6 for a diagram of an example of a floating point mantissa's format and values.

Figure B—6. Mantissa Example

Y = 1.41421356

CHAR1	CHAR2	EXP	MANT1	MANT2	MANT3	MANT4
89	00	129	53	4	243	47

Floating Point Accumulators

The six-byte floating point accumulators at locations 97–102 ($61–66) and 105–110 ($69–6E) have a format much like the floating point scalar variables, except that the mantissa's sign is located in a byte of its own, freeing the high order mantissa bit for more precision. The sign byte of the mantissa is set to 0 for positive numbers and 255 ($FF) for negative numbers. The floating point accumulator format would look something like Figure B-7.

Figure B—7. Floating Point Accumulator

EXP	MANT1	MANT2	MANT3	MANT4	SIGN

For details of the exponent and mantissa bytes, see the explanation of the scalar floating point variables.

Examples of floating point numbers in an accumulator in hexadecimal notation are:

Appendix B

Table B—1. Floating Point Numbers in an Accumulator

Number	Exp	M1	M2	M3	M4	Sign
0	$00	00	00	00	00	00
.25	$7F	80	00	00	00	00
.50	$80	80	00	00	00	00
1	$81	80	00	00	00	00
2	$82	80	00	00	00	00
−1	$81	80	00	00	00	FF
−10	$84	A0	00	00	00	FF
1E10	$A2	95	02	F9	00	00
2E10	$A3	95	02	F9	00	00
4E10	$A4	95	02	F9	00	00
1E38	$FF	96	76	99	52	00
1E-38	$02	D9	C7	EE	EE	00
1E-39	$00	A0	00	00	00	00

Function Descriptors

The seven-byte field for a function descriptor is stored in the scalar variable pool in the format shown in Figure B-8.

Figure B—8. Function Descriptors

CHAR1 +128	CHAR2 +0	EXPRESSION LSB MSB	VARIABLE LSB MSB	FILL CHAR

The first two bytes contain the first two characters of the variable name, with 128 added to the ASCII value of the first character. If the variable has a name of one character (for instance, FN F), the second character in the variable is 0. The expression pointer points to the description of the function in the BASIC statement with the DEF FN for the function. The variable pointer specifies the dependent variable in the scalar variable pool. The dependent variable in DEF FN A (C)=1267/RT, for example, is C. The fill character in the seventh byte is the first character of the expression after the = sign. It has no significance and is placed there only to fill out the format. For more details about user functions, see locations 71–72 ($47–48), 78–79 ($4E–4F), 54195 ($D3B3), and 54260 ($D3F4) in the memory map.

Look at Figure B-9 for a moment for an example of a function descriptor, its format, and its values.

Figure B—9. Function Descriptor Example

DEF FN A (C) = 1267 / RT

CHAR1	CHAR2	EXPRESSION	VARIABLE	FILL
193	0	14 16	39 18	49

At location 4647 (18*256+39) we would find the floating point variable C to be 67 0 0 0 0 0 0.

String Scalar Variables

The seven-byte field for a string descriptor is stored in the scalar variable pool in the format illustrated in Figure B-10.

Figure B—10. String Descriptor

CHAR1	CHAR2	LENGTH	POINTER	0	0
+0	+128		LSB MSB		

The first two bytes of the descriptor contain the first two characters of the variable name, with 128 added to the ASCII value of the second character. If the variable has a name of one character (for example, S$), the second character in the variable is set to the value of 128. The dollar sign is not stored as part of the name.

The length byte contains the current length of the string. The pointer to the string points to either the BASIC statement string definition (for instance, S$="STRING") or to the location of the string in the string pool. Strings may be transferred to the string pool from the program by modifying them in any way, such as S$=S$+"" Take a look at Figure B-11 for an example of a string descriptor, its format, and its values in that format.

Figure B—11. String Descriptor Example

S$="STRING"+""

CHAR1	CHAR2	LENGTH	POINTER	0	0
83	128	6	25029	0	0

Since the string was modified during its definition, it has been stored in the string variable.

Array Variables

The array variables stored in the array pool share a common format of header information, as illustrated in Figure B-12.

Figure B—12. BASIC Array Variable Header

CHAR1	CHAR2	LENGTH	DIMENSIONS	SUBN	...	SUB1
		LSB MSB		MSB LSB	...	MSB LSB

Appendix B

The first two bytes of the descriptor contain the first two characters of the variable name, following the previously described conventions for scalar variables. The length bytes contain the total size of the array. The number-of-dimensions byte indicates how many subscripts must be used to reference the array, among other uses. There are then two bytes for every dimension (starting with the last and ending with the first) that contain the count of elements in the dimension, plus one. These counts are in *reverse* format, MSB/LSB.

Figure B-13 is an example of an array variable, complete with values.

Figure B—13. Array Example

DIM AA(7,3,5)

CHAR1	CHAR2	LENGTH	DIMENSIONS	SUB3	SUB2	SUB1
65	65	203 3	3	0 6	0 4	0 8

After the array descriptor header, the variables are stored in the formats previously described, but without any padding to fill out the seven bytes. An integer is stored in two bytes, a floating point number in five bytes, and a string descriptor is stored in three bytes. The variables are stored in ascending order. For example, the variables in DIM G(1,1,2) would be stored in the order: (0,0,0), (1,0,0), (0,1,0), (1,1,0), (0,0,1), (1,0,1), (0,1,1), (1,1,1), (0,0,2), (1,0,2), (0,1,2), (1,1,2).

When an array is defined, it's stored in the array pool above the scalar variables. When the next scalar variable is to be created, all the arrays must be pushed up seven bytes to make room. Once a variable has been defined, it takes up pool space and is present until a CLR is issued. String variables can be redefined to a shorter length, but this frees storage space only if they were in the string variable pool and not being pointed to in the BASIC statement that defined them. The seven-byte descriptor for the string is still, obviously, in the scalar variable pool.

The routine that locates variables starts searching for them at the start of the appropriate variable pool and works upward to the end of the pool. The time needed to find a variable can be reduced by defining it before other less-used variables.

VIC-20 Code Chart

This chart of VIC-20 codes will allow you to determine the appropriate code number to use for achieving the desired control, screen, and character effects. Further, the chart allows the rapid translation of a code number into its various meanings within the VIC-20. You'll find the chart quite complete and easy to use.

The chart lines are numbered from 0 to 255 decimal on both sides of the page, with the equivalent hexadecimal number listed just inside the decimal number.

The third and fourth columns list the character sets and associated control codes that correspond to the decimal and hexadecimal numbers on that line. Look at line 156 and you'll notice that a PRINT CHR$(156) will set the foreground color code to purple, while on line 169 we see that PRINT CHR$(169) will print a square halved diagonally, or a checkerboard square, depending on the character set that is currently selected.

Line 14 shows that PRINT CHR$(14) will select the lowercase character set (labeled SET2 on the chart) and PRINT CHR$(142) enables uppercase mode (SET1).

The special character codes recognized by the VIC 1515/1525 printer are:

Table C-1. VIC 1515/1525 Printer Codes

CHR$()	Meaning
8	Graphic dot mode
10	Linefeed
13	Carriage return
14	Double-sized characters
15	Normal-sized characters
16	Tab
17	Lowercase
18	Reverse on
26	Repeat
27	Dot addressing
145	Uppercase
146	Reverse off

Consult the printer manual for further information regarding the use of these special printer codes.

The column on the Code Chart labeled *BASIC Token* is used when examining the internal storage format of BASIC programs, either in memory, on tape, or on disk.

313

Appendix C

The *Screen POKE* column can be used to determine the needed code for a character when you're storing characters directly into the screen RAM map. There are sample programs (see location 4096, $1000) for converting between CHR$ and screen POKE codes, when you need to do this from within a program.

The binary bit representation of any desired byte contents can be determined by consulting the *Binary* column for that hexadecimal or decimal number. You'll find this useful when designing your own custom character sets; determining the effects of AND, OR, NOT, and WAIT operations; finding the number to use to set or turn off a particular bit in memory; as well as a host of other uses. Appendix B discusses number systems and includes a program to perform decimal/hexadecimal/binary conversions, as well as subroutines that you can include in your programs.

True ASCII shows the interpretation of each character by a host computer or a non-CBM printer. In almost all cases, *True ASCII* is extended or modified, but it's important to understand the base character set.

The ML 6502 operation code mnemonic that corresponds to each possible byte contents is listed under *6502 Opcode*. Suffixes are appended to this column to denote the addressing mode implied. No suffix indicates that the instruction operates in the absolute, relative, or native addressing mode, depending on the instruction's purpose. A Z suffix indicates that zero page is referenced, and *I* is used to show an indirect addressing mode. Either I or Z can be accompanied with X or Y indexes. The absolute addressing mode also allows these indexes. An *A* suffix is present when the instruction operates on the accumulator, unless that is explicit in the operation code. *IM* marks the immediate-mode instructions. For example, ADC-IX indicates that indirect, indexed addressing is implied by the instruction. Some opcodes that are not listed can function as unsupported opcodes that do rather strange combinations of supported opcodes. These are not recommended, but it's often interesting to explore their effects.

The column titled *MSB* is used to determine the MSB address of a two-byte LSB/MSB address. Look up the MSB number from the address. This column shows the result of multiplying the MSB byte by 256. For example, the address 04/53 in LSB/MSB format would be resolved by looking up 53 in the chart (first determine if 04/53 is represented in hexadecimal or decimal—PEEK gives decimal results); 53 decimal is MSB 13568 (53*256). Now simply add the 04 from the LSB and the address is complete: (13568+4=13572). This column can also be used to perform the reverse operation of changing an address like 47874 into 02/187 ($02/BB) LSB/MSB format.

Table C—1. VIC—20 Code Chart

Dec Hex	CHR$ Set1	CHR$ Set2	BASIC Token	Screen Set1	POKE Set2	Binary	True ASCII	6502 Opcode	MSB Addr
0 00				@	@	0000.0000	NULL	BRK	00
1 01				A	a	0000.0001	SOH	ORA-IX	256
2 02				B	b	0000.0010	STX		512
3 03				C	c	0000.0011	ETX		768
4 04				D	d	0000.0100	EOT		1024
5 05	WHT			E	e	0000.0101	ENQ	ORA-Z	1280
6 06				F	f	0000.0110	ACK	ASLZ	1536
7 07				G	g	0000.0111	BEL		1792
8 08	DISABLE2			H	h	0000.1000	BS	PHP	2048
9 09	ENABLE2			I	i	0000.1001	HT	ORA-IM	2304
10 0A	LINEFEED			J	j	0000.1010	LF	ASL-A	2560
11 0B				K	k	0000.1011	VT		2816
12 0C				L	l	0000.1100	FF		3072
13 0D	RETURN			M	m	0000.1101	CR	ORA	3328
14 0E	LOWCASE			N	n	0000.1110	SO	ASL	3584
15 0F				O	o	0000.1111	SI		3840
16 10				P	p	0001.0000	DLE	BPL	4096
17 11	CRSRDN			Q	q	0001.0001	DC1	ORA-IY	4352
18 12	RVS ON			R	r	0001.0010	DC2		4608
19 13	HOME			S	s	0001.0011	DC3		4864
20 14	DEL			T	t	0001.0100	DC4		5120
21 15				U	u	0001.0101	NAK	ORA-ZX	5376
22 16				V	v	0001.0110	SYN	ASL-ZX	5632
23 17				W	w	0001.0111	ETB		5888
24 18				X	x	0001.1000	CAN	CLC	6144
25 19				Y	y	0001.1001	EM	ORA-Y	6400
26 1A	REPEAT			Z	z	0001.1010	SUB		6656
27 1B	DOTADDR			[	[	0001.1011	ESC		6912
28 1C	RED			£	£	0001.1100	FS		7168
29 1D	CRSRRT			]	]	0001.1101	GS	ORA-X	7424
30 1E	GRN			↑	↑	0001.1110	RS	ASL-X	7680
31 1F	BLU			←	←	0001.1111	US		7936
32 20	SPACE			SPACE		0010.0000	SPACE	JSR	8192
33 21	!	!		!	!	0010.0001	!	AND-IX	8448
34 22	QUOTES			QUOTES		0010.0010		QUOTES	8704
35 23	#	#		#	#	0010.0011	#		8960
36 24	$	$		$	$	0010.0100	$	BITZ	9216
37 25	%	%		%	%	0010.0101	%	ANDZ	9472
38 26	&	&		&	&	0010.0110	&	ROLZ	9728
39 27	'	'		'	'	0010.0111	'		9984
40 28	(	(		(	(	0010.1000	(	PLP	10240
41 29	)	)		)	)	0010.1001	)	AND-IM	10496
42 2A	*	*		*	*	0010.1010	*	ROL-A	10752
43 2B	+	+		+	+	0010.1011	+		11008
44 2C	,	,		,	,	0010.1100	,	BIT	11264
45 2D	—	—		—	—	0010.1101	–	AND	11520
46 2E	.	.		.	.	0010.1110	.	ROL	11776
47 2F	/	/		/	/	0010.1111	/		12032
48 30	0	0		0	0	0011.0000	0	BMI	12288
49 31	1	1		1	1	0011.0001	1	AND-IY	12544
50 32	2	2		2	2	0011.0010	2		12800
51 33	3	3		3	3	0011.0011	3		13056

315

Appendix C

		CHR$	CHR$	BASIC	Screen	POKE	True	6502	MSB
Dec	Hex	Set1	Set2	Token	Set1	Set2	Binary ASCII	Opcode	Addr
52	34	4	4		4	4	0011.0100 4		13312
53	35	5	5		5	5	0011.0101 5	AND-ZX	13568
54	36	6	6		6	6	0011.0110 6	ROL-ZX	13824
55	37	7	7		7	7	0011.0111 7		14080
56	38	8	8		8	8	0011.1000 8	SEC	14336
57	39	9	9		9	9	0011.1001 9	AND-Y	14592
58	3A	:	:		:	:	0011.1010 :		14848
59	3B	;	;		;	;	0011.1011 ;		15104
60	3C	<	<		<	<	0011.1100 <	.	15360
61	3D	=	=		=	=	0011.1101 =	AND-X	15616
62	3E	>	>		>	>	0011.1110 >	ROL-X	15872
63	3F	?	?		?	?	0011.1111 ?		16128
64	40						0100.0000 @	RTI	16384
65	41	A	a			A	0100.0001 A	EOR-IX	16640
66	42	B	b			B	0100.0010 B		16896
67	43	C	c			C	0100.0011 C		17152
68	44	D	d			D	0100.0100 D		17408
69	45	E	e			E	l0100.0101 E	EOR-Z	17664
70	46	F	f			F	0100.0110 F	LSR-Z	17920
71	47	G	g			G	0100.0111 G		18176
72	48	H	h			H	0100.1000 H	PHA	18432
73	49	I	i			I	0100.1001 I	EOR-IM	18688
74	4A	J	j			J	0100.1010 J	LSR-A	18944
75	4B	K	k			K	0100.1011 K		19200
76	4C	L	l			L	0100.1100 L	JMP	19456
77	4D	M	m			M	0100.1101 M	EOR	19712
78	4E	N	n			N	0100.1110 N	LSR	19968
79	4F	O	o			O	0100.1111 O		20224
80	50	P	p			P	0101.0000 P	BVC	20480
81	51	Q	q			Q	0101.0001 Q	EOR-IY	20736
82	52	R	r			R	0101.0010 R		20992
83	53	S	s			S	0101.0011 S		21248
84	54	T	t			T	0101.0100 T		21504
85	55	U	u			U	0101.0101 U	EOR-ZX	21760
86	56	V	v			V	0101.0110 V	LSR-ZX	22016
87	57	W	w			W	0101.0111 W		22272
88	58	X	x			X	0101.1000 X	CLI	22528
89	59	Y	y			Y	0101.1001 Y	EOR-Y	22784
90	5A	Z	z			Z	0101.1010 Z		23040
91	5B	[	[			[	0101.1011 [		23296
92	5C	£	£			£	0101.1100 \		23552
93	5D	]	]			]	0101.1101]	EOR-X	23808
94	5E	↑	↑				0101.1110 ⇑	LSR-X	24064
95	5F	←	←				0101.1111 ←		24320
96	60	—	—		SPACE		0110.0000 SPACE	RTS	24576
97	61		A				0110.0001 a	ADC-IX	24832
98	62		B				0110.0010 b		25088
99	63		C				0110.0011 c		25344
100	64		D				0110.0100 d		25600
101	65		E				0110.0101 e	ADC-Z	25856
102	66		F				0110.0110 f	ROR-Z	26112
103	67		G				0110.0111 g		26368
104	68		H				0110.1000 h	PLA	26624
105	69		I				0110.1001 i	ADC-IM	26880
106	6A		J				0110.1010 j	ROR-A	27136
107	6B		K				0110.1011 k		27392
108	6C	L	L				0110.1100 l	JMP-I	27648

Dec Hex	CHR$ Set1	CHR$ Set2	BASIC Token	Screen Set1	POKE Set2	True Binary ASCII	6502 Opcode	MSB Addr
109 6D		M				0110.1101 m	ADC	27904
110 6E		N				0110.1110 n	ROR	28160
111 6F		O				0110.1111 o		28416
112 70		P				0111.0000 p	BVS	28672
113 71		Q				0111.0001 q	ADC-IY	28928
114 72		R				0111.0010 r		29184
115 73		S				0111.0011 s		29440
116 74		T				0111.0100 t		29696
117 75		U				0111.0101 u	ADC-ZX	29952
118 76		V				0111.0110 v	ROR-ZX	30208
119 77		W				0111.0111 w		30464
120 78		X				0111.1000 x	SEI	30720
121 79		Y				0111.1001 y	ADC-Y	30976
122 7A		Z				0111.1010 z		31232
123 7B		+				0111.1011 {		31488
124 7C						0111.1100 ¦		31744
125 7D						0111.1101 }	ADC-X	32000
126 7E						0111.1110 "	ROR-X	32256
127 7F						0111.1111 DEL		32512
128 80			END			1000.0000		32768
129 81			FOR			1000.0001	STA-IX	33024
130 82			NEXT			1000.0010		33280
131 83	LOAD/RUN		DATA			1000.0011		33536
132 84			INPUT#			1000.0100	STY-Z	33792
133 85	f1		INPUT			1000.0101	STA-Z	34048
134 86	f3		DIM			1000.0110	STX-Z	34304
135 87	f5		READ			1000.0111		34560
136 88	f7		LET			1000.1000	DEY	34816
137 89	f2		GOTO			1000.1001		35072
138 8A	f4		RUN			1000.1010	TXA	35328
139 8B	f6		IF			1000.1011		35584
140 8C	f8		RE-STORE			1000.1100	STY	35840
141 8D	SHRTRN		GOSUB			1000.1101	STA	36096
142 8E	UPCASE		RETURN			1000.1110	STX	36352
143 8F			REM			1000.1111		36608
144 90	BLK		STOP			1001.0000	BCC	36864
145 91	CRSRUP		ON			1001.0001	STA-IY	37120
146 92	RVSOFF		WAIT			1001.0010		37376
147 93	CLR		LOAD			1001.0011		37632
148 94	INST		SAVE			1001.0100	STY-ZX	37888
149 95			VERIFY			1001.0101	STA-ZX	38144
150 96			DEF			1001.0110	STX-ZY	38400
151 97			POKE			1001.0111		38656
152 98			PRINT#			1001.1000	TYA	38912
153 99			PRINT			1001.1001	STA-Y	39168
154 9A			CONT			1001.1010	TXS	39424
155 9B			LIST			1001.1011		39680
156 9C	PUR		CLR			1001.1100		39936
157 9D	CRSRLFT		CMD			1001.1101	STA-X	40192
158 9E	YEL		SYS			1001.1110		40448
159 9F	CYN		OPEN			1001.1111		40704
160 A0	SHSPACE		CLOSE			1010.0000	LDY-IM	40960
161 A1			GET			1010.0001	LDA-IX	41216
162 A2			NEW			1010.0010	LDX-IM	41472
163 A3			TAB(			1010.0011		41728

Appendix C

Dec Hex	CHR$ Set1	CHR$ Set2	BASIC Token	Screen/POKE Set1	Screen/POKE Set2	True Binary ASCII	6502 Opcode	MSB Addr
164 A4			TO			1010.0100	LDY-Z	41984
165 A5			FN			1010.0101	LDA-Z	42240
166 A6			SPC(			1010.0110	LDX-Z	42496
167 A7			THEN			1010.0111		42752
168 A8			NOT			1010.1000	TAY	43008
169 A9			STEP			1010.1001	LDA-IM	43264
170 AA			+			1010.1010	TAX	43520
171 AB			–			1010.1011		43776
172 AC			*			1010.1100	LDY	44032
173 AD			/			1010.1101	LDA	44288
174 AE			↑			1010.1110	LDX	44544
175 AF			AND			1010.1111		44800
176 B0			OR			1011.0000	BCS	45056
177 B1			>			1011.0001	LDA-IY	45312
178 B2			=			1011.0010		45568
179 B3			<			1011.0011		45824
180 B4			SGN			1011.0100	LDY-ZX	46080
181 B5			INT			1011.0101	LDA-ZX	46336
182 B6			ABS			1011.0110	LDX-ZY	46592
183 B7			USR			1011.0111		46848
184 B8			FRE			1011.1000	CLV	47104
185 B9			POS			1011.1001	LDA-Y	47360
186 BA			SQR			1011.1010	TSX	47616
187 BB			RND			1011.1011		47872
188 BC			LOG			1011.1100	LDY-X	48128
189 BD			EXP			1011.1101	LDA-X	48384
190 BE			COS			1011.1110	LDX-Y	48640
191 BF			SIN			1011.1111		48896
192 C0			TAN			1100.0000	CPY-IM	49152
193 C1			ATN			1100.0001	CMP-IX	49408
194 C2			PEEK			1100.0010		49664
195 C3			LEN			1100.0011		49920
196 C4			STR$			1100.0100	CPY-Z	50176
197 C5			VAL			1100.0101	CMP-Z	50432
198 C6			ASC			1100.0110	DEC-Z	50688
199 C7			CHR$			1100.0111		50944
200 C8			LEFT$			1100.1000	INY	51200
201 C9			RIGHT$			1100.1001	CMP-IM	51456
202 CA			MID$			1100.1010	DEX	51712
203 CB			GO			1100.1011		51968
204 CC						1100.1100	CPY	52224
205 CD						1100.1101	CMP	52480
206 CE						1100.1110	DEC	52736
207 CF						1100.1111		52992
208 D0						1101.0000	BNE	53248
209 D1						1101.0001	CMP-IY	53504
210 D2						1101.0010		53760
211 D3						1101.0011		54016
212 D4						1101.0100		54272
213 D5						1101.0101	CMP-ZX	54528
214 D6						1101.0110	DEC-ZX	54784
215 D7						1101.0111		55040
216 D8						1101.1000	CLD	55296
217 D9						1101.1001	CMP-Y	55552
218 DA						1101.1010		55808
219 DB						1101.1011		56064

318

Dec Hex	CHR$ Set1	CHR$ Set2	BASIC Token	Screen Set1	POKE Set2	True Binary ASCII	6502 Opcode	MSB Addr
220 DC						1101.1100		56320
221 DD						1101.1101	CMP-X	56576
222 DE						1101.1110	DEC-X	56832
223 DF						1101.1111		57088
224 E0						1110.0000	CPX-IM	57344
225 E1						1110.0001	SBC-IX	57600
226 E2						1110.0010		57856
227 E3						1110.0011		58112
228 E4						1110.0100	CPX-Z	58368
229 E5						1110.0101	SBC-Z	58624
230 E6						1110.0110	INC-Z	58880
231 E7						1110.0111		59136
232 E8						1110.1000	INX	59392
233 E9						1110.1001	SBC-IM	59648
234 EA						1110.1010	NOP	59904
235 EB						1110.1011		60160
236 EC						1110.1100	CPX	60416
237 ED						1110.1101	SBC	60672
238 EE						1110.1110	INC	60928
239 EF						1110.1111		61184
240 F0						1111.0000	BEQ	61440
241 F1						1111.0001	SBC-IY	61696
242 F2						1111.0010		61952
243 F3						1111.0011		62208
244 F4						1111.0100		62464
245 F5						1111.0101	SBC-ZX	62720
246 F6						1111.0110	INC-ZX	62976
247 F7						1111.0111		63232
248 F8						1111.1000	SED	63488
249 F9						1111.1001	SBC-Y	63744
250 FA						1111.1010		64000
251 FB						1111.1011		64256
252 FC						1111.1100		64512
253 FD						1111.1101	SBC-X	64768
254 FE						1111.1110	INC-X	65024
255 FF						1111.1111		65280

Appendix D

Device Numbers, Secondary Addresses, and Status Codes

This appendix summarizes the information in the body of the map regarding device numbering, secondary address considerations, and status codes returned by the Kernal when it performs I/O and related information.

Within the following chart are note indicators denoted by [*n*], where *n* is the number of the note in the section following the chart. Following the chart are examples illustrating the use of device numbers and secondary addresses. After the example section is a summary of the memory locations related to this subject.

Table D—1. Device Number, Secondary Address, and Status Code

Number = Device		SecAddr	Meaning	ST Meaning
0	KEYBOARD	[1]	[1]	[2]
1	TAPE	0/1 [0]	READ	4 Short block
1	TAPE	0/1 [0]	READ	8 Long block
1	TAPE	0/1 [0]	READ	16 Unrecoverable error or VERIFY mismatch
1	TAPE	0/1 [0]	READ	32 Checksum error
1	TAPE	0/1 [0]	READ	64 End of file
1	TAPE	0/1 [0]	READ	−128 EOT [3]
1	TAPE	1 [0]	WRITE	[2]
1	TAPE	2 [0]	WRITE+EOT [3]	[2]
2	[4] RS-232	[1]	[5]	128 BREAK detected
2	[4] RS-232	[1]	[5]	64 Dataset-ready missing
2	[4] RS-232	[1]	[5]	16 Clear-to-send missing [6]
2	[4] RS-232	[1]	[5]	4 Receive buffer overrun
2	[4] RS-232	[1]	[5]	2 Framing error
2	[4] RS-232	[1]	[5]	1 Parity error
3	SCREEN	[1]	[1]	[2]
4/5	PRINTER	0	UPPERCASE	1 Write timeout
4/5	PRINTER	0	UPPERCASE	−128 Device not present
4/5	PRINTER	7	LOWERCASE	1 Write timeout
4/5	PRINTER	7	LOWERCASE	−128 Device not present
8–11	DISK	0	LOAD	[7]
8–11	DISK	0	LOAD	2 Read timeout
8–11	DISK	0	LOAD	−128 Device not present
8–11	DISK	1	SAVE	[7]
8–11	DISK	1	SAVE	1 Write timeout
8–11	DISK	1	SAVE	−128 Device not present

Number = Device		SecAddr	Meaning	ST Meaning
8–11	DISK	2–14	DATA CHANNEL	[7]
8–11	DISK	2–14	DATA CHANNEL	1 Write timeout
8–11	DISK	2–14	DATA CHANNEL	64 Read timeout
8–11	DISK	2–14	DATA CHANNEL	64 End of input (last item)
8–11	DISK	2–14	DATA CHANNEL	–128 Device not present
8–11	DISK	15	DOS CHANNEL	[7]
8–11	DISK	15	DOS CHANNEL	1 Write timeout
8–11	DISK	15	DOS CHANNEL	2 Read timeout
8–11	DISK	15	DOS CHANNEL	–128 Device not present
4–30	SERIAL DEVICE	0–31	Per device	1 Write timeout
4–30	SERIAL DEVICE	0–31	Per device	2 Read timeout
4–30	SERIAL DEVICE	0–31	Per device	64 End of input (last item)
4–30	SERIAL DEVICE	0–31	Per device	–128 Device not present

Notes

0 Tape secondary addresses are not used in exactly the same way throughout the VIC-20. In an OPEN statement, a 0 indicates READ, a 1 specifies WRITE, and a 2 means WRITE with EOT. When SAVEing or the Kernal's SETLFS routine is called in preparation for a SAVE routine call, an odd-numbered secondary address causes the tape header I.D. to be set to indicate that the program file cannot be relocated. An even-numbered secondary address causes the program file to be relocatable. Adding 2 to the secondary address causes a tape EOT header to also be written.

For LOAD, or the SETLFS routine prior to calling the LOAD routine, an odd secondary address causes the header information (the start of save pointer) to be used to reload the program. For an even-numbered secondary address, an address must be provided that specifies the desired load point. However, even if an even number is passed to SETLFS to indicate a relocatable load and an address is provided for the starting point, if the program was saved as nonrelocatable (via a secondary address that was odd at the time of saving, causing a tape header I.D. 3), it will be loaded back to the original location it resided at when it was saved. See location 828 ($33C) in the memory map for more information about tape header identifiers.

Appendix D

1 The secondary address has no meaning for this device and 0–255 can be used. In the case of devices 0 and 3, the parameter can simply be omitted on the OPEN statement. Omitting the parameter or specifying 0 for other devices inhibits the sending of any following string. User port attached devices can themselves require certain secondary addresses. However, the VIC Modem does not.

2 No status codes are returned.

3 EOT is a tape header with an I.D. of 5 used to indicate that no more files exist on the tape. For additional details, refer to the explanation at location 828 ($33C).

4 The user port is typically used for RS-232 protocol devices, such as modems and printers, but can be used for a variety of other devices as well. For instance, joysticks, remote electrical switching devices, and analog sensors may be connected. You can program the first VIA in the VIC-20 to accomplish a wide range of device support functions.

5 User port functions, including data direction, are controlled by VIA1. RS-232 protocol on the user port is determined by the Kernal RS-232 routines and OPEN settings of the RS-232 control register (described at location 659, $293) and the RS-232 command register (described at location 660, $294).

6 This status code will never be set. See the explanation at location 660 ($294). General RS-232 error recovery suggestions are discussed at location 663 ($297).

7 The DOS channel (secondary address of 15) is used to send commands to the disk resident DOS and to receive error status. Four variables are returned by the DOS channel in response to an INPUT#15,A,B$,C,D. These are: error number (0 means none), error description, track number, and block (sector) number. Consult the *VIC-1541 User's Manual* for details of sending commands to DOS, receiving the error status variables, and a table of the possible error numbers.

Examples

Ten files may be opened at any one time, only five of them being serial (disk/printer) files. File numbers greater than 127 cause a linefeed character to be sent after carriage returns. This feature is designed for non-CBM printers.

The OPEN command parameters are:

OPEN file number,device number,secondary address,"filename or disk command"

● OPEN 16 causes the default device of 1 and secondary address of

0 to be used. The tape will be opened for a read operation. File number is the only mandatory OPEN parameter.

- OPEN 1,0 prepares the keyboard for input. An attempt to PRINT#1 will receive a NOT OUTPUT FILE error message.
- OPEN 1,1 opens the next found tape file for input since the secondary address defaults to zero.
- OPEN 1,1,0,"FAST GAME" searches the tape for the specified filename. Notice that the secondary address was needed in order to specify the filename. OPEN 1,1,,"FAST GAME" could have been used as well.
- OPEN 1,1,1,"ACCOUNTS" causes the creation of a tape header with I.D. 4, indicating that BASIC program or ML generated data is being written to tape. See the explanation of the tape header I.D. at location 828 ($33C).
- OPEN 1,1,2,"HOME INVENTORY" also creates the same type of tape header. Additionally, when the file is closed, an I.D. 5 header will be written, indicating that there are no more files on the tape, whether there actually are or not.
- OPEN 2,2,0,CHR$(6+32)+CHR$(32+16) sets the RS-232 control options to 300 baud and a word length of seven bits, while the RS-232 command options are being set to odd parity and half duplex.
- OPEN 3,3 allows the screen to be used for input or output operations.
- OPEN 4,4 opens the printer in uppercase mode, since the default secondary address is zero.
- OPEN 10,8,12,"0:STOCK ISSUES,S,W" prepares the 0 disk (the zero and colon are optional on the VIC 1540 and 1541) to write by virtue of the ,W parameter, a sequential file (specified by the ,S,) using data channel 12, referred to as file 10 within the program. It may be convenient to use the same file number as the data channel number in order to avoid confusion. Besides ,S, for file type, you can specify ,P, for program and ,U, for user files. The ,W parameter would be ,R when reading the file. Consult the *VIC-1541 User's Manual* for additional details.
- OPEN 15,8,15,"V" requests that the DOS channel be used to send the *V* command (VALIDATE) to DOS for processing. The disk directory will be reorganized by DOS in response to this command.

Related Memory Locations

The following memory locations provide additional information on the subject of device numbers, secondary addresses, and status codes.

| 19 | ($13) | Current channel number for BASIC input/output routines |

Appendix D

144	($90)	ST status register
152	($98)	Number of currently open files
153–154	($99–9A)	Device number of the current input file
154	($9A)	Device number of the current output file
183	($B7)	Number of characters in the filename
184	($B8)	Current logical file number
185	($B9)	Current secondary address being used
186	($BA)	Current device number being used
187–188	($BB–BC)	Pointer to the current filename
601–610	($259–262)	Open file number table
611–620	($263–26C)	Open device number table
621–630	($26D–276)	Open secondary address table
663	($297)	RS-232 ST status register

Automatic and User Relocation of Memory Contents

Three important aspects of the relocation of VIC-20 memory contents are explored in this appendix: the relocations performed automatically by the Kernal when the VIC-20 is powered-on (or reset) with expansion memory plugged in, methods for programs to test and adjust to the standard memory expansion configurations, and considerations when relocating the memory areas to suit your program's needs.

Automatic Relocation by the Kernal

When the VIC-20 is turned on (or a reset switch is pressed on a memory expansion board), the Kernal measures the amount of RAM available and places the BASIC program area, screen map, and color at positions in memory that allow the largest continuous block of memory for the BASIC program area, consistent with restrictions imposed by the addressing scheme used in the 6560 VIC chip. The following table summarizes the major locations affected when adding memory expansion.

Table E–1. Effect of Adding Memory Expansion to the VIC

Expansion Added	Total RAM	Bytes Free	Color Map	Screen Map	BASIC Start	BASIC End
0K	5K	3583	38400	7680	4097	7680
3K	8K	6655	38400	7680	1025	7680
8K	13K	11775	37888	4096	4609	16384
8K+3K	16K	11775	37888	4096	4609	16384
16K	21K	19967	37888	4096	4609	24576
16K+3K	24K	19967	37888	4096	4609	24576
24K	29K	28159	37888	4096	4609	32768
24K+3K	32K	28159	37888	4096	4609	32768
32K	37K	28159	37888	4096	4609	32768
32K+3K	40K	28159	37888	4096	4609	32768

The BASIC start pointer is located at 43–44 ($2B–2C) and the BASIC end pointer is at location 55–56 ($37–38). The Kernal pointer at 641–642 ($281–282) (bottom of RAM) is normally pointing one byte before the BASIC start pointer. The Kernal pointer at 643–644 ($283–284) is normally pointing at the same location as the BASIC end pointer. Appendix B discusses the pointers used by BASIC to

325

Appendix E

manage the various variable pools. If a Super Expander cartridge is used in place of a normal 3K expansion memory board, subtract 135 bytes from the Bytes Free column and from the BASIC End column.

(The term K means 1024 bytes. Expansion memory is usually added in increments of 8K or 16K, except for a single 3K expansion.)

The effects of memory expansion can be summarized as:

- 3K expansion doesn't cause relocation of any areas except the start of the BASIC program.
- Adding 8K or more expansion causes the BASIC program area, the screen map, and the color map to be relocated, but adding additional expansion does not cause further relocation schemes.
- Once 8K or more expansion is added, BASIC won't see or use any 3K expansion.
- Adding expansion memory above 24K (to either the 8K area at 40960 ($A000) and/or to the 3K expansion area) does not increase the amount of RAM available to store a BASIC program. RAM areas that BASIC cannot use for program storage can be used to store user data (with POKEs) or to contain machine language instructions.

Finding the Relocatable Areas

When writing a program, you can include instructions that determine the memory size and adjust the program to operate correctly when additional memory expansion is plugged in. If this were done by all program authors, the warning *Runs only on an unexpanded VIC* could be eliminated. The following line determines the memory expansion environment and sets the variables SM to the screen map location and CM to the location of the color map. Since the screen map location affects the value to be POKEd into location 36869 to cause a switch between the two possible character sets, PRINT CHR$(14) should be used to switch to the lowercase set and PRINT CHR$(142) to switch back to uppercase.

SM=7680:CM=38400:IF PEEK(648)=16 THEN SM=4096:CM=37888

The instructions set the default values for an unexpanded VIC, then test the screen memory page number that was set by the Kernal at power-on/reset. Location 641–642 ($281–282) in the memory map includes a routine that can be used to unexpand the VIC, making it temporarily forget any memory expansion that is plugged in. Instructions for reexpanding the memory are also included.

The screen memory starting location can also be determined by SM=PEEK(648)*256 and the color map can be found with CM=37888+4*(PEEK(36866)AND 128).

Relocation by the User

The subject of user relocation of the BASIC area, the screen map, custom character sets, and the color map has three major considerations which must be resolved before starting any relocation. First, how much expansion memory will be on the VIC? Second, remember that expansion memory is not addressable by the 6560 VIC chip and cannot be used for screen, color, or character memory (but is available for a BASIC program area), and that the VIC chip imposes restrictions on the starting address for the screen and character maps. Also, decide which areas need to be relocated in order to achieve the objectives of your program.

The third issue is not as straightforward as you might assume. Since nonexpansion RAM is only 4096 bytes in length (locations 4096 through 8191) and because that area is normally either the BASIC program area or a part of the BASIC area, plus the screen map (on an 8K+ expanded VIC-20), the relocation of one area to nonexpansion RAM may necessitate relocating the BASIC area and perhaps choosing a more limited custom character set than you would have preferred. It is also possible to expand or contract the amount of space required for the screen map. Contraction is often chosen on an unexpanded VIC-20.

A further complication is the demand by the VIC chip that screen and character maps begin on certain address boundaries. A table of possibilities is included in this appendix to help you determine the possible and desirable combinations. Before you begin any program that depends on relocated areas, you should take the time to consider these issues and sketch out the relocated memory configuration that you'll be working with.

Relocating BASIC

The first relocation of a memory area that you'll probably want to perform will involve lowering or raising the boundaries for the BASIC area. The storage of machine language programs in memory along with your BASIC program usually requires this adjustment of the BASIC area. Sometimes the tape buffer at 828 ($33C) provides some capability for this without adjusting BASIC pointers. In the direct mode,

POKE 642,PEEK(642)+NS:POKE 644,PEEK(644)−NL:SYS 58232

can be used to move the start of the BASIC area up NS pages, the end down NL pages, as well as to cold start BASIC, destroying any current program. By subtracting NS from 642, you can move the start downward. By adding to NL, you make the end point move upward. Locations 642 and 644 can also be set to the desired values directly, without the PEEK. A program is provided at location 55

Appendix E

($37) in the memory map to reserve space in the BASIC area from the top or the bottom.

Moving the BASIC area to an entirely different location is easily accomplished with the methods demonstrated in the program. Place the new values into the appropriate pointers rather than adding to or subtracting from them. At location 641–642 ($281–282) of the memory map is a routine that can be used to unexpand the VIC, making it temporarily forget any memory expansion that's plugged in. Instructions for reexpanding the memory are also included.

Relocating Character Sets

The character sets won't actually be relocated, but the goal is to provide an alternate set of characters to be used alone or in conjunction with the character maps provided in ROM. The number of characters in the custom character set will largely determine where you'll place it in memory. The wraparound method of addressing which is used by the VIC chip can allow you to address part of the standard character set as well as your own custom character set if you choose the address of your custom character set carefully. For the unexpanded VIC-20, lowering the top-of-BASIC pointer at 55–56 ($37–38) by 512 bytes and placing the custom character set there will achieve the wraparound effect.

COMPUTE!'s First Book of VIC includes an article, "Custom Character Sets for the VIC," by David Malmberg, which uses this technique for a custom character design aid. Custom character sets are needed when you tackle bitmapping of the screen, so it's a good place to start your exploration of relocatable memory areas, and it allows you to uniquely customize the screen displays of your programs. Creating custom characters for the VIC-20 is discussed at location 36869, the preamble to 32768, and in Appendix G.

The character map is used as a pixel map of the character for the corresponding screen POKE code. If you never use a screen code above value X, the character map will not be referenced above (8*X)+7 (or (16*X)+15) and may be used for other things. By starting the color map at 6144, for example, you could define up to 256 characters between 6144 and 8191. See location 36869 for details of how to set the pointer to the character map and the subject of wraparound, allowing access to the ROM maps as well as RAM character sets.

Table E-2 shows the possible character map locations on the VIC, as well as the number of characters that can be placed in each.

Table E—2. Character Maps

Character Map	Characters
4096–5119	1–128
4096–6143	1–256
4096–7167	1–384
4096–8191	1–512
5120–6143	1–128
5120–7167	1–256
5120–8191	1–384
6144–7167	1–128
6144–8191	1–256
7168–8191	1–128

Relocating the Screen and Color Map

These two areas are considered together because the color map location is dependent upon the screen map location. The primary reason for relocating the screen map is to free the area it's residing in for custom characters. Additionally, multiple screen maps can be used to quickly flip between screens. More than 23 lines of 22 columns can even be displayed on the TV screen if there is room in memory to expand the screen map.

The screen map location also determines which color map will be used. See location 36869 for additional details of this relationship. The VIC chip can be told to display fewer or more than 22 columns in 23 lines; the amount of screen RAM needed to display the specified lines and columns will depend on these settings of the VIC chip registers. If more than 512 bytes are to be displayed, you'll want to choose a screen RAM location that causes the color map to begin at 37888 so that the 38400 color map can be used for the color nybbles for the screen bytes beyond 512.

Table E-3 shows the possible screen map locations on the VIC, as well as the corresponding color map locations.

Table E—3. Screen Map–Color Map Locations

Screen Map	Color Map
4096–4607	37888
4608–5119	38400
5120–5631	37888
5632–6143	38400
6144–6655	37888
6656–7167	38400
7168–7679	37888
7680–8191	38400

Appendix E

Examples of User Relocation of Character and Screen Maps

In the article "High-Resolution Plotting," written by Paul F. Schatz and included in *COMPUTE!'s First Book of VIC*, the author chose (for an unexpanded VIC-20) locations 4096–5631 for BASIC, 5632–6143 for the screen, and 6144–8191 for custom characters. He demonstrated a 16-line by 16-column bitmapped screen display. For an expanded VIC-20, he chose a 22-column by 20-line display: 4096–7615 holds the character map (220 characters, each 8*16 size), 7680–8191 is used for the screen map, and BASIC is in expansion memory. The Super Expander cartridge also uses this arrangement of screen and character memory.

An alternate arrangement, with different goals in mind, was chosen in "Creating Graphics on an Expanded VIC," by Ed Harris, in the February 1983 issue of *COMPUTE!*. The screen map is relocated to 4096–4607; a 384-character set, with wraparound, was placed at 5120–8191; and BASIC started at 8192 (an 8K+ expansion).

Table E-4 illustrates some typical locations you could use when you relocate the character map, the screen map, and BASIC.

Table E–4. Typical Relocations

Unexpanded
6656 BASIC
6144 Screen Map
4096 Character Map

6144 Character Map
5632 Screen Map
4096 BASIC

Expanded
8192 BASIC
7680 Screen Map
7168 Character Map

8192 BASIC
7168 Character Map
4096 Screen Map

8192 BASIC
7680 Screen Map
4096 Character Map

8192 BASIC
5120 Character Map
4096 Screen Map

Block SAVE/LOAD Using the Kernal Routines from BASIC

This program demonstrates the techniques that can be used to call Kernal routines from a BASIC program by passing the required parameters in the SYS register SAVE area. It also serves as a model to be enhanced or extracted from to suit your particular purposes. As currently coded, the program provides block SAVE and LOAD capabilities similar to the LOAD and SAVE BASIC commands or a machine language monitor such as VICMON. A block of memory may be saved to tape in either a relocatable or nonrelocatable form, later to be reloaded with the same program. Variables define the filename, the address range to be saved, whether the format of the tape is to allow relocation or not, and whether an end-of-tape header should be written following the block of data.

   The program could be enhanced in several ways. It could be changed to perform verification on the saved data, support the disk, supply prompts for the user, or to display instructions for the creation of an external tape label with the filename, address range, and relocatability indicator.

   You could also extract part of this program to use as short routines in your own programs. The 25-line SAVE routine can easily be compressed to a five- or six-line subroutine, for example.

   The subject of relocatable saved blocks may need more clarification. If RELO$=NO is specified for the save operation, the data saved in a program type file format can't be relocated during a subsequent LOAD, even if RELO$=YES is requested during the load operation. If RELO$=YES is specified when saved, the later LOAD may request either YES or NO. When RELO$=YES is coded during a LOAD, the FROM= variable tells the LOAD routine the starting address of where the data should be loaded. The LAST= variable has no meaning during a load operation.

Program F-1. Block SAVE/LOAD

```
100 REM ******** BLOCK SAVE / LOAD USING THE KERNA
    L ROUTINES ********
110 REM MAPPING THE VIC-20{13 SPACES}G.R.DAVIES
    {14 SPACES}8/8/83
120 :
130 ACT$="SAVE"
140 NAME$="A.FILE"
150 FROM=24576
```

```
160 LAST=24678
170 RELO$="YES"
180 EOT$="NO"
190 FILE=11
200 LAST=LAST+1
210 NAME$=LEFT$(NAME$,94) : REM TRUNCATE NAME TO 9
    4 CHARS
220 NL=LEN(NAME$)
230 :
240 IF ACT$ = "SAVE" AND LAST < FROM THEN PRINT "
    {RVS}END < START":END
250 IF ACT$ = "SAVE" AND LAST > 32767 THEN PRINT"
    {RVS}CANT SAVE TO TAPE ABOVE 32767":END
260 :
270 PRINT"{RVS}"ACT$:PRINT"{RVS}NAME="CHR$(34);NAM
    E$;CHR$(34)
280 PRINT"{RVS}START="FROM:PRINT"{RVS}END="LAST:PR
    INT"{RVS}RELO="RELO$" EOT="EOT$
290 PRINT "O.K.? (Y/N)"
300 GET X$:IF X$="" GOTO300
310 IF X$="N" THEN END
320 IF X$<>"Y" GOTO 300
330 :
340 POKE 144,0 : REM RESET STATUS
350 IF ACT$ = "SAVE" GOTO 400
360 IF ACT$ = "LOAD" THEN 670
370 PRINT"{RVS}ACT$ NOT = SAVE OR LOAD":END
380 :
390 ***** SAVE *****
400 POKE 780,128+64 : REM SET KERNAL MSGS ON
410 SYS 65424 : REM CALL SETMSG
420 :
430 POKE 780,FILE : REM FILE NUMBER
440 POKE 781,1{3 SPACES}: REM DEVICE
450 SA=0:IF RELO$= "NO" THEN SA=1
460 IF EOT$ = "YES" THEN SA=SA+2
470 POKE 782,SA : REM SECONDARY ADDRESS
480 SYS 65466 : REM CALL SETLFS
490 :
500 POKE 780,NL : REM NAME LENGTH
510 IF NAME$<>"" THEN FORX=1TONL:POKE672+X,ASC( MI
    D$(NAME$,X,1) ):NEXT
520 POKE 781,161: REM LSB OF NAME PTR
530 POKE 782,2 : REM MSB OF NAME PTR
540 SYS 65469 : REM CALL SETNAM
550 :
560 POKE 780,251 :REM PAGE ZERO OFFSET TO START OF
     SAVE POINTER
570 POKE 252,INT(FROM/256) : REM MSB OF START OF S
    AVE
```

```
580 POKE 251,FROM-PEEK(252)*256 : REM LSB OF START
    OF SAVE
590 POKE 782,INT(LAST/256) : REM MSB OF END PLUS 1
600 POKE 781,LAST-PEEK(782)*256 : REM LSB OF END O
    F SAVE +1
610 SYS 65496 : REM CALL SAVE
620 :
630 IF ST >< 0 THEN PRINT"{2 SPACES}{RVS}ST="ST:EN
    D
640 PRINT:PRINT "* SAVE COMPLETED *":END
650 :
660 ***** LOAD *****
670 POKE 780,128+64 : REM SET KERNAL MSGS ON
680 SYS 65424 : REM CALL SETMSG
690 :
700 POKE 780,FILE : REM FILE NUMBER
710 POKE 781,1{3 SPACES}: REM DEVICE
720 SA=0:IF RELO$= "NO" THEN SA=1 :REM USE HDR
730 POKE 782,SA : REM SECONDARY ADDRESS
740 SYS 65466 : REM CALL SETLFS
750 :
760 POKE 780,NL : REM NAME LENGTH
770 IF NAME$<>"" THEN FORX=1TONL:POKE672+X,ASC( MI
    D$(NAME$,X,1) ):NEXT
780 POKE 781,161: REM LSB OF NAME PTR
790 POKE 782,2 : REM MSB OF NAME PTR
800 SYS 65469 : REM CALL SETNAM
810 :
820 POKE 780,0 :REM LOAD INDICATOR
830 POKE 782,INT(FROM/256) : REM MSB OF START OF L
    OAD
840 POKE 781,FROM-PEEK(782)*256 : REM LSB OF START
     OF LOAD
850 SYS 65493 : REM CALL LOAD
860 :
870 REM{2 SPACES}IF ST = 36 THEN POKE 144,0 : REM
    {SPACE}36 MAY BE NORMAL
880 IF ST >< 0 THEN PRINT" {RVS}ST="ST"{OFF}"
890 IF ST <> 0 THEN PRINT" {RVS}LOAD ERROR":END
900 PRINT:PRINT"* LOAD COMPLETED *" : END
```

Appendix G

Custom Characters and Bitmapping

Custom Characters

The ability to define your own custom character sets is a powerful tool for extending the display capabilities for the VIC. High-resolution bitmapped displays are created using single- or double-sized (8 x 16) characters in a dynamically built custom character set. Separate coloring of pairs of pixels can be obtained, freeing the display from the limitations of character size coloring.

This appendix will briefly demonstrate the use of custom character sets, multicolor mode, and screen bitmapping. Before we begin, you should be familiar with the discussions of the character sets that are presented at locations 32768 ($8000) and 36869 ($9005) of the memory map. You'll be using that information as you define a custom character set.

For a maze-type game program, you could use the standard character graphics to generate the actual maze design, and then use custom characters to represent the figures in the maze. For our example, we'll design characters for a takeoff on a well-known maze game; we'll need a chomping Commodore logo symbol, a few ghosts to chase it around, a cherry bonus symbol, and a power pill symbol.

Because we'll want to run the game on either an unexpanded or expanded VIC-20, the BASIC area, screen map, and character map will be positioned to work in either condition. Since the number of custom characters is small, the routine at 641–642 will be placed on the game's tape to unexpand any memory expansion. To that routine you could add the POKE of characters 131 and 13 into the keyboard buffer at 631–632, and 2 into location 198. This will cause the next tape file to be loaded and run, which will be the maze game main program. The program can then expect the memory configuration to be unexpanded, with the screen at 7680. When the game is stopped by the player's request, a SYS 64802 will be used to reexpand any expansion and reset the VIC-20.

The first thing the game program needs to do is to lower the upper limit of BASIC to protect the custom character set at locations 7168–7679. This location, with its 512 bytes, allows 64 custom characters, as well as the wraparound indexing into the normal character set reverse characters (although they will appear as unreversed) for scores, titles, and the like. POKE 55,0: POKE 56,28 :CLR will protect our custom character set map. POKE 36869,PEEK(36869) AND 240 OR 15 will set the address of the customer character set in the VIC chip register. FOR X=0 TO

8*64:POKE 7168+X,0:NEXT will clear out the 64-character area. The next 64 characters are actually part of the screen map area, and you can avoid them by not using the values from 64 to 127 as screen POKE codes. The four custom characters can be set into the character set with the lines:

```
100 FOR X=0 TO 4*8-1:READ C
110 POKE 7168+X,C:NEXT:GOTO 300
120 DATA 56,124,84,124,124,124,84,0
130 DATA 0,2,4,56,124,116,56,0
140 DATA 120,255,222,192,222,255,120,0
150 DATA 146,56,124,238,108,124,56,146
300 :
```

The rest of your program would begin at line 300.

Now you can POKE into the screen map the locations where you want your custom characters to appear: 0=ghost, 1=cherry, 2=logo, 3=power pill. The remaining 60 characters of the custom character set can be used for any additional characters you'd like. The POKE codes 64–127 should be avoided since they would obtain character pixel mapping information from the ever-active screen map area. The maze can be drawn by using screen POKE codes from 190 to 255. See the Code Chart in Appendix C, remembering to use characters above 63 and that they will *not* appear in reverse. The chart also shows that 129–154 are the screen POKE codes for the alphabet, and 176–185 can be used for keeping score.

More than one 8 x 8 character pixel map can be used if you want to create a shape larger than one character. For example, the following statements define a four-character shape of a three-dimensional cube that is constructed on the screen by POKEing the first and second characters side by side, with the third and fourth below them:

```
DATA 15,16,32,64,255,128,128,128
DATA 255,3,5,9,241,17,17,17
DATA 128,128,128,128,128,128,128,255
DATA 17,17,17,17,18,20,24,240
```

Multicolor mode may be used to color with 4 of 16 colors within a character. Location 37888 describes multicolor mode; locations 36879 and 36878 note the valid color codes. A tiny American flag could be created on the side of a ship with the line DATA 5,10,5,10,85,170,85,170 if the background color was blue, the border red, and you POKE 9 (white foreground and multicolor mode) into the color map location for the character.

You'll find a custom character editing program a great help when you design characters. Some character editors will generate the BASIC DATA statements that can be included in a program. There

are several good character editors available. One such editor
appeared in *COMPUTE!'s First Book of VIC.* By David Malmberg, it's
also included in the article "Custom Characters For the VIC."

Bitmapping

Bitmapping the screen requires a custom character set made of pixel
maps which describes every pixel on the screen to be mapped. This
isn't necessarily the whole screen, and in fact most bitmapping pro-
grams (including the Super Expander cartridge) settle for 20 columns
by 22 lines. An unexpanded VIC-20 has to limit the area to be
bitmapped.

The first thing you must decide is the number of lines and col-
umns that are to be bitmapped. A window on the screen must then
be created to fit that size. Every screen position showing requires
eight bytes to bitmap. If you choose the 22 line by 20 column for-
mat, for instance, you'll need 22*20*8=3520 bytes for a character set
to bitmap that area. A (22*20)/2=220 byte screen map is also
needed. The total amount of memory required for the screen and
character maps for a 20 x 22 display is thus 3740 bytes. It's obvious
that BASIC has to be placed in expansion memory.

You can divide the needed screen map area in half by using the
double-sized character feature. Simply specify it in location 36867,
bit 0. You would need to use that feature anyway since it's the only
way that unique index numbers can be placed in all the displayed
screen map bytes.

For an unexpanded VIC-20, a 12 x 16 (192 bytes) display area
leaves 1535 bytes of BASIC area free after the character set and
screen map are accounted for. This is done by placing BASIC at
6144, the screen at 5632, and 190 custom characters at 4096. Charac-
ters with screen POKE codes of 192–255 will overlap the screen
area, so don't use them. The following direct mode lines set the
BASIC and screen boundaries:

POKE 43,0: POKE 44,24
NEW
POKE 648,22: SYS 58648 (See these two locations in the memory
map.)

Within your program, the statement POKE 36869,PEEK(36869) AND
240 OR 12 sets the character set pointer to 4096. To initialize the
screen map for bitmapping, you could use:

FOR X=0 TO 191:POKE 5632+X,X:NEXT

The character map area will include garbage that can be cleaned
out with FOR X=0 TO 1519:POKE 4096+X,0:NEXT. Locations
36878 and 36879 can be set to the desired color combinations, and
36867 could be set to double-sized characters. The screen size of 13

by 14 is set by POKE 36866,13+128 (the 128 is part of the screen address) and POKE 36867,14*2. To center the window on the display, POKE 36864,13 POKE 36865,40.

The color map starts at location 38400 and can be set to the appropriate color code (multicolor bit included, if desired) when a character is placed on the screen. Remember that when double-sized characters are selected, a color nybble corresponds to a double-sized character. Alternatively, you may want to initialize the entire color map to a particular default color code, so that you don't need to bother with it for the bulk of your work.

All the setup is now done, and you're ready to set the custom character map to whatever strikes your imagination and see it instantly appear at the same offset on the screen as the number of the double-sized character. Location 4096 ($1000) has several formulas that can be used to calculate pixel locations from the desired line and column. Change the number 22 in those formulas to the width of your chosen display window.

A 40-column screen. Here are the definitions for a half-width set of letters and numbers. These definitions will allow you to create the illusion of twice as many characters per line so a 40-column display can be simulated. These definitions can be used with a large custom character set, but are best suited for a bitmapped screen. When using the definitions, store them someplace other than the custom character set, and move the needed 4 x 8 bit pixel map to the appropriate right or left side of a 8 x 8 sized character, or quarter of the double-sized character.

```
DATA 76,170,170,236,170,170,172,0: REM AB
DATA 76,170,138,138,138,170,76,0: REM CD
DATA 238,136,136,204,136,136,232,0: REM EF
DATA 106,138,138,174,170,170,106,0: REM GH
DATA 226,66,66,66,66,74,228,0: REM IJ
DATA 136,136,136,168,200,200,174,0: REM KL
DATA 160,236,234,170,170,170,170,0: REM MN
DATA 236,170,170,172,168,168,232,0: REM OP
DATA 72,172,170,170,168,232,104,0: REM QR
DATA 78,164,132,68,36,164,68,0: REM ST
DATA 170,170,170,170,170,164,228,0: REM UV
DATA 170,170,164,164,228,234,170,0: REM WX
DATA 174,160,226,68,72,64,78,0: REM YZ
DATA 206,74,66,68,72,72,238,0: REM 12
DATA 234,42,42,110,34,34,226,0: REM 34
DATA 238,136,136,206,42,42,206,0: REM 56
DATA 238,170,42,78,74,74,78,0: REM 78
DATA 228,170,170,234,42,42,36,0: REM 90
DATA 0,0,0,0,0,2,68,0: REM .,
DATA 64,160,32,64,64,0,64,0: REM ? SPACE
```

Appendix H

Alphabetical Cross Reference to the Location of Memory Map Labels

Label	Defined at
ABS	56408 ($DC58)
ACPTR	61209 ($EF19)
ADDPRC	1–2 ($1–2)
ADDSTR	54845 ($D63D)
ADRAY1	3–4 ($3–4)
ADRAY2	5–6 ($5–6)
ALCSPC	54516 ($D4F4)
ALC1	54389 ($D475)
ANDD	53225 ($CFE9)
ARG	105–110 ($69–6E)
ARGEXP	105–110 ($69–6E)
ARISGN	111 ($6F)
ARY	53713 ($D1D1)
ARYHED	53652 ($D194)
ARYTAB	47–48 ($2F–30)
ARY2	53837 ($D24D)
ARY6	53857 ($D261)
ARY14	53994 ($D2EA)
ASC	55179 ($D78B)
ASCFLT	56563 ($DCF3)
ASCII	215 ($D7)
ASC18	56702 ($DD7E)
ASRRES	55683 ($D983)
ATN	58123 ($E30B)
ATNCON	58171 ($E33B)
ATOF	56316 ($DBFC)
AUTODN	658 ($292)
A0CBM	64845 ($FD4D)
BACKUP	59624 ($E8E8)
BAD	256–318 ($100–13E)
BADSUB	53829 ($D245)
BASIC	49152–57343 ($C000–DFFF)
BASICSPILL	57344–58527 ($E000–E49F)
BASTACK	320–511 ($140–1FF)
BASVCTRS	58447 ($E44F)
BASZPT	255 ($FF)
BAUDOF	665–666 ($299–29A)
BAUDTBL	65372 ($FF5C)

BITCI	168 ($A8)
BITNUM	664 ($298)
BITS	104 ($68)
BITTS	180 ($B4)
BLKEND	64518 ($FC06)
BLNCT	205 ($CD)
BLNON	207 ($CF)
BLNSW	204 ($CC)
BLOAD	57701 ($E165)
BMSGS	49960 ($C328)
BREAK	65234 ($FED2)
BSAVE	57683 ($E153)
BSIV	64758 ($FCF6)
BSOUR	149 ($95)
BSTOP	51247 ($C82F)
BUF	512–600 ($200–258)
BUFPNT	166 ($A6)
BUMPTP	51451 ($C8FB)
BVECTORS	768–778 ($300–30A)
BVERIF	57698 ($E162)
CACPTR	65445 ($FFA5)
CASEL	34816–35839 ($8800–8BFF)
CASELRV	35840–36863 ($8C00–8FFF)
CASEU	32768–33791 ($8000–83FF)
CASEURV	33792–34815 ($8400–87FF)
CAS1	192 ($C0)
CBINV	790–791 ($316–317)
CBMBASIC	49156 ($C004)
CBMMSG	58409 ($E429)
CCALL	65511 ($FFE7)
CCHROUT	65490 ($FFD2)
CCIOUT	65448 ($FFA8)
CCLOS	65475 ($FFC3)
CCLRCHN	65484 ($FFCC)
CGETL	65508 ($FFE4)
CGIMAG	58247 ($E387)
CHANNL	19 ($13)
CHARAC	7 ($7)
CHARSET	60705 ($ED21)
CHKAUTO	64831 ($FD3F)
CHKIN	62151 ($F2C7)
CHKOUT	62217 ($F309)
CHR	55020 ($D6EC)
CHRERR	57870 ($E20E)
CHRGET	115–138 ($73–8A)
CHRGOT	121 ($79)

CHRIN	61966 ($F20E)
CHRINRS	62063 ($F26F)
CHRINSR	62052 ($F264)
CHRINTP	62000 ($F230)
CHRINTP2	62032 ($F250)
CHROUT	62074 ($F27A)
CHROUTTP	62096 ($F290)
CHRTST	53523 ($D113)
CINCH	65487 ($FFCF)
CINV	788–789 ($314–315)
CIOBASE	65523 ($FFF3)
CIOUT	61156 ($EEE4)
CLALL	62447 ($F3EF)
CLISTEN	65457 ($FFB1)
CLOAD	65493 ($FFD5)
CLOSE	62282 ($F34A)
CLR	50782 ($C65E)
CLRALINE	60045 ($EA8D)
CLRCHN	62451 ($F3F3)
CLSR	58719 ($E55F)
CMD	51846 ($CA86)
CMEMBOT	65436 ($FF9C)
CMEMTOP	65433 ($FF99)
CMPFAC	56411 ($DC5B)
CMPST	53294 ($D02E)
CMP0	176 ($B0)
CNTDN	165 ($A5)
CNVRTCD	59689 ($E929)
COLDBA	58232 ($E378)
COLDST	49152 ($C000)
COLECT	54790 ($D606)
COLOR	646 ($286)
COLORMAPS	37888–38399 ($9400–95FF)
COLORSET	59666 ($E912)
COLORSYN	60082 ($EAB2)
COLORTBL	59681 ($E921)
COMCHK	52989 ($CEFD)
COMFAC	55623 ($D947)
COMPAR	53270 ($D016)
CONT	51287 ($C857)
COPEN	65472 ($FFC0)
COS	57953 ($E261)
COUNT	11 ($B)
CPLOT	65520 ($FFF0)
CRDST	65463 ($FFB7)
CRDTIM	65502 ($FFDE)

CRESTOR	65418 ($FF8A)
CRNCH	50553 ($C579)
CRSW	208 ($D0)
CSAVE	65496 ($FFD8)
CSCNKEY	65439 ($FF9F)
CSCREEN	65517 ($FFED)
CSECOND	65427 ($FF93)
CSETLFS	65466 ($FFBA)
CSETMSG	65424 ($FF90)
CSETNAM	65469 ($FFBD)
CSETTIM	65499 ($FFDB)
CSETTMO	65442 ($FFA2)
CSTEL	63636 ($F894)
CSTE2	63671 ($F8B7)
CS10	63659 ($F8AB)
CTALK	65460 ($FFB4)
CTKSA	65430 ($FF96)
CTRLKEYS	60835 ($EDA3)
CUDTIM	65514 ($FFEA)
CUNLSN	65454 ($FFAE)
CUNTLK	65451 ($FFAB)
CURLIN	57–58 ($39–3A)
CVECTOR	65421 ($FF8D)
C3PO	148 ($94)
C5FFS	65413 ($FF85)
DATLIN	63–64 ($3F–40)
DATPTR	65–66 ($41–42)
DECBIN	51563 ($C96B)
DEF	54195 ($D3B3)
DEFPNT	78–79 ($4E–4D)
DELAY	652 ($28C)
DELST	54947 ($D6A3)
DELTSD	55003 ($D6DB)
DFLTN	153 ($99)
DFLTO	154 ($9A)
DIM	53377 ($D081)
DIMFLG	12 ($C)
DIVIDE	56082 ($DB12)
DIVTEN	56062 ($DAFE)
DPSW	156 ($9C)
DSCPNT	80–81 ($50–51)
EAL	174–175 ($AE–AF)
END	51249 ($C831)
ENDCHR	8 ($8)
ERROR	50231 ($C437)
ERRTAB	49566 ($C19E)

341

Appendix H

EVAL	52867 ($CE83)
EVALFN	54260 ($D3F4)
EVFN3	54351 ($D44F)
EVLVAR	53387 ($D08B)
EXP	57325 ($DFED)
EXPCON	57279 ($DFBF)
EXPONT	57211 ($DF7B)
EXTRA	52476 ($CCFC)
FA	186 ($BA)
FAC	97–102 ($61–66)
FACOV	112 ($70)
FACTFP	56272 ($DBD0)
FACTF1	56266 ($DBCA)
FACTF2	56263 ($DBC7)
FACT10	53005 ($CF0D)
FACT12	53032 ($CF28)
FACT17	53159 ($CFA7)
FAC2	105–110 ($69–6E)
FAH	63407 ($F7AF)
FAT	611–620 ($263–26C)
FBUFPT	113–114 ($71–72)
FCLOSE	57796 ($E1C4)
FILEMSG	63358 ($F77E)
FILENAME	63065 ($F659)
FILFAC	56553 ($DCE9)
FIND2	51462 ($C906)
FINLIN	50707 ($C613)
FINLMR	55137 ($D761)
FIRT	164 ($A4)
FLP05	57105 ($DF11)
FLTASC	56797 ($DDDD)
FLTCON	57110 ($DF16)
FN	54241 ($D3E1)
FNADR	187–188 ($BB–BC)
FNDFLNO	62415 ($F3CF)
FNDHDR	63591 ($F867)
FNDVAR	53479 ($D0E7)
FNLEN	183 ($B7)
FOPEN	57787 ($E1BB)
FOR	51010 ($C742)
FORPNT	73–74 ($49–4A)
FORWARD	59642 ($E8FA)
FOUR6	83 ($53)
FPCTEN	56057 ($DAF9)
FPC1	55740 ($D9BC)
FPC12	56755 ($DDB3)

FPC20	58077 ($E2DD)
FPINT	56475 ($DC9B)
FRE	54141 ($D37D)
FREKZP	251–254 ($FB–FE)
FREMSG	58372 ($E404)
FRESPC	53–54 ($35–36)
FRETOP	51–52 ($33–34)
FRMEVL	52638 ($CD9E)
FSBLK	190 ($BE)
FTOA	56335 ($DC0F)
FUNDSP	49234 ($C052)
GARBFL	15 ($F)
GCOL13	54717 ($D5BD)
GDBLN	206 ($CE)
GDCOL	647 ($287)
GET	52091 ($CB7B)
GETAD	55275 ($D7EB)
GETBYT	55195 ($D79B)
GETIN	61941 ($F1F5)
GETLIN	50528 ($C560)
GETQUE	58853 ($E5E5)
GETSCRN	58959 ($E64F)
GETSUB	53682 ($D1B2)
GET2RTN	58905 ($E619)
GONE	51172 ($C7E4)
GOSUB	51331 ($C883)
GOTO	51360 ($C8A0)
GRAPHMODE	60720 ($ED30)
GRBCOL	54566 ($D526)
GSINFO	55170 ($D782)
HIBASE	648 ($288)
HMSCON	57146 ($DF3A)
HOME	58753 ($E581)
IBASIN	804–805 ($324–325)
IBSOUT	806–807 ($326–327)
ICHKIN	798–799 ($31E–31F)
ICKOUT	800–801 ($320–321)
ICLALL	812–813 ($32C–32D)
ICLOSE	796–797 ($31C–31D)
ICLRCH	802–803 ($322–323)
ICRNCH	772–773 ($304–305)
IERROR	768–769 ($300–301)
IEVAL	778–779 ($30A–30B)
IF	51496 ($C928)
IFCHRG	57859 ($E203)
IGETIN	810–811 ($32A–32B)

IGONE	776–777 ($308–309)
IGRERR	52045 ($CB4D)
ILOAD	816–817 ($330–331)
ILQUAN	53832 ($D248)
IMAIN	770–771 ($302–303)
INBIT	167 ($A7)
INDEX	34–37 ($22–25)
INDX	200 ($C8)
INITBA	58276 ($E34A)
INITMEM	64909 ($FD8D)
INITSK	58648 ($E518)
INITVCTRS	58459 ($E45B)
INITVIA	65017 ($FDF9)
INITVIC	58819 ($E5C3)
INPCHN	65478 ($FFC6)
INPFLG	17 ($11)
INPPTR	65–67 ($43–44)
INPUT	52159 ($CBBF)
INPUTN	52133 ($CBA5)
INSRT	216 ($D8)
INT	56524 ($DCCC)
INTFLG	14 ($E)
INTFP	56380 ($DC3C)
INTFP1	56388 ($DC44)
INTIDX	53674 ($D1AA)
IOBASE	58624 ($E500)
IOPEN	794–795 ($31A–31B)
IQPLOP	774–775 ($306–307)
IRQ	60095 ($EABF)
IRQROUT	65394 ($FF72)
IRQTMP	671–672 ($29F–2A0)
IRQVCTRS	65009 ($FDF1)
ISAVE	818–819 ($332–333)
ISCNTC	65505 ($FFE1)
ISTOP	808–809 ($328–329)
JMPER	84–86 ($54–56)
JTP20	63626 ($F88A)
KERNAL	58528–65535 ($E4A0–FFFF)
KEYD	631–640 ($277–280) ✳
KEYLOG	655–656 ($28F–290)
KEYTAB	245–246 ($F5–F6)
KEYVCTRS	60486 ($EC46)
KMSGSHOW	61926 ($F1E6)
KMSGTBL	61812 ($F174)
KOUNT	651 ($28B)
KVECTORS	788–819 ($314–$333)

LA	184 ($B8)
LADIV	56079 ($DB0F)
LAMIN	55376 ($D850)
LAPLUS	55399 ($D867)
LASTPT	23–24 ($17–18)
LAT	601–610 ($259–262)
LDAD1	63572 ($F854)
LDTB1	217–241 ($D9–F1)
LDTB2	60925 ($EDFD)
LDTND	152 ($98)
LDVRMSG	63082 ($F66A)
LEFT	55040 ($D700)
LEN	55164 ($D77C)
LET	51621 ($C9A5)
LET2	51650 ($C9C2)
LET5	51674 ($C9DA)
LET9	51756 ($CA2C)
LINNUM	20–21 ($14–15)
LINPTR	60030 ($EA7E)
LIST	50844 ($C69C)
LISTEN	60951 ($EE17)
LIST1	60956 ($EE1C)
LNKPRG	50483 ($C533)
LNMX	213 ($D5)
LOAD	62786 ($F542)
LOADSER	62812 ($F55C)
LODARG	55948 ($DA8C)
LODFAC	56226 ($DBA2)
LOG	55786 ($D9EA)
LOGCON	55745 ($D9C1)
LOGOKEYS	60640 ($ECE0)
LPACHK	52986 ($CEFA)
LP2	58831 ($E5CF)
LSTSHF	654 ($28E)
LSTX	197 ($C5)
LXSP	201–202 ($C9–CA)
MAIN	50304 ($C480)
MAKADR	55287 ($D7F7)
MAKFP	54161 ($D391)
MAKINT	53695 ($D1BF)
MAKSPC	50104 ($C3B8)
MAKSTR	54407 ($D487)
MAKVAR	53533 ($D11D)
MAXINT	53669 ($D1A5)
MEMBOT	65154 ($FE82)
MEMERR	50229 ($C435)

Appendix H

346

PAROC	57878 ($E216)
PARSL	57809 ($E1D1)
PATCHBAS	57590 ($E0F6)
PATCHER	58486 ($E476)
PATCHES	58556 ($E4BC)
PCNTR	163 ($A3)
PEEK	55309 ($D80D)
PG3FREE	784–787 ($310–313)
PIVAL	52904 ($CEA8)
PLOT	58634 ($E50A)
PLUS	55402 ($D86A)
PLUS1	55394 ($D862)
PLUS6	55463 ($D8A7)
PNT	209–210 ($D1–D2)
PNTR	211 ($D3)
POKE	55332 ($D824)
POS	54174 ($D39E)
PRDY	50281 ($C469)
PRINT	51872 ($CAA0)
PRINTN	51840 ($CA80)
PRTFIX	56781 ($DDCD)
PRTIN	56770 ($DDC2)
PRTOS	52027 ($CB3B)
PRTSTR	51998 ($CB1E)
PRTY	155 ($9B)
PRT1	51866 ($CA9A)
PRT6	51944 ($CAE8)
PRT7	51960 ($CAF8)
PTR1	158 ($9E)
PTR2	159 ($9F)
PUTSCRN	60074 ($EAAA)
QPLOP	50970 ($C71A)
QTSW	212 ($D4)
QUOTECK	59064 ($E6B8)
RAMBLK0	1024–4095 ($400–FFF)
RAMBLK1	8192–16383 ($2000–3FFF)
RAMBLK2	16384–24575 ($4000–5FFF)
RAMBLK3	24576–32767 ($6000–7FFF)
RAMBLK4	40960–49151 ($A000–BFFF)
RAMSPC	50184 ($C408)
RBLK	63689 ($F8C9)
RDTIM	63328 ($F760)
RDTPBLKS	62680 ($F8C0)
RD300	64466 ($FBD2)
READ	52230 ($CC06)
READIOST	65128 ($FE68)

Appendix H

READST	65111 ($FE57)
READT	63886 ($F98E)
READY	50292 ($C474)
REM	51515 ($C93B)
RESHO	38–42 ($26–2A)
RESLST	49310 ($C09E)
RESTOR	64850 ($FD52)
RESTORE	51229 ($C81D)
RETREAT	59181 ($E72D)
RETURN	51410 ($C8D2)
RETVP	53637 ($D185)
RFTOA	56332 ($DC0C)
RIBUF	247–248 ($F7–F8)
RIDATA	170 ($AA)
RIDBE	667 ($29B)
RIDBS	668 ($29C)
RIGHT	55084 ($D72C)
RINONE	169 ($A9)
RIPRTY	171 ($AB)
RND	57492 ($E094)
RNDC1	57482 ($E08A)
RNDX	139–143 ($8B–8F) ✗
ROBUF	249–250 ($F9–FA)
RODATA	182 ($B6)
RODBE	670 ($29E)
RODBS	669 ($29D)
ROPRTY	189 ($BD)
ROUND	56347 ($DC1B)
RPACHK	52983 ($CEF7)
RPTFLG	650 ($28A) ✗
RSBREAK	61605 ($F0A5)
RSCPTBIT	61479 ($F027)
RSDVCERR	61625 ($F0B9)
RSFRAMER	61608 ($F0A8)
RSINBIT	61494 ($F036)
RSINBYTE	61551 ($F06F)
RSINERR	61610 ($F0AA)
RSINPRTY	61579 ($F08B)
RSMISSNG	61462 ($F016)
RSNMI	65246 ($FEDE)
RSNXTBIT	61347 ($EFA3)
RSNXTBYT	61422 ($EFEE)
RSNXTIN	61775 ($F14F)
RSOPNIN	61718 ($F116)
RSOPNOUT	61628 ($F0BC)
RSOUTSAV	61677 ($F0ED)

RSOVERUN	61602 ($F0A2)
RSPAUSE	61792 ($F160)
RSPREPIN	61531 ($F05B)
RSPREPOT	61698 ($F102)
RSPRTY	61375 ($EFBF)
RSPRTYER	61597 ($F09D)
RSSTAT	663 ($297)
RSSTOPS	61416 ($EFE8)
RSSTPBIT	61515 ($F04B)
RSSTRBIT	61544 ($F068)
RTI	65366 ($FF56)
RTRN	59608 ($E8D8)
RUN	51313 ($C871)
RUNTB	60916 ($EDF4)
RVS	199 ($C7)
SA	185 ($B9)
SAL	172–173 ($AC–AD)
SAREG	780 ($30C)
SAT	621–630 ($26D–276)
SAVE	63093 ($F675)
SAVESER	63122 ($F692)
SAVETP	63217 ($F6F1)
SAVING	63272 ($F728)
SAVREGS	780–783 ($30C–30F)
SCATN	61125 ($EEC5)
SCNKEY	60190 ($EB1E)
SCNSTK	50058 ($C38A)
SCREEN	7680–8191 ($1E00–1FFF)
SCREENX	4096–4607 ($1000–11FF)
SCRL	59765 ($E975)
SCRN	58629 ($E505)
SCRNOUT	59202 ($E742)
SCROLL	59114 ($E6EA)
SECOND	61120 ($EEC0)
SEREVL	57408 ($E040)
SERGET	58546 ($E4B2)
SERNAME	62613 ($F495)
SEROUT0	58537 ($E4A9)
SEROUT1	58528 ($E4A0)
SER2	57430 ($E056)
SETADDR	60014 ($EA6E)
SETCHAR	59077 ($E6C5)
SETFLCH	62431 ($F3DF)
SETIODEF	58811 ($E5BB)
SETKEYS	60380 ($EBDC)
SETLFS	65104 ($FE50)

SETMSG	65126 ($FE66)
SETNAM	65097 ($FE49)
SETSLINK	58759 ($E587)
SETTIM	63335 ($F767)
SETTMO	65135 ($FE6F)
SFDX	203 ($CB)
SGN	56377 ($DC39)
SGNFAC	56363 ($DC2B)
SGNFLG	103 ($67)
SHFLAG	653 ($28D)
SHFTKEYS	60575 ($EC9F)
SIN	57960 ($E268)
SIZE	82 ($52)
SKIPST	51448 ($C8F8)
SKPCOM	57867 ($E20B)
SPMSG	61922 ($F1E2)
SPREG	783 ($30F)
SQR	57201 ($DF71)
SRBAD	61108 ($EEB4)
SRCHING	63047 ($F647)
SRCLKHI	61316 ($EF84)
SRCLKLO	61325 ($EF8D)
SRSEND	61001 ($EE49)
STACK	256–511 ($100–1FF)
STAL	193–194 ($C1–C2) ✳
START	64802 ($FD22)
STATUS	144 ($90)
STKEY	145 ($91)
STKSPC	50171 ($C3FB)
STMDSP	49164 ($C00C)
STOP	63344 ($F770)
STORFAC	56276 ($DBD4)
STR	54373 ($D465)
STREND	49–50 ($31–32)
STT1	63837 ($F95D)
STXTPT	50830 ($C68E)
SUB	55379 ($D853)
SUBFLG	16 ($10)
SVXT	146 ($92)
SXREG	781 ($30D)
SYNCHR	52991 ($CEFF)
SYNERR	53000 ($CF08)
SYNO	150 ($96)
SYNPRT	60065 ($EAA1)
SYREG	782 ($30E)
SYSTEM	57639 ($E127)

TALK	60948 ($EE14)
TAN	58033 ($E2B1)
TANSGN	18 ($12)
TAPE	63732 ($F8F4)
TAPEH	63463 ($F7E7)
TAPE1	178–179 ($B2–B3)
TBLX	214 ($D6)
TBUFFR	828–1019 ($33C–3FB)
TEMPF3	87–96 ($57–60)
TEMPPT	22 ($16)
TEMPST	25–33 ($19–21)
TEMP1	177 ($B1)
TIME	160–162 ($A0–A2)
TIMES	55848 ($DA28)
TIMES3	55897 ($DA59)
TIMOUT	645 ($285)
TKSA	61134 ($EECE)
TNIF	64719 ($FCCF)
TNOFF	64776 ($FD08)
TPBLOCK	829–1019 ($33D–3FB)
TPBUFA	63565 ($F84D)
TPHBGN	829–830 ($33D–33E)
TPHDRID	828 ($33C)
TPHEND	831–832 ($33F–340)
TPHFREE	1020–1023 ($3FC–3FF)
TPHNAME	833–1019 ($341–3FD)
TPSTORE	64173 ($FAAD)
TPTOGLE	64490 ($FBEA)
TRMPOS	9 ($9)
TSTMEM	65169 ($FE91)
TSTOP	63819 ($F94B)
TSTSTOP	51244 ($C82C)
TXTPTR	122–123 ($7A–7B)
TXTTAB	43–44 ($2B–2C)
TYPCHK	52600 ($CD8A)
UDTIM	63284 ($F734)
UNDEF	54190 ($D3AE)
UNLSN	61188 ($EF04)
UNTLK	61174 ($EEF6)
UNUSDNMI	58805 ($E5B5)
USER	243–244 ($F3–F4)
USRCMD	814–815 ($32E–32F)
USRCMDS	820–827 ($334–33B)
USRPGM0K	4096–7679 ($1000–1DFF)
USRPGM3K	4096–8191 ($1000–1FFF)
USRPGM8K	4608–8191 ($1200–1FFF)

Appendix H

USRPOK	0 ($0)
USRVCTRS	673–767 ($2A1–2FF)
VAL	55213 ($D7AD)
VALTYP	13 ($D)
VARNAM	69–70 ($45–46)
VARPNT	71–72 ($47–48)
VARRANGE	53012 ($CF14)
VARTAB	45–46 ($2D–2E)
VCTRIRQ	65534 ($FFFE)
VCTRNMI	65530 ($FFFA)
VCTRRST	65532 ($FFFC)
VECTOR	64855 ($FD57)
VECTORS	64877 ($FD6D)
VERCHK	10 ($A)
VERCK	147 ($93)
VIA1ACR	37147 ($911B)
VIA1DDRA	37139 ($9113)
VIA1DDRB	37138 ($9112)
VIA1IER	37150 ($911E)
VIA1IFR	37149 ($911D)
VIA1PA1	37137 ($9111)
VIA1PA2	37151 ($911F)
VIA1PB	37136 ($9110)
VIA1PCR	37148 ($911C)
VIA1SR	37146 ($911A)
VIA1T1CH	37141 ($9115)
VIA1T1CL	37140 ($9114)
VIA1T1LH	37143 ($9117)
VIA1T1LL	37142 ($9116)
VIA1T2CH	37145 ($9119)
VIA1T2CL	37144 ($9118)
VIA2ACR	37163 ($912B)
VIA2DDRA	37155 ($9123)
VIA2DDRB	37154 ($9122)
VIA2IER	37166 ($912E)
VIA2IFR	37165 ($912D)
VIA2PA1	37153 ($9121)
VIA2PA2	37167 ($912F)
VIA2PB	37152 ($9120)
VIA2PCR	37164 ($912C)
VIA2SR	37162 ($912A)
VIA2T1CH	37157 ($9125)
VIA2T1CL	37156 ($9124)
VIA2T1HL	37159 ($9127)
VIA2T1LL	37158 ($9126)
VIA2T2CH	37161 ($9129)

VIA2T2CL	37160 ($9128)
VICCRA	36874 ($900A)
VICCRB	36875 ($900B)
VICCRC	36876 ($900C)
VICCRD	36877 ($900D)
VICCRE	36878 ($900E)
VICCRF	36879 ($900F)
VICCR0	36864 ($9000)
VICCR1	36865 ($9001)
VICCR2	36866 ($9002)
VICCR3	36867 ($9003)
VICCR4	36868 ($9004)
VICCR5	36869 ($9005)
VICCR6	36870 ($9006)
VICCR7	36871 ($9007)
VICCR8	36872 ($9008)
VICCR9	36873 ($9009)
VICINIT	60900 ($EDE4)
VPRTY	64785 ($FD11)
WAIT	55341 ($D82D)
WAITABIT	61334 ($EF96)
WARMBAS	58471 ($E467)
WARMST	49154 ($C002)
WBLK	63715 ($F8E3)
WHATKEYS	60777 ($ED69)
WRAPLINE	60763 ($ED5B)
WRITE	64523 ($FC0B)
WRTN1	64661 ($FC95)
WRTZ	64680 ($FCA8)
WRT62	64795 ($FD1B)
XFERSTR	54906 ($D67A)
XMAX	649 ($289)
XSAV	151 ($97)
ZERFAC	55543 ($D8F7)

Appendix I

A Beginner's Guide to Typing In Programs

What Is a Program?
A computer cannot perform any task by itself. Like a car without gas, a computer has *potential*, but without a program, it isn't going anywhere. Most of the programs published in this book are written in a computer language called BASIC. BASIC is easy to learn and is built into all VIC-20s.

BASIC Programs
Computers can be picky. Unlike the English language, which is full of ambiguities, BASIC usually has only one right way of stating something. Every letter, character, or number is significant. A common mistake is substituting a letter such as O for the numeral 0, a lowercase l for the numeral 1, or an uppercase B for the numeral 8. Also, you must enter all punctuation such as colons and commas just as they appear in the book. Spacing can be important. To be safe, type in the listings *exactly* as they appear.

Braces and Special Characters
The exception to this typing rule is when you see the 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 J, "How to Type In Programs."

About DATA Statements
Some programs contain a section or sections of DATA statements. These lines provide information needed by the program. Some DATA statements contain actual programs (called machine language); others contain graphics codes. These lines are especially sensitive to errors.

   If a single number in any one DATA statement is mistyped, your machine could lock up, or crash. The keyboard and STOP key may seem dead, and the screen may go blank. Don't panic—no damage is 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 LOAD 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

354

program line that READs the data. *The error is still in the DATA statements, though.*

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 you at least need to know how to backspace. Do you know how to enter reverse video, lowercase, and control characters? It's all explained in your VIC's manual, *Personal Computing on the VIC.*

A Quick Review

1. Type in the program a line at a time, in order. Press RETURN at the end of each line. Use the INST/DEL key to erase 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 as the appropriate control key (see Appendix J, "How to Type in Programs").

Appendix J

How to Type In Programs

Many of the programs in this book contain special control characters (cursor control, color keys, reverse characters, and so on). To make it easy to know exactly what to type when entering one of these programs into your computer, we have established the following listing conventions.

Generally, VIC-20 program listings will contain words within braces which spell out any special characters: {DOWN} would mean to press the cursor down key. {5 SPACES} would mean 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 the SHIFT key. This would appear on your screen as a heart symbol. If you find an underlined key enclosed in braces (e.g., {10 N}), you should type the key as many times as indicated (in our example, you would enter ten shifted N's).

If a key is enclosed in special brackets, [< >], you should hold down the *Commodore key* while pressing the key inside the special brackets. (The Commodore key is the key in the lower left corner of the keyboard.) Again, if the key is preceded by a number, you should press the key as many times as necessary.

Rarely, you'll see a solitary letter of the alphabet enclosed in braces, such as {A}. You should never have to enter such a character on the VIC-20, but if you do, you would have to leave the quote mode (press RETURN and cursor back up to the position where the control character should go), press CTRL-9 (RVS ON), the letter in braces, and then CTRL-0 (RVS OFF).

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. If you type something and then try to change it by moving the cursor left, you'll only get a bunch of reverse-video lines. These are the symbols for cursor left. The only editing key that isn't programmable is the DEL key; you can still use DEL to back up and edit the line. Once you type another quote, you are out of quote mode.

You also go into quote mode when you INSerT spaces into a

line. In any case, the easiest way to get out of quote mode is to just press RETURN. You'll then be out of quote mode and you can cursor up to the mistyped line and fix it.

Use the following table when entering cursor and color control keys:

When You Read:	Press:	See:	When You Read:	Press:	See:
{CLR}	SHIFT CLR/HOME	▨	{GRN}	CTRL 6	▨
{HOME}	CLR/HOME	▨	{BLU}	CTRL 7	▨
{UP}	SHIFT ↑CRSR↓	▨	{YEL}	CTRL 8	▨
{DOWN}	↑CRSR↓	▨	{F1}	f1	▨
{LEFT}	SHIFT ←CRSR→	▨	{F2}	SHIFT f1	▨
{RIGHT}	←CRSR→	▨	{F3}	f3	▨
{RVS}	CTRL 9	▨	{F4}	SHIFT f3	▨
{OFF}	CTRL 0	▨	{F5}	f5	▨
{BLK}	CTRL 1	▨	{F6}	SHIFT f5	▨
{WHT}	CTRL 2	▨	{F7}	f7	▨
{RED}	CTRL 3	▨	{F8}	SHIFT f7	▨
{CYN}	CTRL 4	▨	←	←	▨
{PUR}	CTRL 5	▨	↑	SHIFT ↑	▨

Screen Location Table

Row

0	7680 (4096)
	7702 (4118)
	7724 (4140)
	7746 (4162)
	7768 (4184)
5	7790 (4206)
	7812 (4228)
	7834 (4250)
	7856 (4272)
	7878 (4294)
10	7900 (4316)
	7922 (4338)
	7944 (4360)
	7966 (4382)
	7988 (4404)
15	8010 (4426)
	8032 (4448)
	8054 (4470)
	8076 (4492)
	8098 (4514)
20	8120 (4536)
	8142 (4558)
22	8164 (4580)

0   5   10   15   20

Column

Note: Numbers in parentheses are for VICs with 8K or more of memory expansion.

Screen Color Memory Table

Row

Row	Address	
0	38400 (37888)	
	38422 (37910)	
	38444 (37932)	
	38466 (37954)	
	38488 (37976)	
5	38510 (37998)	
	38532 (38020)	
	38554 (38042)	
	38576 (38064)	
	38598 (38086)	
10	38620 (38108)	
	38642 (38130)	
	38664 (38152)	
	38686 (38174)	
	38708 (38196)	
15	38730 (38218)	
	38752 (38240)	
	38774 (38262)	
	38796 (38284)	
	38818 (38306)	
20	38840 (38328)	
	38862 (38350)	
22	38884 (38372)	

Column: 0   5   10   15   20

Column

Note: Numbers in parentheses are for VICs with 8K or more of memory expansion.

Appendix M

ASCII Codes

ASCII	CHARACTER	ASCII	CHARACTER
5	WHITE	50	2
8	DISABLE	51	3
	SHIFT COMMODORE	52	4
9	ENABLE	53	5
	SHIFT COMMODORE	54	6
13	RETURN	55	7
14	LOWERCASE	56	8
17	CURSOR DOWN	57	9
18	REVERSE VIDEO ON	58	:
19	HOME	59	;
20	DELETE	60	<
28	RED	61	=
29	CURSOR RIGHT	62	>
30	GREEN	63	?
31	BLUE	64	@
32	SPACE	65	A
33	!	66	B
34	"	67	C
35	#	68	D
36	$	69	E
37	%	70	F
38	&	71	G
39	'	72	H
40	(	73	I
41	)	74	J
42	*	75	K
43	+	76	L
44	,	77	M
45	–	78	N
46	.	79	O
47	/	80	P
48	0	81	Q
49	1	82	R

ASCII	CHARACTER	ASCII	CHARACTER
83	S	120	♣
84	T	121	
85	U	122	♦
86	V	123	
87	W	124	
88	X	125	
89	Y	126	π
90	Z	127	
91	[	133	f1
92	£	134	f3
93	]	135	f5
94	↑	136	f7
95	←	137	f2
96		138	f4
97		139	f6
98		140	f8
99		141	SHIFTED RETURN
100		142	UPPERCASE
101		144	BLACK
102		145	CURSOR UP
103		146	REVERSE VIDEO OFF
104		147	CLEAR SCREEN
105		148	INSERT
106		156	PURPLE
107		157	CURSOR LEFT
108		158	YELLOW
109		159	CYAN
110		160	SPACE
111		161	
112		162	
113		163	
114		164	
115	♥	165	
116		166	
117		167	
118	⊠	168	
119	○	169	

Appendix M

ASCII	CHARACTER	ASCII	CHARACTER
170		207	
171		208	
172		209	
173		210	
174		211	
175		212	
176		213	
177		214	
178		215	
179		216	
180		217	
181		218	
182		219	
183		220	
184		221	
185		222	
186		223	
187		224	SPACE
188		225	
189		226	
190		227	
191		228	
192		229	
193		230	
194		231	
195		232	
196		233	
197		234	
198		235	
199		236	
200		237	
201		238	
202		239	
203		240	
204		241	
205		242	
206		243	

ASCII	CHARACTER
244	□
245	▯
246	▮
247	□
248	◪
249	◪
250	□
251	◪
252	◼
253	◩
254	◪
255	π

0–4, 6–7, 10–12, 15–16, 21–27, 128–132, 143, and 149–155 have no effect. 192–223 same as 96–127;224–254 same as 160–190; 255 same as 126.

Appendix N

Screen Codes

POKE	Uppercase and Full Graphics Set	Lower- and Uppercase	POKE	Uppercase and Full Graphics Set	Lower- and Uppercase
0	@	@	31	←	←
1	A	a	32	-space-	
2	B	b	33	!	!
3	C	c	34	"	"
4	D	d	35	#	#
5	E	e	36	$	$
6	F	f	37	%	%
7	G	g	38	&	&
8	H	h	39	'	'
9	I	i	40	(	(
10	J	j	41	)	)
11	K	k	42	*	*
12	L	l	43	+	+
13	M	m	44	,	,
14	N	n	45	-	-
15	O	o	46	.	.
16	P	p	47	/	/
17	Q	q	48	0	0
18	R	r	49	1	1
19	S	s	50	2	2
20	T	t	51	3	3
21	U	u	52	4	4
22	V	v	53	5	5
23	W	w	54	6	6
24	X	x	55	7	7
25	Y	y	56	8	8
26	Z	z	57	9	9
27	[	[	58	:	:
28	£	£	59	;	;
29	]	]	60	<	<
30	↑	↑	61	=	=

POKE	Uppercase and Full Graphics Set	Lower- and Uppercase	POKE	Uppercase and Full Graphics Set	Lower- and Uppercase
62	>	>	95	◧	◧
63	?	?	96	- -space- -	
64	⊟	⊟	97	▯	▯
65	♠	A	98	▬	▬
66	◻	B	99	☐	☐
67	⊟	C	100	☐	☐
68	☐	D	101	☐	☐
69	☐	E	102	▨	▨
70	☐	F	103	☐	☐
71	◻	G	104	▟	▟
72	◻	H	105	◣	◪
73	◖	I	106	◗	◗
74	◗	J	107	⊞	⊞
75	◿	K	108	◤	◤
76	☐	L	109	◳	◳
77	◺	M	110	◹	◹
78	◿	N	111	☐	☐
79	☐	O	112	◱	◱
80	☐	P	113	⊞	⊞
81	●	Q	114	⊞	⊞
82	☐	R	115	⊟	⊟
83	♥	S	116	▮	▮
84	◻	T	117	▮	▮
85	◿	U	118	◗	◗
86	⊠	V	119	☐	☐
87	◯	W	120	☐	☐
88	♣	X	121	☐	☐
89	◻	Y	122	☐	✔
90	◆	Z	123	◰	◰
91	⊞	⊞	124	◳	◳
92	▨	▨	125	◱	◱
93	◻	◻	126	◣	◣
94	π	▨	127	◳	◳

128-255 are reverse video of 0-127.

Index of Subjects by Topic

Location	Description	Subject
+		
49280	($C080)	Math operation dispatch vector table, in token order
54566	($D526)	Garbage collection
54845	($D63D)	BASIC +, concatenate string
55402	($D86A)	BASIC + (add)
−		
49280	($C080)	Math operation dispatch vector table, in token order
55379	($D853)	BASIC − (subtract)
/		
49280	($C080)	Math operation dispatch vector table, in token order
56082	($DB12)	BASIC / divide FAC2 by FAC resulting in FAC
∗		
49280	($C080)	Math operation dispatch vector table, in token order
55848	($DA28)	BASIC ∗ multiply FAC2 by FAC, leaving the result in FAC
<		
49280	($C080)	Math operation dispatch vector table, in token order
53270	($D016)	Compare numerics or strings; also used for BASIC <, =, >
>		
49280	($C080)	Math operation dispatch vector table, in token order
53005	($CF0D)	Set up index for monadic minus
53270	($D016)	Compare numerics or strings; also used for BASIC <, =, >
=		
49280	($C080)	Math operation dispatch vector table, in token order
53270	($D016)	Compare numerics or strings; also used for BASIC <, =, >
⇑		
49280	($C080)	Math operation dispatch vector table, in token order
57211	($DF7B)	BASIC ⇑ (power)
A0CBM		
40960–49151	($A000–BFFF)	8K RAM expansion block 4
64831	($FD3F)	Check for an autostarting program at 40960 ($A000)
64845	($FD4D)	A0CBM characters with the high order bit on in the last three
ABS		
49234	($C052)	Function dispatch vector table, in token order
56408	($DC58)	BASIC ABS

Accumulator—*see* FAC and FAC2

Alternate screens—*see* Multiple screen

AND

7	($7)	Search-character for scanning BASIC statements
11	($B)	BASIC buffer index/array dimensions
49280	($C080)	Math operation dispatch vector table, in token order
53225	($CFE9)	BASIC AND

Array

11	($B)	BASIC buffer index/array dimensions
12	($C)	Flags for locate-or-build-array routines
38–42	($26–2A)	BASIC multiplication work area
45–46	($2D–2E)	Pointer to the end of BASIC program, start of variables
47–48	($2F–30)	Pointer to the end of BASIC variables, start of arrays
49–50	($31–32)	Pointer to the end of BASIC arrays, start of free area
113–114	($71–72)	Series evaluation pointer
50782	($C65E)	BASIC CLR
51621	($C9A5)	BASIC LET
52638	($CD9E)	Formula/expression evaluation
53377	($C081)	BASIC DIM
53387	($D08B)	Locate or create variable
53479	($D0E7)	Locate the variable
53533	($D11D)	Create new variable
53652	($D194)	Calculate the length of an array descriptor
53713	($D1D1)	Find an array item or create an array
53837	($D24D)	Found the array, check the subscript range
53857	($D261)	Create an array
53994	($D2EA)	Locate a particular array element
54092	($D34C)	Compute multidimension array size
54141	($D37D)	BASIC FRE
54566	($D526)	Garbage collection
Appendix B		Format of variables and floating point accumulators

ASC

49234	($C052)	Function dispatch vector table, in token order
55170	($D782)	Get string information
55179	($D78B)	BASIC ASC

ASCII

56563	($DCF3)	Converts an ASCII string to a floating point number in FAC
56797	($DDDD)	Convert FAC to TI$ or an ASCII string
60510	($EC5E)	Table used for decoding unshifted keys into ASCII
60575	($EC9F)	Table used for decoding SHIFTed keys into ASCII
60640	($ECE0)	Table used for decoding Commodore SHIFTed keys into ASCII
60835	($EDA3)	Table used for decoding CTRL SHIFTed keys into ASCII

ATN

49234	($C052)	Function dispatch vector table, in token order
58123	($E30B)	BASIC ATN
58171	($E33B)	Table of constant values for ATN evaluation

Auxiliary color—*see* Color

Auxiliary register

37146	($911A)	Shift register for parallel/serial conversion
37147	($911B)	Auxiliary control register
37159	($9127)	Timer 1 high order (MSB) latch byte
37162	($912A)	Shift register for parallel/serial conversion
37163	($912B)	Auxiliary control register
65017	($FDF9)	Initialize the 6522 VIA registers

Background—*see* Color

Bitmap

4096–4607	($1000–11FF)	Screen map RAM on VIC-20 with 8K or more expansion
32768–36863	($8000–8FFF)	Character maps
36867	($9003)	Number of character lines displayed, part of raster location
36869	($9005)	Screen map and character map addresses
37888–38399	($9400–95FF)	Screen color map (8K+ expanded VIC-20)

Border—*see* Color

BREAK

57–58	($39–3A)	Line number of the BASIC statement being executed
170	($AA)	Tape: input status flags, sync countdown/RS-232 byte assembly
663	($297)	RS-232 status register
790–791	($316–317)	Vector to the BREAK interrupt routine at 65234 ($FED2)
814–815	($32E–32F)	User vector can be placed here, held over from PET ML monitor
49154	($C002)	Vector to the routine to do the warm start of BASIC 58471 ($E467)
49566	($C19E)	Table of BASIC error messages
50020	($C364)	Miscellaneous messages
50281	($C469)	Display ERROR or another message pointed to
51249	($C831)	BASIC END
57590	($E0F6)	BASIC patch routines
58471	($E467)	Perform a warm start of BASIC
60095	($EABF)	IRQ handler
61605	($F0A5)	RS-232: break detected on input
64850	($FD52)	Cause the RAM system vectors to be reset to provided defaults
65017	($FDF9)	Initialize the 6522 VIA registers
65193	($FEA9)	NMI handler routine
65234	($FED2)	BREAK interrupt entry
65394	($FF72)	IRQ routine initial 6502 entry point

Cartridges

24576–32767	($6000–7FFF)	8K RAM expansion block 3
40960–49151	($A000–BFFF)	8K RAM expansion block 4
49152	($C000)	Vector to the routine for the cold start of BASIC 58232 ($E378)
64802	($FD22)	Power-on/reset routine (checks for autostart cartridge)
64831	($FD3F)	Check for an autostarting program at 40960 ($A000)
65193	($FEA9)	NMI handler routine
65532	($FFFC)	6502 vector to 64802 ($FD22)

369

CA1 and CA2

146	($92)	Tape: 0/1 bit timebase fluctuation during read operations
181	($B5)	Tape: flag for currently reading data or leader
37136–37151	($9110–911F)	6522 VIA chip 1
37137	($9111)	Port A I/O register
37139	($9113)	Eight bits, each of which corresponds to the same-numbered bit
37147	($911B)	Auxiliary control register
37148	($911C)	Peripheral control register for handshaking
37149	($911D)	Interrupt flag register
37150	($911E)	Interrupt enable register
37151	($911F)	Mirror of port A I/O register at 37137 ($9111)
37152–37167	($9120–912F)	6522 VIA chip 2
37153	($9121)	Port A I/O register
37155	($9123)	Eight bits, each of which corresponds to the same-numbered bit
37163	($912B)	Auxiliary control register
37164	($912C)	Peripheral control register for handshaking
37165	($912D)	Interrupt flag register
37166	($912E)	Interrupt enable register
37167	($912F)	Mirror of port A I/O register at 37153 ($9121)
51756	($CA2C)	LET: assign string variable
61316	($EF84)	Serial: set clock line high
61325	($EF8D)	Serial: set clock line low
63732	($F8F4)	Tape: common tape read/write; start tape operations
63886	($F98E)	Tape: read tape data bits into location 191 ($BF) (IRQ driven)
65017	($FDF9)	Initialize the 6522 VIA registers

CB1 and CB2

181	($B5)	Tape: flag for currently reading data or leader
37136–37151	($9110–911F)	6522 VIA chip 1
37136	($9110)	Port B I/O register
37137	($9111)	Port A I/O register
37138	($9112)	Data direction register for port B
37146	($911A)	Shift register for parallel/serial conversion
37147	($911B)	Auxiliary control register
37148	($911C)	Peripheral control register for handshaking
37149	($911D)	Interrupt flag register
37150	($911E)	Interrupt enable register
37152–37167	($9120–912F)	6522 VIA chip 2
37152	($9120)	Port B I/O register
37162	($912A)	Shift register for parallel/serial conversion
37163	($912B)	Auxiliary control register
37164	($912C)	Peripheral control register for handshaking
37165	($912D)	Interrupt flag register
37166	($912E)	Interrupt enable register
50231	($C437)	BASIC error message routine
50292	($C474)	Display READY. message
51998	($CB1E)	Part of PRINT: print a string ended by a carriage return
52159	($CBBF)	BASIC INPUT
56781	($DDCD)	Decimal number display routine
58528	($E4A0)	Serial: output a 1 on the serial data line
58537	($E4A9)	Serial: output a 0 on the serial data line

61347	($EFA3)	RS-232: send the next bit (NMI continuation routine)
61531	($F05B)	RS-232: prepare to receive the next input byte
61718	($F116)	RS-232: open an RS-232 channel for input
65017	($FDF9)	Initialize the 6522 VIA registers
65246	($FEDE)	RS-232: NMI sequences
Character		
199	($C7)	Flag for reversed screen characters
245–246	($F5–F6)	Pointer to which keyboard table being used of four possible
631–640	($277–280)	Ten-byte keyboard buffer
646	($286)	Current foreground color selected by color keys
653	($28D)	Current SHIFT keys pattern
657	($291)	Flag to enable or disable combined SHIFT and Commodore keys
4096–4607	($1000–11FF)	Screen map RAM on VIC-20 with 8K+ expansion
32768–36863	($8000–8FFF)	Character maps
32768–33791	($8000–83FF)	Uppercase and graphics nonreversed screen character map
33792–34815	($8400–87FF)	Reversed uppercase and graphics screen character map
34816–35839	($8800–8BFF)	Lowercase and uppercase nonreversed screen character
35840–36863	($8C00–8FFF)	Reversed lowercase and uppercase screen character map
36867	($9003)	Number of character lines displayed, part of raster location
36868	($9004)	Raster beam location bits 8–1
36869	($9005)	Screen map and character map addresses
40960–49151	($A000–BFFF)	8K RAM expansion block 4
60380	($EBDC)	Set keyboard decode table address in 245–246 ($F5–F6)
60705	($ED21)	Used to set uppercase/graphics character set
60720	($ED30)	Set the environment specified by graphics control characters
Appendix E		Valid character set relocation addresses
CHR$		
36869	($9005)	Screen map and character map addresses
55020	($D6EC)	BASIC CHR$
55179	($D78B)	BASIC ASC
CHRGET		
61–62	($3D–3E)	Saved TXTPTR of statement executing, to CONT on
122–123	($7A–7B)	Get-BASIC-character routine
139–143	($8B–8F)	BASIC RND work area, last random number or initial seed
50332	($C49C)	Store/replace a BASIC program line
51360	($C8A0)	BASIC GOTO
52638	($CD9E)	Formula/expression evaluation
52991	($CEFF)	Syntax check for a specific character in .A from CHRGET
55123	($D7AD)	BASIC VAL
57859	($E203)	Check whether more characters are in the current statement
57870	($E20E)	Insure that a parameter is present after a delimiting comma

58232	($E232)	Perform a cold start of BASIC
58247	($E387)	CHRGET routine and RND seed to be copied to page 0 RAM
58276	($E3A4)	Initialize BASIC: restore CHRGET and page 0 pointers

Clear to send

663	($297)	RS-232 status register
37136	($9110)	Port B I/O register
61422	($EFEE)	RS-232: prepare the next byte to be sent from buffer

Clock—*see* TI, TI$, or Timer (VIA)

CLOSE

73–74	($49–4A)	Pointer to BASIC variable used in FOR loop
152	($98)	Number of currently open files, cannot exceed ten
247–248	($F7–F8)	RS-232: pointer to start of receiving buffer
643–644	($283–284)	Pointer to the end of user RAM memory, plus one
49164	($C00C)	Keyword dispatch vector table, in token order
57796	($E1C4)	BASIC CLOSE
57878	($E216)	Handle parameters for OPEN and CLOSE

CLR

22	($16)	Pointer to available slot in temporary string stack
45–46	($2D–2E)	Pointer to the end of BASIC program, start of variables
47–48	($2F–30)	Pointer to the end of BASIC variables, start of arrays
49–50	($31–32)	Pointer to the end of BASIC arrays, start of free area
51–52	($33–34)	Pointer to the bottom of BASIC active strings
55–56	($37–38)	Pointer to the end of BASIC memory
65–66	($41–42)	Pointer to the current BASIC data item
122–123	($7A–7B)	Get-BASIC-character routine
247–248	($F7–F8)	RS-232: pointer to start of receiving buffer
256–511	($100–1FF)	STACK routine
601–610	($259–262)	Open logical file number table (ten one-byte entries)
643–644	($283–284)	Pointer to the end of user RAM memory, plus one
49164	($C00C)	Keyword dispatch vector table, in token order
50782	($C65E)	BASIC CLR
51313	($C871)	BASIC RUN
57590	($E0F6)	BASIC patch routines
57701	($E165)	BASIC LOAD

CMD

19	($13)	Current channel number for BASIC I/O routines
154	($9A)	Device number of output device
37159	($9127)	Timer 1 high order (MSB) latch byte
49164	($C00C)	Keyword dispatch vector table, in token order
50231	($C437)	BASIC error message routine
50844	($C69C)	BASIC LIST
51840	($CA80)	BASIC PRINT#
51846	($CA86)	BASIC CMD
52159	($CBBF)	BASIC INPUT

Cold restart

64802	($FD22)	Power-on/reset routine (checks for autostart cartridge)

Cold start of BASIC

49152	($C000)	Vector to the routine for the cold start of BASIC
58232	($E378)	Perform a cold start of BASIC
58276	($E3A4)	Initialize BASIC: restore CHRGET and page 0 pointers
58372	($E404)	Display cold start of BASIC messages
58459	($E45B)	Copy BASIC vectors from ROM to RAM

Color

201–202	($C9–CA)	Cursor current logical position (line, column)
209–210	($D1–D2)	Pointer to the start of the logical line that the cursor is on
217–241	($D9–F1)	Screen line link table
243–244	($F3–F4)	Pointer to the current physical screen lines color map area
646	($286)	Current foreground color selected by color keys
647	($287)	Cursor: original color at this screen location
780–783	($30C–30F)	The BASIC SYS command uses this area to save and load
4096–4607	($1000–11FF)	Screen map RAM on VIC-20 with 8K+ expansion
32768–36863	($8000–8FFF)	Character maps
36864–37135	($9000–910F)	6560 VIC chip
36865	($9001)	Bits 7–0: vertical TV picture origin
36866	($9002)	Number of columns displayed, part of screen map address
36868	($9004)	Raster beam location
36869	($9005)	Screen map and character map addresses
36878	($900E)	Sound volume and auxiliary color
36879	($900F)	Background color, border color, inverse color switch
37888–38399	($9400–95FF)	Screen color map (8K+ expanded VIC-20)
38400–38911	($9600–97FF)	Screen color map (unexpanded or 3K expanded VIC-20)
58648	($E518)	Initialize the 6550 VIC chip, screen, and related pointers
59077	($E6C5)	Set up display of a character on the screen
59202	($E742)	Handle characters going to the screen
59666	($E912)	Set the current foreground color code
59681	($E921)	Color code key table
59765	($E975)	Scroll the screen
59990	($EA56)	Move screen line
60014	($EA6E)	The address of the screen line and color line is set in memory
60045	($EA8D)	Blank out a physical screen line
60065	($EAA1)	Synchronize color to byte and store character on screen
60074	($EAAA)	Store a character on the screen
60082	($EAB2)	The address of the color map byte for screen map byte is found

Color map—*see* Color

Commodore key

203	($CB)	Matrix coordinate of current key pressed (64 if none)

657	($291)	Flag to enable or disable combined SHIFT and Commodore keys
4096–4607	($1000–11FF)	Screen map RAM on VIC-20 with 8K+ expansion
32768–33791	($8000–83FF)	Uppercase and graphics nonreversed screen character map
33792–34815	($8400–87FF)	Reversed uppercase and graphics screen character map
34816–35839	($8800–8BFF)	Lowercase and uppercase nonreversed screen character map
35840–36863	($8C00–8FFF)	Reversed lowercase and uppercase screen character map
36869	($9005)	Screen map and character map addresses
59202	($E742)	Handle characters going to the screen
60510	($EC5E)	Table used for decoding unshifted keys into ASCII
60575	($EC9F)	Table used for decoding SHIFTed keys into ASCII
60640	($ECE0)	Table used for decoding Commodore SHIFTed keys into ASCII
60777	($ED69)	Apparently unused keyboard decoding table
60835	($EDA3)	Table used for decoding CTRL SHIFTed keys into ASCII

Also see CTRL; Keyboard

Commodore 64

0	($0)	6502 JMP opcode
1–2	($1–2)	The USR jump vector in LSB/MSB (displacement/page) form
512–600	($200–258)	89-byte BASIC input buffer
631–640	($277–280)	Ten-byte keyboard buffer
645	($285)	Serial: timeout enable/disable flag
663	($297)	RS-232 status register
673–767	($2A1–2FF)	User indirect vectors or other storage area
784–787	($310–313)	Four bytes of unused page 3 space for your use
60380	($EBDC)	Set keyboard decode table address in 245–246 ($F5–F6)
60777	($ED69)	Apparently unused keyboard decoding table
65111	($FE57)	Reset RS-232 status, branch to 65128 ($FE68) for non-RS-232 status

CONT

57–58	($39–3A)	Line number of the BASIC statement being executed
59–60	($3B–3C)	Previous BASIC line number executed
61–62	($3D–3E)	Saved TXTPTR of statement executing, to CONT on
49164	($C00C)	Keyword dispatch vector table, in token order
50844	($C69C)	BASIC LIST
51249	($C831)	BASIC END
51287	($C857)	BASIC CONT

COS

49234	($C052)	Function dispatch vector table, in token order
57953	($E261)	BASIC COS
57960	($E268)	BASIC SIN
58077	($E2DD)	Trig evaluation constant values used for COS, SIN, and TAN

Counter (VIA)—*see* Timer

CTRL

197	($C5)	Matrix coordinate of last key pressed (64 if none)
203	($CB)	Matrix coordinate of current key pressed (64 if none)
212	($D4)	Flag to indicate if within quote marks
245–246	($F5–F6)	Pointer to which keyboard table being used of four possible
646	($286)	Current foreground color selected by color keys
653	($28D)	Current SHIFT keys pattern
657	($291)	Flag to enable or disable combined SHIFT and Commodore keys
32768–36863	($8000–8FFF)	Character maps
37153	($9121)	Port A I/O register
37159	($9127)	Timer 1 high order (MSB) latch byte
37888–38399	($9400–95FF)	Screen color map (8K+ expanded VIC-20)
60190	($EB1E)	Scan the keyboard for keypresses using 6522 VIA2
60486	($EC46)	Keyboard decode table addresses
60510	($EC5E)	Table used for decoding unshifted keys into ASCII
60575	($EC9F)	Table used for decoding SHIFTed keys into ASCII
60777	($ED69)	Apparently unused keyboard decoding table
60835	($EDA3)	Table used for decoding CTRL SHIFTed keys into ASCII

Cursor

9	($9)	Column that the cursor was on just before last TAB or SPC
19	($13)	Current channel number for BASIC I/O routines
145	($91)	Keyswitch PIA: bottom keyboard row scan
153	($99)	Device number of the current input device
197	($C5)	Matrix coordinate of last key pressed (64 if none)
198	($C6)	Number of characters (0–10) in the keyboard buffer at 631 ($277)
200	($C8)	Pointer to the end of line for input
201–202	($C9–CA)	Cursor current logical position (line, column)
204	($CC)	Cursor blink switch: 0=flash, non-0=quiet
205	($CD)	Cursor countdown before blink
206	($CE)	Character under cursor (in screen POKE code)
207	($CF)	Cursor blink status; 1=reversed character, 0=nonreversed
209–210	($D1–D2)	Cursor position within the logical screen line
211	($D3)	Cursor position within the logical screen line
212	($D4)	Flag to indicate if within quote marks
213	($D5)	Current screen line logical length (21,43,65,87)
214	($D6)	Cursor: current physical screen line cursor is on (0–22)
217–241	($D9–F1)	Screen line link table
243–244	($F3–F4)	Pointer to the current physical screen lines color map area
631–640	($277–280)	Ten-byte keyboard buffer
646	($286)	Current foreground color selected by color keys
647	($287)	Cursor: original color at this screen location
650	($28A)	Keyboard repeater flags
780–783	($30C–30F)	The BASIC SYS command uses this area to save and load
4096–4607	($1000–11FF)	Screen map RAM on VIC-20 with 8K+ expansion
36864	($9000)	Left edge of TV picture and interlace switch

375

36865	($9001)	Bits 7–0: vertical TV picture origin
36866	($9002)	Number of columns displayed, part of screen map address
37153	($9121)	Port A I/O register
37159	($9127)	Timer 1 high order (MSB) latch byte
37888–38399	($9400–95FF)	Screen color map (8K+ expanded VIC-20)
50844	($C69C)	BASIC LIST
51960	($CAF8)	BASIC TAB, BASIC SPC
52027	($CB3B)	Part of PRINT: print format characters of space, cursor right
52091	($CB7B)	BASIC GET
52159	($CBBF)	BASIC INPUT
54174	($D39E)	BASIC POS
58634	($E50A)	Read or set the current cursor column and line
58648	($E518)	Initialize the 6550 VIC chip, screen, and related pointers
58753	($E581)	Move the cursor to the screen home position
58853	($E5E5)	Wait for character to appear in the keyboard buffer
59077	($E6C5)	Set up display of a character on the screen
59114	($E6EA)	Advance the cursor on the screen, add lines, and scroll
59181	($E72D)	Back up cursor into the previous logical screen line
59202	($E742)	Handle characters going to the screen
59587	($E8C3)	Advance the cursor to the next logical screen line
59608	($E8D8)	Handle the carriage return key
59624	($E8E8)	Move the cursor to the end of the previous screen line
59642	($E8FA)	Move the cursor to the start of the next screen line
59765	($E975)	Scroll the screen
60065	($EAA1)	Synchronize color to byte and store character on screen
60095	($EABF)	IRQ handler
65520	($FFF0)	JuMP to 58634 ($E50A)

Custom Characters—*see* Character

DATA

15	($F)	Flag byte: LIST quote/collect done/tokenize character
17	($11)	Indicate which of READ, INPUT, or GET is active
43–44	($2B–2C)	Pointer to the start of the tokenized BASIC program
57–58	($39–3A)	Line number of the BASIC statement being executed
63–64	($3F–40)	Current DATA line number in LSB/MSB form
65–66	($43–44)	Pointer to source of INPUT, GET, and READ information
631–640	($277–280)	Ten-byte keyboard buffer
4096–4607	($1000–11FF)	Screen map RAM on VIC-20 with 8K+ expansion
32768–36863	($8000–8FFF)	Character maps
49164	($C00C)	Keyword dispatch vector table, in token order
51229	($C81D)	BASIC RESTORE
51448	($C8F8)	BASIC DATA
52230	($CC06)	BASIC READ
54566	($D526)	Garbage collection

DEF

49164	($C00C)	Keyword dispatch vector table, in token order

54195	($D3B3)	BASIC DEF
54241	($D3E1)	Check DEF FN and FN syntax
54260	($D3F4)	BASIC FN
54351	($D44F)	Store DEF FN values into the function descriptor from stack

Device number

19	($13)	Current channel number for BASIC I/O routines
43–44	($2B–2C)	Pointer to the start of the tokenized BASIC program
73–74	($49–4A)	Pointer to BASIC variable used in FOR loop
152	($98)	Number of currently open files, cannot exceed ten
153	($99)	Device number of the current input device
154	($9A)	Device number of output device
184	($B8)	Current logical file number being used
186	($BA)	Current device number being used
611–620	($263–26C)	Open device number table (ten one-byte entries)
631–640	($277–280)	Ten-byte keyboard buffer
659	($293)	RS-232 pseudo-6551 control register
828–1019	($33C–3FB)	Tape buffer area, 192 bytes, for headers and BASIC program device
828	($33C)	Tape header identifier byte (1–5)
37137	($9111)	Port A I/O register
49310	($C09E)	BASIC keyword table in token number order
50844	($C69C)	BASIC LIST
57701	($E165)	BASIC LOAD
57809	($E1D1)	Set LOAD, VERIFY, and SAVE parameters
57878	($E216)	Handle parameters for OPEN and CLOSE
58811	($E5BB)	Reset the default device numbers
60948	($EE14)	Serial: send talk with attention
60951	($EE17)	Serial: send listen with attention
62151	($F3C7)	Open .X file number channel for input
62217	($F309)	Open .X file number channel for output
62282	($F34A)	Close logical file number in .A
62431	($F3DF)	Set file characteristics of file (.X) into 184–186 ($B8–BA)
62474	($F40A)	Open a logical file, file number in 184 ($B8)
62786	($F542)	LOAD (or VERIFY) to RAM from device number specified in 186 ($BA)
63093	($F675)	Save RAM to device number specified in 186 ($BA)
65104	($FE50)	Set the current file number, device, and secondary address
65111	($FE57)	Reset RS-232 status, branch to 65128 ($FE68) for non-RS-232 status
65130	($FE6A)	OR .A with the contents of 144 ($90) ST and store there
65493	($FFD5)	JuMP to 62786 ($F542)
65496	($FFD8)	JuMP to 63093 ($F675)
Appendix D		Device, secondary address, status chart

DIM

11	($B)	BASIC buffer index/array dimensions
12	($C)	Flags for locate-or-build-array routines
49164	($C00C)	Keyword dispatch vector table, in token order
53377	($D081)	BASIC DIM
53837	($D24D)	Check for redimensioning of an array

Dipole—*see* Tape

377

Double-sized characters

Disk

19	($13)	Current channel number for BASIC I/O routines
43–44	($2B–2C)	Pointer to the start of the tokenized BASIC program
153	($99)	Device number of the current input device
174–175	($AE–AF)	Tape: ending address for LOAD, SAVE, and VERIFY
183	($B7)	Number of characters in filename (0–187 or 0–16)
184	($B8)	Current logical file number being used
185	($B9)	Current secondary address being used (also called *command*)
186	($BA)	Current device number being used
187–188	($BB–BC)	Pointer to the current filename
193–194	($C1–C2)	Tape/Serial: pointer to the start of the I/O area
631–640	($277–280)	Ten-byte keyboard buffer
32768–36863	($8000–8FFF)	Character maps
37146	($911A)	Shift register for parallel/serial conversion
37162	($912A)	Shift register for parallel/serial conversion
52091	($CB7B)	BASIC GET
57701	($E165)	BASIC LOAD
57796	($E1C4)	BASIC CLOSE
62812	($F55C)	Load or verify RAM from a serial device
63122	($F692)	Save RAM to serial device
Appendix D		Device, secondary address, status chart

Also see Serial

Double-sized characters—*see* Character

Dynamic keyboard

198	($C6)	Number of characters (0–10) in the keyboard buffer at 631 ($277)
631–640	($277–280)	Ten-byte keyboard buffer

END

57–58	($39–3A)	Line number of the BASIC statement being executed
59–60	($3B–3C)	Previous BASIC line number executed
61–62	($3D–3E)	Saved TXTPTR of statement executing, to CONT on
49164	($C00C)	Keyword dispatch vector table, in token order
50020	($C364)	Miscellaneous messages
51247	($C82F)	BASIC STOP
51249	($C831)	BASIC END
51287	($C857)	BASIC CONT

Error messages, BASIC

49566	($C19E)	Table of BASIC error messages
49960	($C328)	BASIC error message table vectors

Also see Message

EXP

49234	($C052)	Function dispatch vector table, in token order
57279	($DFBF)	Tables for LOG and EXP, in floating point format
57325	($DFED)	BASIC EXP

Expansion

0–143	($0–8F)	Page 0 working storage for BASIC
43–44	($2B–2C)	Pointer to the start of the tokenized BASIC program

378

55463	($D8A7)	Make the result negative if a borrow was done
55543	($D8F7)	Zero out FAC and make sign positive since result was zero
55550	($D8FE)	Renormalize the FAC result
55623	($D947)	Complement FAC entirely
55786	($D9EA)	BASIC LOG
55848	($DA28)	BASIC * (multiply FAC2 by FAC, leaving the result in FAC)
55948	($DA8C)	Move floating point memory locations to FAC2
55991	($DAB7)	Add exponents of FAC and FAC2
56034	($DAE2)	Multiply FAC by 10
56062	($DAFE)	Divide FAC by 10
56079	($DB0F)	Move floating point in memory to FAC2
56082	($DB12)	BASIC / (divide FAC2 by FAC resulting in FAC)
56226	($DBA2)	Move floating point memory into FAC
56263	($DBC7)	Move FAC to memory
56266	($DBCA)	Move FAC to memory
56272	($DBD0)	Move FAC to memory
56276	($DBD4)	Perform move of FAC to memory
56316	($DBFC)	Transfer FAC2 to FAC
56332	($DC0C)	Move FAC to FAC2, with rounding
56335	($DC0F)	Move FAC to FAC2, without rounding
56347	($DC1B)	Round FAC by adjusting the rounding byte
56363	($DC2B)	Test the sign of FAC
56380	($DC3C)	Convert the sign obtained above to 0 or −1 in FAC
56388	($DC44)	Convert a two-byte integer to floating point in FAC
56408	($DC58)	BASIC ABS
56411	($DC5B)	Compare FAC to memory
56475	($DC9B)	Convert FAC floating point to signed integer
56524	($DCCC)	BASIC INT
56563	($DCF3)	Convert an ASCII string to a floating point number in FAC
56702	($DD7E)	Add .A to FAC
56781	($DDCD)	Decimal number display routine
56797	($DDDD)	Convert FAC to TI$ or an ASCII string
57201	($DF71)	BASIC SQR
57211	($FD7B)	BASIC ↑ (power)
57268	($DFB4)	Monadic minus
57325	($DFED)	BASIC EXP
57430	($E056)	Math series evaluation routine
57639	($E127)	BASIC SYS
57953	($E261)	BASIC COS
57960	($E268)	BASIC SIN
58033	($E2B1)	BASIC TAN
58123	($E30B)	BASIC ATN
Appendix B		Format of variables and floating point accumulators

FAC2—*see* FAC

File number

19	($13)	Current channel number for BASIC I/O routines
73–74	($49–4A)	Pointer to BASIC variable used in FOR loop
152	($98)	Number of currently open files, cannot exceed ten
153	($99)	Device number of the current input device
154	($9A)	Device number of output device
184	($B8)	Current logical file number being used
185	($B9)	Current secondary address being used (also called *command*)

601–610	($259–262)	Open logical file number table (ten one-byte entries)
659	($293)	RS-232 pseudo-6551 control register
51846	($CA86)	BASIC CMD
52091	($CB7B)	BASIC GET
57878	($E216)	Handle parameters for OPEN and CLOSE
62151	($F2C7)	Open .X file number channel for input
62217	($F309)	Open .X file number channel for output
62282	($F34A)	Close logical file number in .A
62415	($F3CF)	Find file number (.X) in file table at 601 ($259)
62431	($F3DF)	Set file characteristics of file (.X) into 184–186 ($B8–BA)
62474	($F40A)	Open a logical file, file number in 184 ($B8)
65104	($FE50)	Set the current file number, device, and secondary address
65466	($FFBA)	JuMP to 65104 ($FE50)
Fire button		
37136–37151	($9110–911F)	6522 VIA chip 1
37137	($9111)	Port A I/O register
FN		
70	($10)	Subscript or FN x flag byte
71–72	($47–48)	Pointer to the descriptor of the current BASIC variable
78–79	($4E–4D)	Pointer to current FN descriptor (in variable storage)
49164	($C00C)	Keyword dispatch vector table, in token order
49310	($C09E)	BASIC keyword table in token number order
52909	($CEAD)	Factoring is continued
53387	($D08B)	Locate or create variable
54190	($D3AE)	Issue an UNDEF'D FUNCTION message for EVALFN ($D3F4)
54195	($D3B3)	BASIC DEF
54241	($D3E1)	Check DEF FN and FN syntax
54260	($D3F4)	BASIC FN
54351	($D44F)	Store DEF FN values into the function descriptor from stack
FOR		
47–48	($2F–30)	Pointer to the end of BASIC variables, start of arrays
57–58	($39–3A)	Line number of the BASIC statement being executed
73–74	($49–4A)	Pointer to BASIC variable used in FOR loop
256–511	($100–1FF)	STACK routine
631–640	($277–280)	Ten-byte keyboard buffer
49164	($C00C)	Keyword dispatch vector table, in token order
50058	($C38A)	Find FOR and GOSUB entries on the stack
50171	($C3FB)	Check stack requested space available
51010	($C742)	BASIC FOR
51410	($C8D2)	BASIC RETURN
52510	($CD1E)	BASIC NEXT
55740	($D9BC)	Constant to zero a floating point accumulator
Foreground—*see* Color		
FRE		
49–50	($31–32)	Pointer to the end of BASIC arrays of free area
51–52	($33–34)	Pointer to the bottom of BASIC active strings

49234	($C052)	Function dispatch vector table, in token order
54141	($D37D)	BASIC FRE

Game port

36864–37135	($9000–910F)	6560 VIC chip
36864	($9000)	Left edge of TV picture and interlace switch
36865	($9001)	Bits 7–0: vertical TV picture origin
36868	($9004)	Raster beam location
36870	($9006)	Light pen horizontal screen location
36871	($9007)	Light pen vertical screen location
36872	($9008)	Potentiometer X/Paddle X value 255
37136–37151	($9110–911F)	6522 VIA chip 1
37136	($9110)	Port B I/O register
37137	($9111)	Port A I/O register
37152–37167	($9120–912F)	6522 VIA chip 2
37152	($9120)	Port B I/O register
37154	($9122)	Data direction register for port B

Garbage collection

15	($F)	Flag byte: LIST quote/collect done/tokenize character
22	($16)	Pointer to available slot in temporary string stack
49–50	($31–32)	Pointer to the end of BASIC arrays, start of free area
51–52	($33–34)	Pointer to the bottom of BASIC active strings
78–79	($4E–4D)	Pointer to current FN descriptor (in variable storage)
83	($43)	Constant for garbage collection (3 or 7)
84–86	($54–56)	Jump opcode and vector to function routine
87–96	($57–60)	BASIC numeric work area
50104	($C3B8)	Open space in memory for a new BASIC line or variable
50111	($C3BF)	Move a block of memory
50184	($C408)	Check that requested space in dynamic area is available
54141	($D37D)	BASIC FRE
54566	($D526)	Garbage collection

GET

17	($11)	Indicate which of READ, INPUT, or GET is active
19	($13)	Current channel number for BASIC I/O routines
67–68	($43–44)	Pointer to source of INPUT, GET, and READ information
73–74	($49–4A)	Pointer to BASIC variable used in FOR loop
75–76	($4B–4C)	Math operator displacement/INPUT TXTPTR
113–114	($71–72)	Series evaluation pointer
145	($91)	Keyswitch PIA: bottom keyboard row scan
198	($C6)	Number of characters (0–10) in the keyboard buffer at 631 ($277)
201–202	($C9–CA)	Cursor current logical position (line, column)
512–600	($200–258)	89-byte BASIC input buffer
631–640	($277–280)	Ten-byte keyboard buffer
37153	($9121)	Port A I/O register
49164	($C00C)	Keyword dispatch vector table, in token order
52045	($CB4D)	Error message formatting routine for GET, INPUT, and READ
52901	($CB7B)	BASIC GET
52230	($CC06)	BASIC READ, also common routines for GET and INPUT

61941	($F1F5)	Routing routine for obtaining a character of input data
GET#		
19	($13)	Current channel number for BASIC I/O routines
153	($99)	Device number of the current input device
828–1019	($33C–3FB)	Tape buffer area, 192 bytes, for headers and BASIC program data
829–1019	($33D–3FB)	Tape block of 191 user data bytes from a BASIC program
52091	($CB7B)	BASIC GET
62063	($F26F)	Obtain a byte from the RS-232 device
GOSUB		
20–21	($14–15)	Line number integer in two-byte LSB/MSB format
43–44	($2B–2C)	Pointer to the start of the tokenized BASIC program
57–58	($39–3A)	Line number of the BASIC statement being executed
73–74	($49–4A)	Pointer to BASIC variable used in FOR loop
256–511	($100–1FF)	STACK
780–783	($30C–30F)	The BASIC SYS command uses this area to save and load
49164	($C00C)	Keyword dispatch vector table, in token order
49234	($C052)	Function dispatch vector table, in token order
50058	($C38A)	Find FOR and GOSUB entries on the stack
50707	($C613)	Find the BASIC line from its line number
51331	($C883)	BASIC GOSUB
51410	($C8D2)	BASIC RETURN
51531	($C94B)	BASIC ON
GOTO		
20–21	($14–15)	Line number integer in two-byte LSB/MSB format
43–44	($2B–2C)	Pointer to the start of the tokenized BASIC program
57–58	($39–3A)	Line number of the BASIC statement being executed
49164	($C00C)	Keyword dispatch vector table, in token order
49280	($C080)	Math operation dispatch vector table, in token order
49310	($C09E)	BASIC keyword table in token number order
50707	($C613)	Find the BASIC line from its line number
51172	($C7E4)	Execute the current BASIC statement
51331	($C883)	BASIC GOSUB
51360	($C8A0)	BASIC GOTO
51496	($C928)	BASIC IF
51531	($C94B)	BASIC ON
51563	($C96B)	Convert decimal line number to LSB/MSB format

GO TO—*see* GOTO

I.D.—*see* Tape

IF		
49164	($C00C)	Keyword dispatch vector table, in token order
51496	($C928)	BASIC IF
51515	($C93B)	BASIC REM
INPUT		
17	($11)	Indicate which of READ, INPUT, or GET is active

19	($13)	Current channel number for BASIC I/O routines
67–68	($43–44)	Pointer to source of INPUT, GET, and READ information
73–74	($49–4A)	Pointer to BASIC variable used in FOR loop
75–76	($4B–4C)	Math operator displacement/INPUT TXTPTR
113–114	($71–72)	Series evaluation pointer
198	($C6)	Number of characters (0–10) in the keyboard buffer at 631 ($277)
201–202	($C9–CA)	Cursor current logical position (line, column)
512–600	($200–258)	89-byte BASIC input buffer
631–640	($277–280)	Ten-byte keyboard buffer
49164	($C00C)	Keyword dispatch vector table, in token order
50231	($C437)	BASIC error message routine
50528	($C560)	Receive input from device and fill the BASIC text buffer
52027	($CB3B)	Part of PRINT: print format characters of space, cursor right
52045	($CB4D)	Error message formatting routine for GET, INPUT, and READ
52091	($CB7B)	BASIC GET
52159	($CBBF)	BASIC INPUT
52230	($CC06)	BASIC READ, also common routines for GET and INPUT
52476	($CCFC)	INPUT error messages
54407	($D487)	Scan and set up string
58959	($E64F)	Obtain INPUT from screen
61941	($F1F5)	Routing routine for obtaining a character of input data

INPUT#

19	($13)	Current channel number for BASIC I/O routines
153	($99)	Device number of the current input device
828–1019	($33C–3FB)	Tape buffer area, 192 bytes, for headers and BASIC program data
829–1019	($33D–3FB)	Tape block of 191 user data bytes from a BASIC program
49164	($C00C)	Keyword dispatch vector table, in token order
50528	($C560)	Receive input from device and fill the BASIC text buffer
52133	($CBA5)	BASIC INPUT#
58853	($E5E5)	Wait for character to appear in the keyboard buffer

INST

212	($D4)	Flag to indicate if within quote marks
216	($D8)	Number of outstanding inserts remaining
59077	($E6C5)	Set up display of a character on the screen

INT

| 49234 | ($C052) | Function dispatch vector table, in token order |
| 56524 | ($DCCC) | BASIC INT |

Interlace

| 36864 | ($9000) | Left edge of TV picture and interlace switch |

Interrupt

Also see IRQ *and* NMI

146	($92)	Tape: 0/1 bit timebase fluctuation during read operations
192	($C0)	Tape: motor interlock switch
197	($C5)	Matrix coordinate of last key pressed (64 if none)

205	($CD)	Cursor countdown before blink
256–511	($100–1FF)	STACK
631–640	($277–280)	Ten-byte keyboard buffer
647	($287)	Cursor: original color at this screen location
652	($28C)	Delay before first repeat of key
788–789	($314–315)	Vector to the routine IRQ at 60095 ($EABF)
790–791	($316–317)	Vector to the interrupt routine BREAK* at 65234 ($FED2)
792–793	($318–319)	Vector to the routine NMI* at 65197 ($FEAD)
814–815	($32E–32F)	User vector can be placed here; held over from PET ML monitor
34816–35839	($8800–8BFF)	Lowercase and uppercase nonreversed screen character map
36868	($9004)	Raster beam location
37136–37151	($9110–911F)	6522 VIA chip 1
37140	($9114)	Timer 1 least significant byte (LSB) of count
37141	($9115)	Timer 1 most significant byte (MSB) of count
37143	($9117)	Timer 1 high order (MSB) latch byte
37144	($9118)	Timer 2 low order (LSB) counter and LSB latch
37145	($9119)	Timer 2 high order (MSB) counter and MSB latch
37146	($911A)	Shift register for parallel/serial conversion
37147	($911B)	Auxiliary control register
37148	($911C)	Peripheral control register for handshaking
37149	($911D)	Interrupt flag register
37150	($911E)	Interrupt enable register
37152–37167	($9120–912F)	6522 VIA chip 2
37153	($9121)	Port A I/O register
37156	($9124)	Timer 1 least significant byte (LSB) of count
37157	($9125)	Timer 1 most significant byte (MSB) of count
37158	($9126)	Timer 1 low (LSB) latch byte
37159	($9127)	Timer 1 high order (MSB) latch byte
37160	($9128)	Timer 2 low order (LSB) counter and LSB latch
37161	($9129)	Timer 2 high order (MSB) counter and MSB latch
37162	($912A)	Shift register for parallel/serial conversion
37163	($912B)	Auxiliary control register
37164	($912C)	Peripheral control register for handshaking
37165	($912D)	Interrupt flag register
37166	($912E)	Interrupt enable register
40960–49151	($A000–BFFF)	8K RAM expansion block 4
51287	($C857)	BASIC CONT
52091	($CB7B)	BASIC GET
60095	($EABF)	IRQ handler
60951	($EE17)	Serial: send listen with attention
61347	($EFA3)	RS-232: send the next bit (NMI continuation routine)
61494	($F036)	RS-232: receive an input bit (NMI driven)
61531	($F05B)	RS-232: prepare to receive the next input byte
61677	($F0ED)	RS-232: store a character in the transmit buffer
61698	($F102)	RS-232: set up NMI interrupts for transmission
61718	($F116)	RS-232: open an RS-232 channel for input
63689	($F8C9)	Tape: read blocks from tape
63715	($F8E3)	Tape: write blocks to tape
63732	($F8F4)	Tape: common tape read/write; start tape operations
63837	($F95D)	Tape: set time limit for tape dipole
63886	($F98E)	Tape: read tape data bits into location 191 ($BF) (IRQ driven)

64518	($FC06)	Tape: end of block write processing
64523	($FC0B)	Tape: data write (IRQ driven)
64680	($FCA8)	Tape: leader write (IRQ driven)
64802	($FD22)	Power-on/reset routine (checks for autostart cartridge)
65017	($FDF9)	Initialize the 6522 VIA registers
65193	($FEA9)	NMI handler routine
65234	($FED2)	BREAK interrupt entry
65246	($FEDE)	RS-232: NMI sequences
65366	($FF56)	Restore 6502 registers from the stack and return from interrupt
65394	($FF72)	IRQ routine initial 6502 entry point
65530	($FFFA)	6502 vector to 65193 ($FEA9)
65534	($FFFE)	6502 vector to 65394 ($FF72)

Inverse color—*see* Screen

IRQ

19	($13)	Current channel number for BASIC I/O routines
192	($C0)	Tape: motor interlock switch
197	($C5)	Matrix coordinate of last key pressed (64 if none)
205	($CD)	Cursor countdown before blink
256–511	($100–1FF)	STACK
631–640	($277–280)	Ten-byte keyboard buffer
647	($287)	Cursor: original color at this screen location
652	($28C)	Delay before first repeat of key
671–672	($29F–2A0)	Temporary save area for the normal IRQ vector during tape I/O
788–789	($314–315)	Vector to the routine IRQ at 60095 ($EABF)
790–791	($316–317)	Vector to the routine BREAK* at 65234 ($FED2)
792–793	($318–319)	Vector to the routine NMI* at 65197 ($FEAD)
37136–37151	($9110–911F)	6522 VIA chip 1
37145	($9119)	Timer 2 high order (MSB) counter and MSB latch
37148	($911C)	Peripheral control register for handshaking
37149	($911D)	Interrupt flag register
37150	($911E)	Interrupt enable register
37152–37167	($9120–912F)	6522 VIA chip 2
37153	($9121)	Port A I/O register
37156	($9124)	Timer 1 least significant byte (LSB) of count
37157	($9125)	Timer 1 most significant byte (MSB) of count
37158	($9126)	Timer 1 low (LSB) latch byte
37159	($9127)	Timer 1 high order (MSB) latch byte
37161	($9129)	Timer 2 high order (MSB) counter and MSB latch
37165	($912D)	Interrupt flag register
37166	($912E)	Interrupt enable register
52091	($CB7B)	BASIC GET
60095	($EABF)	IRQ handler
63284	($F734)	Increment the jiffy clock at 160–162 ($A0–A2)
63689	($F8C9)	Tape: read blocks from tape
63715	($F8E3)	Tape: write blocks to tape
63732	($F8F4)	Tape: common tape read/write; start tape operations
63819	($F94B)	Tape: check for the RUN/STOP key
63837	($F95D)	Tape: set time limit for tape dipole
63886	($F98E)	Tape: read tape data bits into location 191 ($BF) (IRQ driven)

64173	($FAAD)	Tape: determine if to store the input character from tape
64523	($FC0B)	Tape: data write (IRQ driven)
64661	($FC95)	Tape: block leader write (IRQ driven)
64680	($FCA8)	Tape: leader write (IRQ driven)
64719	($FCCF)	Tape: restore IRQ vector
64758	($FCF6)	Tape: reset the current IRQ vector
65009	($FDF1)	IRQ vectors table
65193	($FEA9)	NMI handler routine
65394	($FF72)	IRQ routine initial 6502 entry point
65534	($FFFE)	6502 vector to 65394

Joy—*see* Game port

Jiffy

145	($91)	Keyswitch PIA: bottom keyboard row scan
160–162	($A0–A2)	Jiffy clock, realtime clock
651	($28B)	Delay before other than first repeat of key
652	($28C)	Delay before first repeat of key
788–789	($314–315)	Vector to the routine IRQ at 60095 ($EABF)
790–791	($316–317)	Vector to the routine BREAK* at 65234 ($FED2)
37152–37167	($9120–912F)	6522 VIA chip 2
37157	($9125)	Timer 1 most significant byte (MSB) of count
37159	($9127)	Timer 1 high order (MSB) latch byte
60095	($EABF)	IRQ handler
63284	($F734)	Increment the jiffy clock at 160–162 ($A0–A2)
63328	($F760)	Put jiffy clock from 160–162 ($A0–A2) into .Y, .X, and .A
63335	($F767)	Set time into jiffy clock 160–162 ($A0–A2) from .Y, .X, and .A
63732	($F8F4)	Tape: common tape read/write; start tape operations
65193	($FEA9)	NMI handler routine
65499	($FFDB)	JuMP to 63335 ($F767)
65502	($FFDE)	JuMP to 63328 ($F760)
65514	($FFEA)	JuMP to 63284 ($F734)

Jitters—*see* Interlace

JuMP to

64850	($FD52)	Cause the RAM system vectors to be reset to provided defaults
64855	($FD57)	Read or set system RAM vectors
65418	($FF8A)	JuMP to 64850 ($FD52)
65421	($FF8D)	JuMP to 64855 ($FD57)
65424	($FF90)	JuMP to 65126 ($FE66)
65427	($FF93)	JuMP to 61120 ($EEC0)
65430	($FF96)	JuMP to 61134 ($EECE)
65433	($FF99)	JuMP to 65139 ($FE73)
65436	($FF9C)	JuMP to 65154 ($FE82)
65439	($FF9F)	JuMP to 60190 ($EB1E)
65442	($FFA2)	JuMP to 65135 ($FE6F)
65445	($FFA5)	JuMP to 61209 ($EF19)
65448	($FFA8)	JuMP to 61156 ($EEE4)
65451	($FFAB)	JuMP to 61174 ($EEF6)
65454	($FFAE)	JuMP to 61188 ($EF04)
65457	($FFB1)	JuMP to 60951 ($EE17)

65460	($FFBA)	JuMP to 60948 ($EE14)
65463	($FFB7)	JuMP to 65111 ($FE57)
65466	($FFBA)	JuMP to 65104 ($FE50)
65469	($FFBD)	JuMP to 65097 ($FE49)
65493	($FFD5)	JuMP to 62786 ($F542)
65496	($FFD8)	JuMP to 63093 ($F675)
65499	($FFDB)	JuMP to 63335 ($F767)
65502	($FFDE)	JuMP to 63328 ($F760)
65514	($FFEA)	JuMP to 63284 ($F734)
65517	($FFED)	JuMP to 58629 ($E505)
65520	($FFF0)	JuMP to 58634 ($E50A)
65523	($FFF3)	JuMP to 58624 ($E500)

Keyboard

19	($13)	Current channel number for BASIC I/O routines
43–44	($2B–2C)	Pointer to the start of the tokenized BASIC program
57–58	($39–3A)	Line number of the BASIC statement being executed
145	($91)	Keyswitch PIA: bottom keyboard row scan
153	($99)	Device number of the current input device
154	($9A)	Device number of output device
184	($B8)	Current logical file number being used
185	($B9)	Current secondary address being used (also called *command*)
186	($BA)	Current device number being used
197	($C5)	Matrix coordinate of last key pressed (64 if none)
198	($C6)	Number of characters (0–10) in the keyboard buffer at 631 ($277)
204	($CC)	Cursor blink switch; 0=flash, non-0=quiet
208	($D0)	Flag indicating if input from screen (3) or keyboard (0)
212	($D4)	Flag to indicate if within quote marks
216	($D8)	Number of outstanding inserts remaining
245–246	($F5–F6)	Pointer to which keyboard table being used of four possible
631–640	($277–280)	Ten-byte keyboard buffer
649	($289)	Maximum number of characters in the keyboard buffer
650	($28A)	Keyboard repeater flags
651	($28B)	Delay before other than first repeat of key
652	($28C)	Delay before first repeat of key
653	($28D)	Current SHIFT keys pattern
654	($28E)	Previous SHIFT key pattern
655–656	($28F–290)	Pointer to the default keyboard table setup routine
658	($292)	Screen scroll down enabled flag
788–789	($314–315)	Vector to the routine IRQ at 60095 ($EABF)
810–811	($32A–32B)	Vector to the routine GETIN at 61941 ($F1F5)
32768–36863	($8000–8FFF)	Character maps
37136–37151	($9110–911F)	6522 VIA chip 1
37137	($9111)	Port A I/O register
37152–37167	($9120–912F)	6522 VIA chip 2
37152	($9120)	Port B I/O register
37153	($9121)	Port A I/O register
37154	($9122)	Data direction register for port B
37159	($9127)	Timer 1 high order (MSB) latch byte
37888–38399	($9400–95FF)	Screen color map (8K+ expanded VIC-20)

LIST

15	($F)	Flag byte: LIST quote/collect done/tokenize character
19	($13)	Current channel number for BASIC I/O routines
20–21	($14–15)	Line number integer in two-byte LSB/MSB format
43–44	($2B–2C)	Pointer to the start of the tokenized BASIC program
73–74	($49–4A)	Pointer to BASIC variable used in FOR loop
87–96	($57–60)	BASIC numeric work area
97–102	($61–66)	BASIC floating point accumulator one
153	($99)	Device number of the current input device
37159	($9127)	Timer 1 high order (MSB) latch byte
49164	($C00C)	Keyword dispatch vector table, in token order
49310	($C09E)	BASIC keyword table in token number order
50707	($C613)	Find the BASIC line from its line number
50844	($C69C)	BASIC LIST
50970	($C71A)	List detokenized BASIC keywords
51563	($C96B)	Convert decimal line number to LSB/MSB format

LOAD

10	($A)	Tape: 0=LOAD, 1=VERIFY
19	($13)	Current channel number for BASIC I/O routines
43–44	($2B–2C)	Pointer to the start of the tokenized BASIC program
45–46	($2D–2E)	Pointer to the end of BASIC program, start of variables
47–48	($2F–30)	Pointer to the end of BASIC variables, start of arrays
49–50	($31–32)	Pointer to the end of BASIC arrays, start of free area
61–62	($3D–3E)	Saved TXTPTR of statement executing, to CONT on
73–74	($49–4A)	Pointer to BASIC variable used in FOR loop
147	($93)	Tape: 0=LOAD, 1=VERIFY
150	($96)	Tape: block found flag, tape leader length bit count
155	($9B)	Tape: character parity
156	($9C)	Tape: dipole switch/byte-received flag
158	($9E)	Tape: error log index/filename index/header I.D./out byte
159	($9F)	Tape: pass 2 error pointer/tape buffer filename index
164	($A4)	Serial: input byte/cycle counter. Tape: dipole number
165	($A5)	Tape: block sync countdown. Serial: countdown
167	($A7)	Tape: write leader count/read block reverse counter
168	($A8)	Tape: error flags; 0=no errors/long word marker switch
169	($A9)	Tape: dipole balance counter/medium word marker switch
170	($AA)	Tape: input status flags, sync countdown/RS-232 byte assembly
171	($AB)	Tape: write leader counter/read checksum comparison
172–173	($AC–AD)	Tape/Serial: start address for LOAD, SAVE, and VERIFY
174–175	($AE–AF)	Tape: end address for LOAD, SAVE, and VERIFY
176	($B0)	Tape: dipole timing adjustment values

177	($B1)	Tape: dipole timing timer 2 difference
178–179	($B2–B3)	Tape: pointer to tape buffer
181	($B5)	Tape: flag for currently reading data or leader
182	($B6)	Tape: accumulator for number of read errors
183	($B7)	Number of characters in filename (0–187 or 0–16)
185	($B9)	Current secondary address being used (also called *command*)
187–188	($BB–BC)	Pointer to the current filename
189	($BD)	RS-232: send parity calculation work byte
190	($BE)	Tape: which copy of block remaining to read/write
193–194	($C1–C2)	Tape/Serial: pointer to the start of the I/O area
195–196	($C3–C4)	Pointer to the RAM area being LOADed
256–318	($100–13E)	62 bytes of tape error log, indexes of bad data
816–817	($330–331)	Vector to the routine LOAD at 62793 ($F549)
828–1019	($33C–3FB)	Tape buffer area, 192 bytes, for headers and BASIC program data
828	($33C)	Tape header identifier byte (1–5)
829–830	($33D–33E)	Starting address of where the tape data was written from
49164	($C00C)	Keyword dispatch vector table, in token order
57701	($E165)	BASIC LOAD
57809	($E1D1)	Set LOAD, VERIFY, and SAVE parameters
58486	($E476)	Program patch area
60916	($EDF4)	LOAD and RUN words for SHIFT and RUN keys
61625	($F089)	RS-232: ILLEGAL DEVICE message for LOAD or SAVE
62786	($F542)	Load (or verify) to RAM from device number specified in 186 ($BA)
62812	($F55C)	Load or verify RAM from a serial device
63407	($F7AF)	Tape: find next tape header, .X back contains header I.D. number
63565	($F84D)	Tape: load tape buffer address from 178–179 ($B2–B3) into .X and .Y
63572	($F854)	Tape: set LOAD/SAVE start and end pointers to the tape buffer
63680	($F8C0)	Tape: initiate tape header read
63715	($F8E3)	Tape: write blocks to tape
63732	($F8F4)	Tape: common tape read/write; start tape operations
63837	($F95D)	Tape: set time limit for tape dipole
64173	($FAAD)	Tape: determine if to store the input character from tape
64466	($F8D2)	Tape: called to reset the tape read pointer
64523	($FC0B)	Tape: data write (IRQ driven)
64680	($FCA8)	Tape: leader write (IRQ driven)
64785	($FD11)	Compare current to end of LOAD/SAVE pointers (tape and serial)
64795	($FD18)	Increment current LOAD/SAVE pointer (tape and serial)
65097	($FE49)	The filename pointer and length are stored from .X, .Y, and .A
65104	($FE50)	Set the current file number, device, and secondary address
Appendix D		Device, secondary address, status chart
Appendix F		Block SAVE/LOAD from BASIC programs

391

LOG

49234	($C052)	Function dispatch vector table, in token order
55745	($D9C1)	Constants for LOG function
55786	($D9EA)	BASIC LOG
57279	($DFBF)	Tables for LOG and EXP, in floating point format
57325	($DFED)	BASIC EXP

Lowercase—*see* Character

Message

1–2	($1–2)	The USR jump vector in LSB/MSB (displacement/page) form
12	($C)	Flags for locate-or-build-array routines
15	($F)	Flag byte: LIST quote/collect done/tokenize character
17	($11)	Indicate which of READ, INPUT, or GET is active
19	($13)	Current channel number for BASIC I/O routines
22	($16)	Pointer to available slot in temporary string stack
43–44	($2B–2C)	Pointer to the start of the tokenized BASIC program
57–58	($39–3A)	Line number of the BASIC statement being executed
63–64	($3F–40)	Current DATA line number in LSB/MSB form
153	($99)	Device number of the current input device
157	($9D)	Kernal message control flag
183	($B7)	Number of characters in filename (0–187 or 0–16)
198	($C6)	Number of characters (0–10) in the keyboard buffer at 631 ($277)
256–511	($100–1FF)	STACK
320–511	($140–1FF)	Stack area used by BASIC, OUT OF MEMORY message if exceeded
631–640	($277–280)	Ten-byte keyboard buffer
768–778	($300–30A)	Table of indirect BASIC vectors
768–769	($300–301)	Vector to the routine to print a BASIC error message from a table
828–1019	($33C–3FB)	Tape buffer area, 192 bytes, for headers and BASIC program data
49310	($C09E)	BASIC keyword table in token number order
49566	($C19E)	Table of BASIC error messages
49960	($C328)	BASIC error message table vectors
50171	($C3FB)	Check stack requested space available
50184	($C408)	Check that requested space in dynamic area is available
50229	($C435)	Set OUT OF MEMORY error message code
50231	($C437)	BASIC error message routine
50281	($C469)	Display ERROR or another message pointed to
50292	($C474)	Display READY. message
50528	($C560)	Receive input from device and fill the BASIC text buffer
50782	($C65E)	BASIC CLR
50844	($C59C)	BASIC LIST
51287	($C857)	BASIC CONT
51998	($CB1E)	Part of PRINT: print a string ended by a carriage return
52045	($CB4D)	Error message formatting routine for GET, INPUT, and READ
52159	($CBBF)	BASIC INPUT

52230	($CC06)	BASIC READ, also common routines for GET and INPUT
52476	($CCFC)	INPUT error messages
52510	($CD1E)	BASIC NEXT
52600	($CD8A)	Variable type checking
53000	($CF08)	Cause SYNTAX ERROR message via jump to $C437
53533	($D11D)	Create new variable
53829	($D245)	Display BAD SUBSCRIPT message
53832	($D248)	Display ILLEGAL QUANTITY message
53837	($D24D)	Found the array, check the subscript range
54182	($D2A6)	Check if a statement is entered in direct mode
54190	($D3AE)	Issue an UNDEF'D FUNCTION message for EVALFN ($D3F4)
54407	($D487)	Scan and set up string
54516	($D4F4)	Allocate memory space for a string
54845	($D63D)	BASIC +, concatenate string
55678	($D97E)	Issue OVERFLOW message and exit
56770	($DDC2)	Issue message IN
57325	($DFED)	BASIC EXP
57590	($E0F6)	BASIC patch routines
57701	($E165)	BASIC LOAD
57870	($E20E)	Insure that a parameter is present after a delimiting comma
58232	($E378)	Perform a cold start of BASIC
58372	($E404)	Display cold start of BASIC messages
58409	($E429)	BASIC cold start messages
58471	($E467)	Perform a warm start of BASIC
58566	($E4BC)	Program patch area
61625	($F0B9)	RS-232: ILLEGAL DEVICE message for LOAD or SAVE
61812	($F174)	Table of Kernal messages
61922	($F1E2)	Display LOADING or VERIFYING if control messages wanted
61926	($F1E6)	Print Kernal control messages
62812	($F55C)	Load or verify RAM from a serial device
63047	($F647)	Display SEARCHING.... for tape device
63082	($F66A)	Display LOADING or VERIFYING
63272	($F728)	Display SAVING message
63358	($F77E)	I/O error file error message handler
63407	($F7AF)	Tape: find next tape header, .X back contains header I.D. number
63636	($F894)	Tape: display PRESS PLAY ON TAPE message
63671	($F8B7)	Tape: display PRESS RECORD & PLAY ON TAPE message
63689	($F8C9)	Tape: read blocks from tape
63715	($F8E3)	Tape: write blocks to tape
65126	($FE66)	Set the byte used to enable/disable Kernal message display
65424	($FF90)	JuMP to 65126 ($FE66)
MID$		
84–86	($54–56)	Jump opcode and vector to function routine
54566	($D526)	Garbage collection
55095	($D737)	BASIC MID$
55137	($D761)	Obtain string parameters for LEFT$, MID$, and RIGHT$

61792	($F160)	RS-232: check that serial and tape are idle, to protect from RS-232
63284	($F734)	Increment the jiffy clock at 160–162 ($A0–A2)
63732	($F8F4)	Tape: common tape read/write, start tape operations
64831	($FD3F)	Check for an autostarting program at 40960 ($A000)
65193	($FEA9)	NMI handler routine
65246	($FEDE)	RS-232: NMI sequences
65530	($FFFA)	6502 vector to 65193 ($FEA9)

NOT

49164	($C00C)	Keyword dispatch vector table, in token order
49280	($C080)	Math operation dispatch vector table, in token order
49310	($C09E)	BASIC keyword table in token number order
52909	($CEAD)	Factoring is continued
52948	($CED4)	BASIC NOT
57268	($DFB4)	BASIC monadic —

ON

20–21	($14–15)	Line number integer in two-byte LSB/MSB format
49164	($C00C)	Keyword dispatch vector table, in token order
51531	($C94B)	BASIC ON
51563	($C96B)	Convert decimal line number to LSB/MSB format

OPEN

152	($98)	Number of currently open files, cannot exceed ten
183	($B7)	Number of characters in filename (0–187 or 0–16)
184	($B8)	Current logical file number being used
187–188	($BB–BC)	Pointer to the current filename
247–248	($F7–F8)	RS-232: pointer to start of receiving buffer
643–644	($283–284)	Pointer to the end of user RAM memory, plus one
660	($294)	RS-232 pseudo-6551 command register
833–1019	($341–3FD)	Filename of tape data
49164	($C00C)	Keyword dispatch vector table, in token order
50782	($C65E)	BASIC CLR
57809	($E1D1)	Set LOAD, VERIFY, and SAVE parameters
57878	($E216)	Handle parameters for OPEN and CLOSE
Appendix D		Device, secondary address, status chart

OR

7	($7)	Search-character for scanning BASIC statements
11	($B)	BASIC buffer index/array dimensions
18	($12)	TAN/SIN sign/comparison results
780–783	($30C–30F)	The BASIC SYS command uses this area to save and load
49280	($C080)	Math operation dispatch vector table, in token order
53222	($CFE6)	BASIC OR
53225	($CFE9)	BASIC AND

Oscillator—*see* Sound

PA0, PA1, PA2, PA3, PA4, PA5, PA6, PA7—*see* VIA

Paddle—*see* Game port

Pause key—*see* Keyboard

PB0, PB1, PB2, PB3, PB4, PB5, PB6, PB7—*see* VIA

PEEK

20–21	($14–15)	Line number integer in two-byte LSB/MSB format
49234	($C052)	Function dispatch vector table, in token order
55309	($D80D)	BASIC PEEK

PI (?)

52867	($CE83)	Evaluate a single term of an expression
52904	($CEA8)	The floating point number ?=$82 49 0F DA A1

Pixel—*see* Screen

Pixel map—*see* Screen

POKE

20–21	($14–15)	Line number integer in two-byte LSB/MSB format
157	($9D)	Kernal message control flag
49164	($C00C)	Keyword dispatch vector table, in token order
55195	($D79B)	Obtain number 1–255
55275	($D7EB)	Get two parameters for POKE and WAIT
55332	($D824)	BASIC POKE
61812	($F174)	Table of Kernal messages

Port A—*see* VIA

Port B—*see* VIA

POS

49234	($C052)	Function dispatch vector table, in token order
54174	($D39E)	BASIC POS
58634	($E50A)	Read or set the current cursor column and line

Power on

0–143	($0–8F)	Page 0 working storage for BASIC
0	($0)	6502 JMP opcode 76 ($4C)
1–2	($1–2)	The USR jump vector in LSB/MSB (displacement/ page) form
3–4	($3–4)	Vector to floating point to integer conversion routines
5–6	($5–6)	Vector to the integer to floating point conversion routines
19	($13)	Current channel number for BASIC I/O routines
22	($16)	Pointer to available slot in temporary string stack
43–44	($2B–2C)	Pointer to the start of the tokenized BASIC program
51–52	($33–34)	Pointer to the bottom of BASIC active strings
55–56	($37–38)	Pointer to the end of BASIC memory
122–123	($7A–7B)	Get-BASIC-character routine
139–143	($8B–8F)	BASIC RND work area, last random number or initial seed
153	($99)	Device number of the current input device
154	($9A)	Device number of output device
160–162	($A0–A2)	Jiffy clock, realtime clock
178–179	($B2–B3)	Tape: pointer to tape buffer
195–196	($C3–C4)	Pointer to the RAM area being LOADed
256–511	($100–1FF)	STACK
320–511	($140–1FF)	Stack area used by BASIC, OUT OF MEMORY message if exceeded
641–642	($281–282)	Pointer to the start of user RAM memory
643–644	($283–284)	Pointer to the end of user RAM memory, plus one

646	($286)	Current foreground color selected by color keys
648	($288)	Screen map RAM page number
655–656	($28F–290)	Pointer to the default keyboard table setup routine
788–819	($314–333)	Table of 16 Kernal indirect vectors
36866	($9002)	Number of columns displayed, part of screen map address
37136–37151	($9110–911F)	6522 VIA chip 1
37136	($9110)	Port B I/O register
37138	($9112)	Data direction register for port B
37139	($9113)	Eight bits, each of which corresponds to the same-numbered bit
37147	($911B)	Auxiliary control register
37148	($911C)	Peripheral control register for handshaking
37152–37167	($9120–912F)	6522 VIA chip 2
37154	($9122)	Data direction register for port B
37155	($9123)	Eight bits, each of which corresponds to the same-numbered bit
37163	($912B)	Auxiliary control register
37164	($912C)	Peripheral control register for handshaking
37888–38399	($9400–95FF)	Screen color map (8K+ expanded VIC-20)
40960–49151	($A000–BFFF)	8K RAM expansion block 4
49152	($C000)	Vector to the routine for the cold start of BASIC 58232 ($E378)
50754	($C642)	BASIC NEW
58232	($E378)	Perform a cold start of BASIC
58528–65535	($E4A0–FFFF)	Kernal* routine
58648	($E518)	Initialize the 6550 VIC chip, screen, and related pointers
58819	($E5C3)	Reset the VIC chip registers
60900	($EDE4)	Initial values for VIC chip registers
64802	($FD22)	Power-on/reset routine (check for autostart cartridge)
64850	($FD52)	Cause the RAM system vectors to be reset to provided defaults
65017	($FDF9)	Initialize the 6522 VIA registers
65532	($FFFC)	6502 vector to 64802 ($FD22)

Precedence order

49280	($C080)	Math operation dispatch vector table, in token order

PRINT

19	($13)	Current channel number for BASIC I/O routines
653	($28D)	Current SHIFT keys pattern
657	($291)	Flag to enable or disable combined SHIFT and Commodore keys
32768–36863	($800–8FFF)	Character maps
36866	($9002)	Number of columns displayed, part of screen map address
37888–38399	($9400–95FF)	Screen color map (8K+ expanded VIC-20)
49164	($C00C)	Keyword dispatch vector table, in token order
50553	($C579)	Tokenize the BASIC line in BASIC text buffer
51872	($CAA0)	BASIC PRINT
51944	($CAE8)	Part of PRINT: tab to the correct column for comma delimiter
51960	($CAF8)	BASIC TAB, BASIC SPC
51998	($CB1E)	Part of PRINT: print a string ended by a carriage return

52027	($CB3B)	Part of PRINT: print format characters of space, cursor right
54407	($D487)	Scan and set up string
58634	($E50A)	Read or set the current cursor column and line

PRINT#

19	($13)	Current channel number for BASIC I/O routines
154	($9A)	Device number of output device
174–175	($AE–AF)	Tape: ending address for LOAD, SAVE, and VERIFY
828–1019	($33C–3FB)	Tape buffer area, 192 bytes, for headers and BASIC program data
829–1019	($33D–3FB)	Tape block of 191 user data bytes from a BASIC program
49164	($C00C)	Keyword dispatch vector table, in token order
51840	($CA80)	BASIC PRINT#
51846	($CA86)	BASIC CMD

Printer

19	($13)	Current channel number for BASIC I/O routines
153	($99)	Device number of the current input device
185	($B9)	Current secondary address being used (also called *command*)
186	($BA)	Current device number being used
32768–36863	($8000–8FFF)	Character maps
37136	($9110)	Port B I/O register
37159	($9127)	Timer 1 high order (MSB) latch byte
50844	($C69C)	BASIC LIST
51960	($CAF8)	BASIC TAB, BASIC SPC
54174	($D39E)	BASIC POS
Appendix D		Device, secondary address, status chart

Also see Serial

Raster—*see* Screen

READ

17	($11)	Indicate which of READ, INPUT, or GET is active
63–64	($3F–40)	Current DATA line number in LSB/MSB form
65–66	($41–42)	Pointer to the current BASIC data item
67–68	($43–44)	Pointer to source of INPUT, GET, and READ information
73–74	($49–4A)	Pointer to BASIC variable used in FOR loop
75–76	($4B–4C)	Math operator displacement/INPUT TXTPTR
113–114	($71–72)	Series evaluation pointer
49164	($C00C)	Keyword dispatch vector table, in token order
52045	($CB4D)	Error message formatting routine for GET, INPUT, and READ
52230	($CC06)	BASIC READ, also common routines for GET and INPUT
54407	($D487)	Scan and set up string
61941	($F1F5)	Routing routine for obtaining a character of input data

REM

43–44	($2B–2C)	Pointer to the start of the tokenized BASIC program
49164	($C00C)	Keyword dispatch vector table, in token order
50844	($C69C)	BASIC LIST

51496	($C928)	BASIC IF
51515	($C93B)	BASIC REM

Repeat—*see* Keyboard

Reset

0	($0)	6502 JMP opcode 76 ($4C)
1–2	($1–2)	The USR jump vector in LSB/MSB (displacement/page) form
3–4	($3–4)	Vector to floating point to integer conversion routines
5–6	($5–6)	Vector to the integer to floating point conversion routines
19	($13)	Current channel number for BASIC I/O routines
22	($16)	Pointer to available slot in temporary string stack
43–44	($2B–2C)	Pointer to the start of the tokenized BASIC program
45–46	($2D–2E)	Pointer to the end of BASIC program, start of variables
51–52	($33–34)	Pointer to the bottom of BASIC active strings
55–56	($37–38)	Pointer to the end of BASIC memory
122–123	($7A–7B)	Get-BASIC-character routines
139–143	($8B–8F)	BASIC RND work area, last random number of initial seed
153	($99)	Device number of the current input device
154	($9A)	Device number of output device
178–179	($B2–B3)	Tape: pointer to tape buffer
195–196	($C3–C4)	Pointer to the RAM area being LOADed
256–511	($100–1FF)	STACK
320–511	($140–1FF)	Stack area used by BASIC, OUT OF MEMORY message if exceeded
641–642	($281–282)	Pointer to the start of user RAM memory
643–644	($283–284)	Pointer to the end of user RAM memory, plus one
646	($286)	Current foreground color selected by color keys
648	($288)	Screen map RAM page number
655–656	($28F–290)	Pointer to the default keyboard table setup routine
788–819	($314–333)	Table of 16 Kernal indirect vectors
36866	($9002)	Number of columns displayed, part of screen map address
37136–37151	($9110–911F)	6522 VIA chip 1
37136	($9110)	Port B I/O register
37138	($9112)	Data direction register for port B
37139	($9113)	Eight bits, each of which corresponds to the same-numbered bit
37147	($911B)	Auxiliary control register
37148	($911C)	Peripheral control register for handshaking
37152–37167	($9120–912F)	6522 VIA chip 2
37154	($9122)	Data direction register for port B
37155	($9123)	Eight bits, each of which corresponds to the same-numbered bit
37163	($912B)	Auxiliary control register
37164	($912C)	Peripheral control register for handshaking
37888–38399	($9400–95FF)	Screen color map (8K+ expanded VIC-20)
40960–4915	($A000–BFFF)	8K RAM expansion block 4
49152	($C000)	Vector to the routine for the cold start of BASIC 58232 ($E378)

50754	($C642)	BASIC NEW
58232	($E378)	Perform a cold start of BASIC
58528–65535	($E4A0–FFFF)	Kernal* routine
58648	($E518)	Initialize the 6550 VIC chip, screen, and related pointers
58819	($E5C3)	Reset the VIC chip registers
60900	($EDE4)	Initial values for VIC chip registers
64802	($FD22)	Power-on/reset routine (check for autostart cartridge)
64850	($FD52)	Cause the RAM system vectors to be reset to provided defaults
65017	($FDF9)	Initialize the 6522 VIA registers
65532	($FFFC)	6502 vector to 64802 ($FD22)

Reset switch

0–143	($0–8F)	Page 0 working storage for BASIC
0	($0)	6502 JMP opcode 76 ($4C)
45–46	($2D–2E)	Pointer to the end of BASIC program, start of variables
37136–37151	($9110–911F)	6522 VIA chip 1

RESTORE

65–66	($41–42)	Pointer to the current BASIC data item
49164	($C00C)	Keyword dispatch vector table, in token order
51229	($C81D)	BASIC RESTORE
58486	($E476)	Program patch area

RESTORE key—*see* STOP/RESTORE

RETURN

57–58	($39–3A)	Line number of the BASIC statement being executed
73–74	($49–4A)	Pointer to BASIC variable used in FOR loop
49164	($C00C)	Keyword dispatch vector table, in token order
51331	($C883)	BASIC GOSUB
51410	($C8D2)	BASIC RETURN

Reverse—*see* Screen; Keyboard

RIGHT$

84–86	($54–56)	Jump opcode and vector to function routine
54566	($D526)	Garbage collection
55084	($D72C)	BASIC RIGHT$
55137	($D761)	Obtain string parameters for LEFT$, MID$, and RIGHT$

RND

139–143	($8B–8F)	BASIC RND work area, last random number or initial seed
49234	($C052)	Function dispatch vector table, in token order
57482	($E08A)	Table of constants for RND
57492	($E094)	BASIC RND
58247	($E387)	CHRGET routine and RND seed to be copied to page 0 RAM
58624	($E500)	Retrieve the address of the I/O memory page

RS-232

19	($13)	Current channel number for BASIC I/O routines
55–56	($37–38)	Pointer to the end of BASIC memory
139–143	($8B–8F)	BASIC RND work area, last random number or initial seed

400

146	($92)	Tape: 0/1 bit timebase fluctuation during read operations
153	($99)	Device number of the current input device
167	($A7)	Tape: write leader count/read block reverse counter
168	($A8)	Tape: error flags; 0=no errors/long word marker switch
169	($A9)	Tape: dipole balance counter/medium word marker switch
170	($AA)	Tape: input status flags, sync countdown/RS-232 byte assembly
171	($AB)	Tape: write leader counter/read checksum comparison
178–179	($B2–B3)	Tape: pointer to tape buffer
180	($B4)	Tape: miscellaneous flags/RS-232: various usage
181	($B5)	Tape: flag for currently reading data or leader
182	($B6)	Tape: accumulator for number of read errors
183	($B7)	Number of characters in filename (0–187 or 0–16)
186	($BA)	Current device number being used
187–188	($BB–BC)	Pointer to the current filename
189	($BD)	RS-232: send parity calculation work byte
247–248	($F7–F8)	RS-232: pointer to start of receiving buffer
249–250	($F9–FA)	RS-232: pointer to the start of the transmitting buffer
643–644	($283–284)	Pointer to the end of user RAM memory, plus one
659	($293)	RS-232 pseudo-6551 control register
660	($294)	RS-232 pseudo-6551 command register
661–662	($295–296)	RS-232 nonstandard bit timing specification
663	($297)	RS-232 status register
664	($298)	RS-232 number of bits to be sent/received
665–666	($299–29A)	RS-232 system clock divided by baud rate=microseconds
667	($29B)	RS-232 dynamic index to the end of the receive buffer
668	($29C)	RS-232 dynamic index to the start of the receive buffer
669	($29D)	RS-232 dynamic index to the start of the transmit buffer
670	($29E)	RS-232 dynamic index to the end of the transmit buffer
37136–37151	($9110–911F)	6522 VIA chip 1
37136	($9110)	Port B I/O register
37137	($9111)	Port A I/O register
37140	($9114)	imer 1 least significant byte (LSB) of count
37141	($9115)	Timer 1 most significant byte (MSB) of count
37143	($9117)	Timer 1 high order (MSB) latch byte
37145	($9119)	Timer 2 high order (MSB) counter and MSB latch
37148	($911C)	Peripheral control register for handshaking
37149	($911D)	Interrupt flag register
37150	($911E)	Interrupt enable register
50782	($C65E)	BASIC CLR
57590	($E0F6)	BASIC patch routines
58853	($E5E5)	Wait for character to appear in the keyboard buffer
60951	($EE17)	Serial: send listen with attention
61347	($EFA3)	RS-232: send the next bit (NMI continuation routine)
61375	($EFBF)	RS-232: calculate parity and stop bits value

47–48	($2F–30)	Pointer to the end of BASIC variables, start of arrays
49–50	($31–32)	Pointer to the end of BASIC arrays, start of free area
157	($9D)	Kernal message control flag
40960–49151	($A000–BFFF)	8K RAM expansion block 4
49164	($C00C)	Keyword dispatch vector table, in token order
50782	($C65E)	BASIC CLR
51313	($C871)	BASIC RUN
58853	($E5E5)	Wait for character to appear in the keyboard buffer
60916	($EDF4)	LOAD and RUN words for SHIFT and RUN keys

RVSOFF

199	($C7)	Flag for reversed screen characters
32768–36863	($8000–8FFF)	Character maps
32678–33791	($8000–83FF)	Uppercase and graphics nonreversed screen character map
34816–35839	($8800–8BFF)	Lowercase and uppercase nonreversed screen character map
35840–36863	($8C00–8FFF)	Reversed lowercase and uppercase screen character map
36869	($9005)	Screen map and character map addresses
59202	($E742)	Handle characters going to the screen

RVSON

199	($C7)	Flag for reversed screen characters
212	($D4)	Flag to indicate if within quote marks
32768–36863	($8000–8FFF)	Character maps
33792–34815	($8400–87FF)	Reversed uppercase and graphics screen character map
35840–36863	($8C00–8FFF)	Reversed lowercase and uppercase screen character map
36869	($9005)	Screen map and character map addresses
59202	($E742)	Handle characters going to the screen

SAVE

43–44	($2B–2C)	Pointer to the start of the tokenized BASIC program
45–46	($2D–2E)	Pointer to the end of BASIC program, start of variables
147	($93)	Tape: 0=LOAD, 1=VERIFY
155	($9B)	Tape: character parity
158	($9E)	Tape: error log index/filename index/header I.D./out byte
165	($A5)	Tape: block sync countdown/Serial: countdown
168	($A8)	Tape: error flags, 0=no errors/long word marker switch
169	($A9)	Tape: dipole balance counter/medium word marker switch
170	($AA)	Tape: input status flags, sync countdown/RS-232 byte assembly
171	($AB)	Tape: write leader counter/read checksum comparison
172–173	($AC–AD)	Tape/Serial: start address for LOAD/SAVE/VERIFY
174–175	($AE–AF)	Tape: ending address for LOAD/SAVE/VERIFY
178–179	($B2–B3)	Tape: pointer to tape buffer
183	($B7)	Number of characters in filename (0–187 or 0–16)

186	($BA)	Current device number being used
193–194	($C1–C2)	Tape/Serial: pointer to the start of the I/O area
197	($C5)	Matrix coordinate of last key pressed (64 if none)
198	($C6)	Number of characters (0–10) in the keyboard buffer at 631 ($277)
199	($C7)	Flag for reversed screen characters
200	($C8)	Pointer to the end of line for input
201–202	($C9–CA)	Cursor current logical position (line, column)
206	($CE)	Character under cursor (in screen POKE code)
208	($D0)	Flag indicating if input from screen (3) or keyboard (0)
209–210	($D1–D2)	Pointer to the start of the logical line that the cursor is on
211	($D3)	Cursor position within the logical screen line
212	($D4)	Flag to indicate if within quote marks
213	($D5)	Current screen line logical length (21,43,65,87)
214	($D6)	Cursor: current physical screen line cursor is on (0–22)
215	($D7)	ASCII value of last key pressed
216	($D8)	Number of outstanding inserts remaining
217–241	($D9–F1)	Screen line link table
242	($F2)	Save byte for screen line link table byte
243–244	($F3–F4)	Pointer to the current physical screen lines color map area
512–600	($200–258)	89-byte BASIC input buffer
631–640	($277–280)	Ten-byte keyboard buffer
646	($286)	Current foreground color selected by color keys
647	($287)	Cursor: original color at this screen location
648	($288)	Screen map RAM page number
653	($28D)	Current SHIFT keys pattern
658	($292)	Screen scroll down enabled flag
7680–8191	($1E00–1FFF)	Screen map RAM on VIC-20 with less than 8K expansion
7680–8191	($1E00–1FFF)	Screen map RAM on VIC-20 with only 3K expansion
4096–4607	($1000–11FF)	Screen map RAM on VIC-20 with 8K+ expansion
32768–36863	($8000–8FFF)	Character maps
32768–33791	($8000–83FF)	Uppercase and graphics nonreversed screen character map
33792–34815	($8400–87FF)	Reversed uppercase and graphics screen character map
34816–35839	($8800–8BFF)	Lowercase and uppercase nonreversed screen character map
35840–36863	($8C00–8FFF)	Reversed lowercase and uppercase screen character map
36864–37135	($9000–910F)	6560 VIC chip
36864	($9000)	Left edge of TV picture and interlace switch
36865	($9001)	Bits 7–0: vertical TV picture origin
36866	($9002)	Number of columns displayed, part of screen map address
36867	($9003)	Number of character lines displayed, part of raster location
36868	($9004)	Raster beam location
36869	($9005)	Screen map and character map addresses
36870	($9006)	Light pen horizontal screen location
36871	($9007)	Light pen vertical screen location

36879	($900F)	Background color, border color, inverse color switch
37159	($9127)	Timer 1 high order (MSB) latch byte
37888–38399	($9400–95FF)	Screen color map (8K+ expanded VIC-20)
50528	($C560)	Receive input from device and fill the BASIC text buffer
50844	($C69C)	BASIC LIST
52159	($CBBF)	BASIC INPUT
54174	($D39E)	BASIC POS
58471	($E467)	Perform a warm start of BASIC
58629	($E505)	Retrieve the maximum number of screen columns and lines
58634	($E50A)	Read or set the current cursor column and line
58648	($E518)	Initialize the 6550 VIC chip, screen, and related pointers
58719	($E55F)	Clear the screen
58753	($E581)	Move the cursor to the screen home position
58759	($E587)	Reset the screen line link table pointers
58811	($E5BB)	Reset the default device numbers
58959	($E64F)	Obtain INPUT from screen
59064	($E6B8)	Test for quotes and set flag
59077	($E6C5)	Set up display of a character on the screen
59144	($E6EA)	Advance the cursor on the screen, add lines, and scroll
59181	($E72D)	Back up cursor into the previous logical screen line
59202	($E742)	Handle characters going to the screen
59587	($E8C3)	Advance cursor to the next logical screen line
59624	($E8E8)	Move the cursor to the end of the previous screen line
59642	($E8FA)	Move the cursor to the start of the next screen line
59765	($E975)	Scroll the screen
59886	($E9EE)	Open up a blank physical line on the screen for inserts
59990	($EA56)	Move screen line
60014	($EA6E)	The address of the screen line and color line is set in memory
60030	($EA7E)	Set a pointer to the address of the start of a screen line
60045	($EA8D)	Blank out a physical screen line
60065	($EAA1)	Synchronize color to byte and store character on screen
60074	($EAAA)	Store a character on the screen
60082	($EAB2)	The address of the color map byte for screen map is found
60763	($ED5B)	Called by routine SCROLL ($E6EA) to mark the next physical screen line
60925	($EDFD)	Screen line link table LSB of lines in screen map
61966	($F20E)	Input characters from current input device
62074	($F27A)	Output character to current output device
62451	($F3F3)	Abort all open channels
62786	($F542)	Load (or verify) to RAM from device number specified in 186 ($BA)
63122	($F692)	Save RAM to serial device
64802	($FD22)	Power-on/reset routine (checks for autostart cartridge)
64909	($FD8D)	Initialize system memory
65234	($FED2)	BREAK interrupt entry

153	($99)	Device number of the current input device
163	($A3)	Serial: input bit count/Tape: input/output bit count
164	($A4)	Serial: input byte/cycle counter/Tape: dipole number
165	($A5)	Tape: block sync countdown/Serial: countdown
172–173	($AC–AD)	Tape/Serial: start address for LOAD/SAVE/VERIFY
174–175	($AE–AF)	Tape: ending address for LOAD, SAVE, and VERIFY
178–179	($B2–B3)	Tape: pointer to tape buffer
185	($B9)	Current secondary address being used (also called *command*)
186	($BA)	Current device number being used
193–194	($C1–C2)	Tape/Serial: pointer to the start of the I/O area
198	($C6)	Number of characters (0–10) in the keyboard buffer at 631 ($277)
645	($285)	Serial: timeout enable/disable flag
828–1019	($33C–3FB)	Tape buffer area, 192 bytes, for headers and BASIC program data
37137	($9111)	Port A I/O register
37139	($9113)	Eight bits, each of which corresponds to the same-numbered bit
37146	($911A)	Shift register for parallel/serial conversion
37148	($911C)	Peripheral control register for handshaking
37150	($911E)	Interrupt enable register
37160	($9128)	Timer 2 low order (LSB) counter and LSB latch
37161	($9129)	Timer 2 high order (MSB) counter and MSB latch
37162	($912A)	Shift register for parallel/serial conversion
37164	($912C)	Peripheral control register for handshaking
37166	($912E)	Interrupt enable register
57878	($E216)	Handle parameters for OPEN and CLOSE
58528	($E4A0)	Serial: output a 1 on the serial data line
58537	($E4A9)	Serial: output a 0 on the serial data line
58546	($E4B2)	Serial: get an input bit from VIA1 and stabilize
60948	($EE14)	Serial: send talk with attention
60951	($EE17)	Serial: send listen with attention
60956	($EE1C)	Serial: prepare to send serial command with attention
61001	($EE49)	Serial: send command or data to serial devices
61108	($EEBA)	Serial: set ST for timeout or DEVICE NOT PRESENT
61120	($EEC0)	Serial: send secondary address after listen command
61125	($EEC5)	Serial: clear attention
61134	($EECE)	Serial: send secondary address after talk command
61156	($EEE4)	Serial: send a byte on the serial line
61174	($EEF6)	Serial: send untalk command to serial devices
61188	($EF04)	Serial: send unlisten command to serial devices
61209	($EF19)	Serial: receive byte from serial device
61316	($EF84)	Serial: set clock line high
61325	($EF8D)	Serial: set clock line low
61334	($EF96)	Serial: delay one millisecond
61792	($F160)	RS-232: check that serial and tape are idle, to protect from RS-232
61966	($F20E)	Input characters from current input device
62052	($F264)	Obtain a byte from the serial line

62074	($F27A)	Output character to current output device
62151	($F2C7)	Open .X file number channel for input
62217	($F309)	Open .X file number channel for output
62451	($F3F3)	Abort all open channels
62474	($F40A)	Open a logical file, file number in 184 ($B8)
62613	($F495)	Send secondary address and filename in a serial device
62786	($F542)	Load (or verify) to RAM from device number specified in 186 ($BA)
62812	($F55C)	Load or verify RAM from a serial device
63122	($F692)	Save RAM to serial device
64785	($FD11)	Compare current to end of LOAD/SAVE pointers (tape and serial)
64795	($FD1B)	Increment current LOAD/SAVE pointer (tape and serial)
65427	($FF93)	JuMP to 61120 ($EEC0)
65430	($FF96)	JuMP to 61134 ($EECE)
65445	($FFA5)	JuMP to 61209 ($EF19)
65448	($FF48)	JuMP to 61156 ($EEE4)
65451	($FFAB)	JuMP to 61174 ($EEF6)
65454	($FFAE)	JuMP to 61188 ($EF04)
65457	($FFB1)	JuMP to 60951 ($EE17)
65460	($FFB4)	JuMP to 50948 ($EE14)
Appendix D		Device, secondary address, status chart

Serial port—*see* Serial

SGN

97–102	($61–66)	BASIC Floating Point Accumulator 1
49234	($C052)	Function dispatch vector table, in token order
52909	($CEAD)	Factoring is continued
56377	($DC39)	BASIC SGN

SHIFT

183	($B7)	Number of characters in filename (0–187 or 0–16)
203	($CB)	Matrix-coordinate of current key pressed (64 if none)
657	($291)	Flag to enable or disable combined SHIFT and Commodore keys
4096–4607	($1000–11FF)	Screen map RAM on VIC-20 with 8K+ expansion
33792–34815	($8400–87FF)	Reversed uppercase and graphics screen character map
34816–35839	($8800–88FF)	Lowercase and uppercase nonreversed screen character map
35840–36863	($8C00–8FFF)	Reversed lowercase and uppercase screen character map
36869	($9005)	Screen map and character map addresses
59202	($E742)	Handle characters going to the screen
60510	($EC5E)	Table used for decoding unshifted keys into ASCII
60835	($EDA3)	Table used for decoding CTRL SHIFTed keys into ASCII

Also see Keyboard

Shift register—*see* VIA

SIN

18	($12)	TAN/SIN sign/comparison results
49234	($C052)	Function dispatch vector table, in token order

57960	($E268)	BASIC SIN
58077	($E2DD)	Trig evaluation constant values used for COS, SIN, and TAN
Sound		
36864–37135	($9000–910F)	6560 VIC CHIP
36869	($9005)	Screen map and character map addresses
36874	($900A)	Relative frequency of sound oscillator 1 (bass)
36875	($900B)	Relative frequency of sound oscillator 2 (alto)
36876	($900C)	Relative frequency of sound oscillator 3 (soprano)
36877	($900D)	Relative frequency of sound oscillator 4 (noise)
36878	($900E)	Sound volume and auxiliary color
37146	($911A)	Shift register for parallel/serial conversion
37147	($911B)	Auxiliary control register
37162	($912A)	Shift register for parallel/serial conversion
37163	($912B)	Auxiliary control register
SPC		
9	($9)	Column that the cursor was on just before last TAB or SPC
49164	($C00C)	Keyword dispatch vector table, in token order
49310	($C09E)	BASIC keyword table in token number order
51872	($CAA0)	BASIC PRINT
51960	($CAF8)	BASIC TAB, BASIC SPC
58634	($E50A)	Read or set the current cursor column and line
SQR		
49234	($C052)	Function dispatch vector table, in token order
57105	($DF11)	0.5 constant for rounding and SQR
57201	($DF71)	BASIC SQR
ST		
144	($90)	ST status of I/O completion
663	($297)	RS-232 status register
51872	($CAA0)	BASIC PRINT
52091	($CB7B)	BASIC GET
53032	($CF28)	Obtain variable name and type
53533	($D11D)	Create new variable
57701	($E165)	BASIC LOAD
61108	($EEB4)	Serial: set ST for timeout or DEVICE NOT PRESENT
62000	($F230)	Obtain a byte from the tape buffer
62052	($F264)	Obtain a byte from the serial line
64173	($FAAD)	Tape: determine if to store the input character from tape
65111	($FE57)	Reset RS-232 status, branch to 61528 ($FE68) for non-RS-232 status
65128	($FE68)	Load .A with the non-RS-232 I/O status ST
65130	($FE6A)	OR .A with the contents of 144 ($90) ST and store there
Appendix D		Device, secondary address, status chart
STEP		
73–74	($49–4A)	Pointer to BASIC variable used in FOR loop
49164	($C00C)	Keyword dispatch vector table, in token order
49310	($C09E)	BASIC keyword table in token number order
52510	($CD1E)	BASIC NEXT
55740	($D9BC)	Constant to zero a floating point accumulator
STOP		
57–58	($29–3A)	Line number of the BASIC statement being executed

410

59–60	($3B–3C)	Previous BASIC line number executed
49164	($C00C)	Keyword dispatch vector table, in token order
50020	($C364)	Miscellaneous messages
51247	($C82F)	BASIC STOP
51287	($C857)	BASIC CONT

STOP key

57–58	($39–3A)	Line number of the BASIC statement being executed
59–60	($3B–3C)	Previous BASIC line number executed
145	($91)	Keyswitch PIA: bottom keyboard row scan
160–162	($A0–A2)	Jiffy clock, realtime clock
195–196	($C3–C4)	Pointer to the RAM area being LOADed
197	($C5)	Matrix coordinate of last key pressed (64 if none)
198	($C6)	Number of characters (0–10) in the keyboard buffer at 631 ($277)
671–672	($29F–2A0)	Temporary save area for the normal IRQ vector during tape I/O
788–819	($314–333)	Table of 16 Kernal indirect vectors
788–789	($314–315)	Vector to the IRQ at 60095 ($EABF)
808–809	($328–329)	Vector to the test STOP key routine at 63344 ($F770)
37136–37151	($9110–911F)	6522 VIA chip 1
37138	($9112)	Data direction register for port B
37139	($9113)	Eight bits, each of which corresponds to the same-numbered bit
37147	($911B)	Auxiliary control register
37148	($911C)	Peripheral control register for handshaking
37149	($911D)	Interrupt flag register
37152–37167	($9120–912F)	6522 VIA chip 2
37152	($9120)	Port B I/O register
37153	($9121)	Port A I/O register
37154	($9122)	Data direction register for port B
37155	($9123)	Eight bits, each of which corresponds to the same-numbered bit
37159	($9127)	Timer 1 high order (MSB) latch byte
37163	($912B)	Auxiliary control register
37164	($912C)	Peripheral control register for handshaking
50020	($C364)	Miscellaneous messages
50844	($C69C)	BASIC LIST
51118	($C7AE)	Find (for execution) the next BASIC statement
51244	($C82C)	Test for STOP key
51287	($C857)	BASIC CONT
54566	($D526)	Garbage collection
59765	($E975)	Scroll the screen
60095	($EABF)	IRQ handler
62451	($F3F3)	Abort all open channels
62812	($F55C)	Load or verify RAM from a serial device
63122	($F692)	Save RAM to serial device
63217	($F6F1)	Save RAM to tape
63284	($F734)	Increment the jiffy clock at 160–162 ($A0–A2)
63344	($F770)	Check for STOP key in $91, purge keyboard queue and channels if so
63407	($F7AF)	Tape: find next tape header, .X back contains header I.D. number
63463	($F7E7)	Tape: build an output tape header in the tape buffer area

63591	($F867)	Tape: find the tape header for a specified filename (or next)
63636	($F894)	Tape: display PRESS PLAY ON TAPE message
63732	($F8F4)	Tape: common tape read/write, start tape operations
63819	($F94B)	Tape: check for the STOP key
64719	($FCCF)	Tape: restore IRQ vector
65193	($FEA9)	NMI handler routine
65505	($FFE1)	JuMP off 808–809 ($328–329)

STOP/RESTORE

19	($13)	Current channel number for BASIC I/O routines
198	($C6)	Number of characters (0–10) in the keyboard buffer at 631 ($277)
646	($286)	Current foreground color selected by color keys
649	($289)	Maximum number of characters in the keyboard buffer
788–819	($314–333)	Table of 16 Kernal indirect vectors
790–791	($316–317)	Vector to the routine BREAK* at 65234 ($FED2)
792–793	($318–319)	Vector to the routine NMI* at 65197 ($FEAD)
36864–37135	($9000–910F)	6560 VIC chip
36866	($9002)	Number of columns displayed, part of screen map address
37136–37151	($9110–911F)	6522 VIA chip 1
37138	($9112) Data direction register for port B	
37139	($9113)	Eight bits, each of which corresponds to the same-numbered bit
37147	($911B)	Auxiliary control register
37148	($911C)	Peripheral control register for handshaking
37152–37167	($9120–912F)	6522 VIA chip 2
37154	($9122) Data direction register for port B	
37155	($9123)	Eight bits, each of which corresponds to the same-numbered bit
37159	($9127)	Timer 1 high order (MSB) latch byte
37163	($912B)	Auxiliary control register
37164	($912C)	Peripheral control register for handshaking
49154	($C002)	Vector to the routine to do the warm start of BASIC
50782	($C65E)	BASIC CLR
58471	($E467)	Perform a warm start of BASIC
58648	($E518)	Initialize the 6550 VIC chip, screen, and related pointers
58819	($E5C3)	Reset the VIC chip registers
60900	($EDE4)	Initial values for VIC chip registers
64580	($FD52)	Cause the RAM system vectors to be reset to provided defaults
65017	($FDF9)	Initialize the 6522 VIA registers
65193	($FEA9)	NMI handler routine
65234	($FED2)	BREAK interrupt entry

STR$

54373	($D465)	BASIC STR$
54407	($D487)	Scan and set up string
55179	($D78B)	BASIC ASC

SYS

1–2	($1–2)	The USR jump vector in LSB/MSB (displacement/page) form
19	($13)	Current channel number for BASIC I/O routines
20–21	($14–15)	Line number integer in two-byte LSB/MSB format
153	($99)	Device number of the current input device
154	($9A)	Device number of output device
217–241	($D9–F1)	Screen line link table
780–783	($30C–30F)	The BASIC SYS command uses this area to save and load
828–1019	($33C–3FB)	Tape buffer area, 192 bytes, for headers and BASIC program data
49164	($C00C)	Keyword dispatch vector table, in token order
55287	($D7F7)	Convert floating point FAC to two-byte positive integer
58639	($E127)	BASIC SYS

TAB

9	($9)	Column that the cursor was on just before last TAB or SPC
49164	($C00C)	Keyword dispatch vector table, in token order
49310	($C09E)	BASIC keyword table in token number order
51872	($CAA0)	BASIC PRINT
51960	($CAF8)	BASIC TAB, BASIC SPC
58634	($E50A)	Read or set the current cursor column and line

TAN

18	($12)	TAN/SIN sign/comparison results
49234	($C052)	Function dispatch vector table, in token order
57960	($E268)	BASIC SIN
58033	($E2B1)	BASIC TAN
58077	($E2DD)	Trig evaluation constant values used for COS, SIN, and TAN

Tape

19	($13)	Current channel number for BASIC I/O routines
113–114	($71–72)	Series evaluation pointer
146	($92)	Tape: 0/1 bit timebase fluctuation during read operations
150	($96)	Tape: block found flag, tape leader length bit count
153	($99)	Device number of the current input device
155	($9B)	Tape: character parity
156	($9C)	Tape: dipole switch/byte-received flag
158	($9E)	Tape: error log index/filename index/header I.D./out byte
159	($9F)	Tape: pass 2 error pointer/tape buffer filename index
163	($A3)	Serial: input bit count/Tape: input/output bit count
164	($A4)	Serial: input byte/cycle counter/Tape: dipole number
165	($A5)	Tape: block sync countdown/Serial: countdown
166	($A6)	Tape: count of characters in the tape buffer
167	($A7)	Tape: write leader count/read block reverse counter
168	($A8)	Tape: error flags; 0=no errors/long word marker switch
169	($A9)	Tape: dipole balance counter/medium word marker switch
170	($AA)	Tape: input status flags, sync countdown/RS-232 byte assembly

171	($AB)	Tape: write leader counter/read checksum comparison
172–173	($AC–AD)	Tape/Serial: start address for LOAD, SAVE, VERIFY
174–175	($AE–AF)	Tape: ending address for LOAD, SAVE, and VERIFY
176	($B0)	Tape: dipole timing adjustment values
177	($B1)	Tape: dipole timing timer 2 difference
178–179	($B2–B3)	Tape: pointer to tape buffer
180	($B4)	Tape: miscellaneous flags/RS-232: various usage
181	($B5)	Tape: flag for currently reading data or leader
182	($B6)	Tape: accumulator for number of read errors
183	($B7)	Number of characters in filename (0–187 or 0–16)
185	($B9)	Current secondary address being used (also called *command*)
186	($BA)	Current device number being used
187–188	($BB–BC)	Pointer to the current filename
189	($BD)	RS-232: send parity calculation work byte
191	($BF)	Tape: input byte currently being constructed
192	($C0)	Tape: motor interlock switch
193–194	($C1–C2)	Tape/Serial: pointer to the start of the I/O area
195–196	($C3–C4)	Pointer to the RAM area being LOADed
215	($D7)	ASCII value of last key pressed
256–318	($100–13E)	62 bytes of tape error log, indexes of bad data
631–640	($277–280)	Ten-byte keyboard buffer
671–672	($29F–2A0)	Temporary save area for the normal IRQ vector during tape I/O
788–789	($314–315)	Vector to the routine IRQ at 60095 ($EABF)
828–1019	($33C–3FB)	Tape buffer area, 192 bytes, for headers and BASIC program data
828	($33C)	Tape header identifier byte (1–5)
829–830	($33D–33E)	Starting address of where the tape data was written from
831–832	($33F–340)	Ending address, plus one, of where the tape data was written from
833–1019	($341–3FD)	Filename of tape data
1020–1023	($3FC–3FF)	Unused area
37137	($9111)	Port A I/O register
37140	($9114)	Timer 1 least significant byte (LSB) of count
37143	($9117)	Timer 1 high order (MSB) latch byte
37148	($911C)	Peripheral control register for handshaking
37150	($911E)	Interrupt enable register
37152	($9120)	Port B I/O register
37156	($9124)	Timer 1 least significant byte (LSB) of count
37157	($9125)	Timer 1 most significant byte (MSB) of count
37158	($9126)	Timer 1 low (LSB) latch byte
37159	($9127)	Timer 1 high order (MSB) latch byte
37160	($9128)	Timer 2 low order (LSB) counter and LSB latch
37161	($9129)	Timer 2 high order (MSB) counter and MSB latch
37164	($912C)	Peripheral control register for handshaking
37166	($912E)	Interrupt enable register
50844	($C69C)	BASIC LIST
52091	($CB7B)	BASIC GET
57701	($E165)	BASIC LOAD
57796	($E1C4)	BASIC CLOSE
58566	($E4BC)	Program patch area

414

60095	($EABF)	IRQ handler
61792	($F160)	RS-232: check that serial and tape are idle, to protect from RS-232
62000	($F230)	Obtain a byte from the tape buffer
62032	($F250)	Load .A with next tape character, getting block when needed
62096	($F290)	Output a character to tape
62282	($F34A)	Close logical file number in .A
62474	($F40A)	Open a logical file, file number in 184 ($B8)
62786	($F542)	Load (or verify) to RAM from device number specified in 186 ($BA)
62812	($F55C)	Load or verify RAM from a serial device
63047	($F647)	Display SEARCHING.... message for tape device
63122	($F692)	Save RAM to serial device
63217	($F6F1)	Save RAM to tape
63407	($F7AF)	Tape: find next tape header, .X back contains header I.D. number
63463	($F7E7)	Tape: build an output tape header in the tape buffer area
65365	($F84D)	Tape: load tape buffer address from 178–179 ($B2–B3) into .X and .Y
63572	($F854)	Tape: set LOAD/SAVE start and end pointers to the tape buffer
63591	($F867)	Tape: find the tape header for a specified filename (or next)
63626	($F88A)	Tape: increment the tape buffer character counter
63636	($F894)	Tape: display PRESS PLAY ON TAPE message
63659	($F8AB)	Tape: check tape play/rewind/forward button status
63671	($F8B7)	Tape: display PRESS RECORD & PLAY ON TAPE message
63680	($F8C0)	Tape: initiate tape header read
63689	($F8C9)	Tape: read blocks from tape
63715	($F8E3)	Tape: write blocks to tape
63732	($F8F4)	Tape: common tape read/write, start tape operations
63819	($F94B)	Tape: check for the STOP key
63837	($F95D)	Tape: set time limit for tape dipole
63886	($F98E)	Tape: read tape data bits into location 191 ($BF) (IRQ driven)
64173	($FAAD)	Tape: determine if to store the input character from tape
64466	($FBD2)	Tape: called to reset the tape read pointer
64475	($FBDB)	Tape: new tape character setup
64490	($FBEA)	Tape: toggle the tape write line to invert the output signal
64523	($FC0B)	Tape: data write (IRQ driven)
64661	($FC95)	Tape: block leader write (IRQ driven)
64680	($FCA8)	Tape: leader write (IRQ driven)
64719	($FCCF)	Tape: restore IRQ vector
64748	($FCF6)	Tape: reset the current IRQ vector
64776	($FD08)	Tape: kill motor
64785	($FD11)	Compare current to end of LOAD/SAVE pointers (tape and serial)
64795	($FD1B)	Increment current LOAD/SAVE pointer (tape and serial)

415

64909	($FD8D)	Initialize system memory
65009	($FDF1)	IRQ vectors table
65097	($FE49)	The filename pointer and length are stored from .X, .Y, and .A
Appendix D		Device, secondary address, status chart

Tape button—*see* Tape

Tape port—*see* Tape

TV

36864	($9000)	Left edge of TV picture and interlace switch
36865	($9001)	Bits 7–0: vertical TV picture origin
36866	($9002)	Number of columns displayed, part of screen map address
36867	($9003)	Number of character lines displayed, part of raster location
36868	($9004)	Raster beam location
36870	($9006)	Light pen horizontal screen location
36871	($9007)	Light pen vertical screen location
36878	($900E)	Sound volume and auxiliary color
37159	($9127)	Timer 1 high order (MSB) latch byte

TV columns, TV lines, TV picture origin—*see* TV

THEN

49164	($C00C)	Keyword dispatch vector table, in token order
49310	($C09E)	BASIC keyword table in token number order
51496	($C928)	BASIC IF

TI—*see* Timer

TI$

113–114	($71–72)	Series evaluation pointer
160–162	($A0–A2)	Jiffy clock, realtime clock
256–266	($100–10A)	Temporary floating point to ASCII work area for printing numbers
37157	($9125)	Timer 1 most significant byte (MSB) of count
37159	($9127)	Timer 1 high order (MSB) latch byte
51621	($C9A5)	BASIC LET
51674	($C9DA)	LET: assign TI$
51872	($CAA0)	BASIC PRINT
53032	($CF28)	Obtain variable name and type
53533	($D11D)	Create new variable
55341	($D82D)	BASIC WAIT
56034	($DAE2)	Multiply FAC by 10
56797	($DDDD)	Convert FAC to TI$ or an ASCII string
57146	($DF3A)	Constants for TI$ division conversion
63328	($F760)	Put jiffy clock from 160–162 ($A0–A2) into .Y, .X, and .A
63335	($F767)	Set time into jiffy clock 160–162 ($A0–A2) from .Y, .X, and .A

Timer

139–143	($8B–8F)	BASIC RND work area, last random number or initial seed
177	($B1)	Tape: dipole timing timer 2 difference
180	($B4)	Tape: miscellaneous flags/RS-232: various usage
665–666	($299–29A)	RS-232 system clock divided by baud rate=microseconds

788–789	($314–315)	Vector to the routine IRQ at 60095 ($EABF)
792–793	($318–319)	Vector to the routine NMI* at 65197 ($FEAD)
37136	($9110)	Port B I/O register
37140	($9114)	Timer 1 least significant byte (LSB) of count
37141	($9115)	Timer 1 most significant byte (MSB) of count
37143	($9117)	Timer 1 high order (MSB) latch byte
37145	($9119)	Timer 2 high order (MSB) counter and MSB latch
37146	($911A)	Shift register for parallel/serial conversion
37147	($911B)	Auxiliary control register
37149	($911D)	Interrupt flag register
37156	($9124)	Timer 1 least significant byte (LSB) of count
37157	($9125)	Timer 1 most significant byte (MSB) of count
37158	($9126)	Timer 1 low (LSB) latch byte
37159	($9127)	Timer 1 high order (MSB) latch byte
37160	($9128)	Timer 2 low order (LSB) counter and LSB latch
37161	($9129)	Timer 2 high order (MSB) counter and MSB latch
37162	($912A)	Shift register for parallel/serial conversion
37163	($912B)	Auxiliary control register
37165	($912D)	Interrupt flag register
37166	($912E)	Interrupt enable register
58624	($E500)	Retrieve the address of the I/O memory page
60095	($EABF)	IRQ handler
61334	($EF96)	Serial: delay one millisecond
61494	($F036)	RS-232: receive an input bit (NMI driven)
61531	($F05B)	RS-232: prepare to receive the next input byte
61698	($F102)	RS-232: set up NMI for transmission
61718	($F116)	RS-232: open an RS-232 channel for input
62663	($F4C7)	RS-232: open RS-232 device
63732	($F8F4)	Tape: common tape read/write, start tape operations
63837	($F95D)	Tape: set time limit for tape dipole
64490	($FBEA)	Tape: toggle the tape write line to invert the output signal
64523	($FCOB)	Tape: data write (IRQ driven)
64680	($FCA8)	Tape: leader write (IRQ driven)
65017	($FDF9)	Initialize the 6522 VIA registers
65193	($FEA9)	NMI handler routine
65246	($FEDE)	RS-232: NMI sequences
65372	($FF5C)	RS-232: VIA timer 2 values for baud rate table

TO

73–74	($49–4A)	Pointer to BASIC variable used in FOR loop
49164	($C00C)	Keyword dispatch vector table, in token order
49310	($C09E)	BASIC keyword table in token number order
52510	($CD1E)	BASIC NEXT

Token

8	($8)	Scan-quotes flag for scanning BASIC statements
11	($B)	BASIC buffer index/array dimensions
15	($F)	Flag byte: LIST quote/collect done/tokenize character
43–44	($2B–2C)	Pointer to the start of the tokenized BASIC program
61–62	($3D–3E)	Saved TXTPTR of statement executing, to CONT on
113–114	($71–72)	Series evaluation pointer
122–123	($7A–7B)	Get-BASIC-character routine
153	($99)	Device number of the current input device

512–600	($200–258)	89-byte BASIC input buffer
772–773	($304–305)	Vector to the BASIC tokenization routine
774–775	($306–307)	Vector to the BASIC routine that expands and prints tokens
776–777	($308–309)	Vector to the BASIC routine that executes the next BASIC token
49164	($C00C)	Keyword dispatch vector table, in token order
49234	($C052)	Function dispatch vector table, in token order
49280	($C080)	Math operation dispatch vector table, in token order
49310	($C09E)	BASIC keyword table in token number order
50332	($C49C)	Store/replace a BASIC program line
50553	($C579)	Tokenize the BASIC line in BASIC text buffer
50844	($C69C)	BASIC LIST
50970	($C71A)	List detokenized BASIC keywords
51172	($C7E4)	Execute the current BASIC statement
51531	($C94B)	BASIC ON
54241	($D3E1)	Check DEF FN and FN syntax
Appendix C		Tokens for BASIC keywords in numerical order

Uppercase—*see* Character

User port

19	($13)	Current channel number for BASIC I/O routines
153	($99)	Device number of the current input device
186	($BA)	Current device number being used
37136–37151	($9110–911F)	6522 VIA chip 1
37136	($9110)	Port B I/O register
37137	($9111)	Port A I/O register
37140	($9114)	Timer 1 least significant byte (LSB) of count
37146	($911A)	Shift register for parallel/serial conversion
37162	($912A)	Shift register for parallel/serial conversion

USR

0	($0)	6502 JMP opcode 76 ($4C)
1–2	($1–2)	The USR jump vector in LSB/MSB (displacement/page) form
49234	($C052)	Function dispatch vector table, in token order

VAL

113–114	($71–72)	Series evaluation pointer
49234	($C052)	Function dispatch vector table, in token order
55170	($D782)	Get string information
55213	($D7AD)	BASIC VAL
56563	($DCF3)	Convert an ASCII string to a floating point number in FAC

Variables

45–46	($2D–2E)	Pointer to the end of BASIC program, start of variables
49–50	($31–32)	Pointer to the end of BASIC arrays, start of free area
50104	($C3B8)	Open space in memory for a new BASIC line or variable
50111	($C3BF)	Move a block of memory
50782	($C65E)	BASIC CLR
51010	($C742)	BASIC FOR
51621	($C9A5)	BASIC LET

52638	($CD9E)	Formula/expression evaluation
53377	($D081)	BASIC DIM
54141	($D37D)	BASIC FRE
Appendix B		Format of variables and floating point accumulators

Vector to

3–4	($3–4)	Vector to the routines that convert floating point to integer
5–6	($5–6)	Vector to the routines that convert integer to floating point
84–86	($54–56)	Jump opcode and vector to function routine
768–769	($300–301)	Vector to the routine to print a BASIC error message from a table
700–711	($302–303)	Vector to the BASIC main routine (execute or store statement)
772–773	($304–305)	Vector to the BASIC tokenization routine
774–775	($306–307)	Vector to the BASIC routine that expands and prints tokens
776–777	($308–309)	Vector to the BASIC routine that executes the next BASIC token
778–779	($30A–30B)	Vector to the BASIC routine that evaluates a variable
788–789	($314–315)	Vector to the routine IRQ at 60095 ($EABF)
790–791	($316–317)	Vector to the routine BREAK* at 65234 ($FED2)
792–793	($318–319)	Vector to the routine NMI* at 65197 ($FEAD)
794–795	($31A–31B)	Vector to the open logical file routine OPEN at 62474 ($F40A)
796–797	($31C–31D)	Vector to the close logical file routine CLOSE at 62282 ($F34A)
798–799	($31E–31F)	Vector to the open input channel routine CHKIN at 62151 ($F2C7)
800–801	($320–321)	Vector to the open output channel routine CHKOUT at 62217 ($F309)
802–803	($322–323)	Vector to the reset all channels routine CLRCHN at 62451 ($F3F3)
804–805	($324–325)	Vector to the input-from-device routine CHRIN at 61966 ($F20E)
806–807	($326–327)	Vector to the output-to-device routine CHROUT at 62074 ($F27A)
808–809	($328–329)	Vector to the test STOP key routine STOP at 63344 ($F770)
810–811	($32A–32B)	Vector to the get-from-keyboard routine GETIN at 61941 ($F1F5)
812–813	($32C–32D)	Vector to the abort all files routine CLALL at 62447 ($F3EF)
814–815	($32E–32F)	User vector can be placed here; held over from PET ML monitor
816–817	($330–331)	Vector to the load from device routine LOAD at 62793 ($F549)
818–819	($332–333)	Vector to the Kernal SAVE to device routine SAVE at 63109 ($F685)
49152	($C000)	Vector to the routine for the cold start of BASIC 58232 ($E378)
49154	($C002)	Vector to the routine to do the warm start of BASIC 58471 ($E467)
49164	($C00C)	BASIC keyword handler routines dispatch vector table

49234	($C052)	BASIC function handler routines dispatch vector table
49280	($C080)	BASIC math operation handler routines dispatch vector table
64719	($FCCF)	Tape: restore IRQ vector
65472	($FFC0)	JuMP off 794–795 ($31A–31B)
65475	($FFC3)	JuMP off 796–797 ($31C–31D)
65478	($FFC6)	JuMP off 798–799 ($31E–31F)
65481	($FFC9)	JuMP off 800–801 ($320–321)
65484	($FFCC)	JuMP off 802–803 ($322–323)
65487	($FFCF)	JuMP off 804–805 ($324–325)
65490	($FFD2)	JuMP off 806–807 ($326–327)
65505	($FFE1)	JuMP off 808–809 ($328–329)
65508	($FFE4)	JuMP off 810–811 ($32A–32B)
65511	($FFE7)	JuMP off 812–813 ($32C–32D)
65530	($FFFA)	6502 vector to 65193 ($FEA9)
65532	($FFFC)	6502 vector to 64802 ($FD22)
65534	($FFFE)	6502 vector to 65394 ($FF72)

Also see JuMP to

VERIFY

10	($A)	Tape: 0=LOAD, 1=VERIFY
172–173	($AC–AD)	Tape/Serial: start address for LOAD/SAVE/ VERIFY
183	($B7)	Number of characters in filename (0–187 or 0–16)
187–188	($BB–BC)	Pointer to the current filename
828–1019	($33C–3FB)	Tape buffer area, 192 bytes, for headers and BASIC program data
49164	($C00C)	Keyword dispatch vector table, in token order
50020	($C364)	Miscellaneous messages
57698	($E162)	BASIC VERIFY
57701	($E165)	BASIC LOAD
57809	($E1D1)	Set LOAD, VERIFY, and SAVE parameters

VIA

139–143	($8B–8F)	BASIC RND work area, last random number of initial seed
145	($91)	Keyswitch PIA: bottom keyboard row scan
176	($B0)	Tape: dipole timing adjustment values
177	($B1)	Tape: dipole timing timer 2 difference
181	($B5)	Tape: flag for currently reading data or leader
631–640	($277–280)	Ten-byte keyboard buffer
645	($285)	Serial: timeout enable/disable flag
660	($294)	RS-232 pseudo-6551 command register
665–666	($299–29A)	RS-232 system clock divided by baud rate=microseconds
792–793	($318–319)	Vector to the routine NMI* at 65197 ($FEAD)
37136–37151	($9110–911F)	6522 VIA chip 1
37136	($9110)	Port B I/O register
37137	($9111)	Port A I/O register
37138	($9112)	Data direction register for port B
37139	($9113)	Eight bits, each of which corresponds to the same-numbered bit
37140	($9114)	Timer 1 least significant byte (LSB) of count
37141	($9115)	Timer 1 most significant byte (MSB) of count
37142	($9116)	Timer 1 low (LSB) latch byte

37143	($9117)	Timer 1 high order (MSB) latch byte
37144	($9118)	Timer 2 low order (LSB) counter and LSB latch
37145	($9119)	Timer 2 high order (MSB) counter and MSB latch
37146	($911A)	Shift register for parallel/serial conversion
37147	($911B)	Auxiliary control register
37148	($911C)	Peripheral control register for handshaking
37149	($911D)	Interrupt flag register
37150	($911E)	Interrupt enable register
37151	($911F)	Mirror of port A I/O register at 37137 ($9111)
37152–37167	($9120–912F)	6522 VIA chip 2
37152	($9120)	Port B I/O register
37153	($9121)	Port A I/O register
37154	($9122)	Data direction register for port B
37155	($9123)	Eight bits, each of which corresponds to the same-numbered bit
37156	($9124)	Timer 1 least significant byte (LSB) of count
37157	($9125)	Timer 1 most significant byte (MSB) of count
37158	($9126)	Timer 1 low order (LSB) latch byte
37159	($9127)	Timer 1 high order (MSB) latch byte
37160	($9128)	Timer 2 low order (LSB) counter and LSB latch
37161	($9129)	Timer 2 high order (MSB) counter and MSB latch
37162	($912A)	Shift register for parallel/serial conversion
37163	($912B)	Auxiliary control register
37164	($912C)	Peripheral control register for handshaking
37165	($912D)	Interrupt flag register
37166	($912E)	Interrupt enable register
37167	($912F)	Mirror of port A I/O register at 37153 ($9121)
55341	($D82D)	BASIC WAIT
58528	($E4A0)	Serial: output a 1 on the serial data line
58537	($E4A9)	Serial: output a 0 on the serial data line
58546	($E4B2)	Serial: get an input bit from VIA1 and stabilize
58624	($E500)	Retrieve the address of the I/O memory page
60095	($EABF)	IRQ handler
60190	($EB1E)	Scan the keyboard for keypresses using 6522 VIA2
61125	($EEC5)	Serial: clear attention
61316	($EF84)	Serial: set clock line high
61325	($EF8D)	Serial: set clock line low
61334	($EF96)	Serial: delay one millisecond
61347	($EFA3)	RS–232: send the next bit (NMI continuation routine)
61531	($F05B)	RS–232: prepare to receive the next input byte
61698	($F102)	RS–232: set up NMI for transmission
61718	($F116)	RS–232: open an RS–232 channel for input
62663	($F4C7)	RS–232: open RS–232 device
63284	($F734)	Increment the jiffy clock at 160–162 ($A0–A2)
63659	($F8AB)	Tape: check tape play/rewind/forward button status
63732	($F8F4)	Tape: common tape read/write, start tape operations
64523	($FC0B)	Tape: data write (IRQ driven)
64719	($FCCF)	Tape: restore IRQ vector
64776	($FD08)	Tape: kill motor
64802	($FD22)	Power-on/reset routine (checks for autostart cartridge)
65017	($FDF9)	Initialize the 6522 VIA registers
65193	($FEA9)	NMI handler routine

65234	($FED2)	BREAK interrupt entry
65246	($FEDE)	RS-232: NMI sequences
65372	($FF5C)	RS-232: VIA timer 2 values for baud rate table
65439	($FF9F)	JuMP to 60190 ($EB1E)

VIC chip

1024–4095	($400–FFF)	3072 bytes of expansion RAM area
36864–37135	($9000–910F)	6560 VIC chip
36865	($9001)	Bits 7–0: vertical TV picture origin
36866	($9002)	Number of columns displayed, part of screen map address
36868	($9004)	Raster beam location •
36869	($9005)	Screen map and character map addresses
36870	($9006)	Light pen horizontal screen location
36872	($9008)	Potentiometer X/Paddle X value
36880–37135	($9010–910F)	Future expansion RAM/ROM space
37136–37151	($9110–911F)	6522 VIA chip 1
37159	($9127)	Timer 1 high order (MSB) latch byte
37888–38399	($9400–95FF)	Screen color map (8K+ expanded VIC-20)
58629	($E505)	Retrieve the maximum number of screen columns and lines
58648	($E518)	Initialize the 6550 VIC chip, screen, and related pointers
58819	($E5C3)	Reset the VIC chip registers
60380	($EBDC)	Set keyboard decode table address in 245–246 ($F5–F6)
60705	($ED21)	Used to set uppercase/graphics character set
60900	($EDE4)	Initial values for VIC chip registers
64802	($FD22)	Power-on/reset routine (checks for autostart cartridge)
65234	($FED2)	BREAK interrupt entry

Video matrix

4096–4607	($1000–11FF)	Screen map RAM on VIC-20 with 8K+ expansion

Voice—*see* Sound

WAIT

20–21	($14–15)	Line number integer in two-byte LSB/MSB format
73–74	($49–4A)	Pointer to BASIC variable used in FOR loop
49164	($C00C)	Keyword dispatch vector table, in token order
55195	($D79B)	Obtain number 1–255
55275	($D7EB)	Get two parameters for POKE and WAIT
55341	($D82D)	BASIC WAIT

Warm start

49154	($C002)	Vector to the routine to do the warm start of BASIC
58471	($E467)	Perform a warm start of BASIC
65234	($FED2)	BREAK interrupt entry

6502

0	($0)	6502 JMP opcode 76 ($4C)
1–2	($1–2)	The USR jump vector in LSB/MSB (displacement/page) form
84–86	($54–56)	Jump opcode and vector to function routine
87–96	($57–60)	BASIC numeric work area.
122–123	($7A–7B)	Get-BASIC-character routine
255	($FF)	BASIC temporary area for floating point to ASCII conversion
256–511	($100–1FF)	STACK

780–783	($30C–30F)	The BASIC SYS command uses this area to save and load
780	($30C)	6502 .A register
781	($30D)	6502 .X register
782	($30E)	6502 .Y register
783	($30F)	6502 .P processor status register
37143	($9117)	Timer 1 high order (MSB) latch byte
37145	($9119)	Timer 2 high order (MSB) counter and MSB latch
37148	($911C)	Peripheral control register for handshaking
37149	($911D)	Interrupt flag register
37150	($911E)	Interrupt enable register
37152–37167	($9120–912F)	6522 VIA chip 2
37159	($9127)	Timer 1 high order (MSB) latch byte
37161	($9129)	Timer 2 high order (MSB) counter and MSB latch
37164	($912C)	Peripheral control register for handshaking
37165	($912D)	Interrupt flag register
37166	($912E)	Interrupt enable register
50782	($C65E)	BASIC CLR
58232	($E378)	Perform a cold start of BASIC
58471	($E467)	Perform a warm start of BASIC
58486	($E476)	Program patch area
60095	($EABF)	IRQ handler
64802	($FD22)	Power-on/rest routine (checks for autostart cartridge)
65193	($FEA9)	NMI handler routine
65366	($FF56)	Restore 6502 registers from the stack and return from interrupt
65394	($FF72)	IRQ routine initial 6502 entry point
65526	($FFF6)	Four unused bytes of 255 ($FF)
65530	($FFFA)	6502 vector to 65193 ($FEA9)
65532	($FFFC)	6502 vector to 64802 ($FD22)
65534	($FFFE)	6502 vector to 65394 ($FF72)
Appendix C		Code Chart of 6502 operation codes

If you've enjoyed the articles in this book, you'll find the same style and quality in every monthly issue of **COMPUTE!'s Gazette** for Commodore.

For Fastest Service
Call Our **Toll-Free** US Order Line
800-334-0868
In NC call 919-275-9809

COMPUTE!'s GAZETTE

P.O. Box 5406
Greensboro, NC 27403

My computer is:

☐ Commodore 64   ☐ VIC-20   ☐ Other _____
01   02   03

☐ $20 One Year US Subscription
☐ $36 Two Year US Subscription
☐ $54 Three Year US Subscription

Subscription rates outside the US:

☐ $25 Canada
☐ $45 Air Mail Delivery
☐ $25 International Surface Mail

Name _____

Address _____

City _____ State _____ Zip _____

Country _____

Payment must be in US Funds drawn on a US Bank, International Money Order, or charge card. Your subscription will begin with the next available issue. Please allow 4–6 weeks for delivery of first issue. Subscription prices subject to change at any time.

☐ Payment Enclosed   ☐ VISA
☐ MasterCard   ☐ American Express

Acct. No. _____ Expires _____

COMPUTE! Books

P.O. Box 5406   Greensboro, NC 27403

Ask your retailer for these **COMPUTE! Books**. If he or she
has sold out, order directly from **COMPUTE!**

For Fastest Service
Call Our **TOLL FREE US Order Line**

800-334-0868
In NC call 919-275-9809

Quantity	Title	Price	Total
_____	Machine Language for Beginners	**$14.95***	_____
_____	Home Energy Applications	**$14.95***	_____
_____	COMPUTE!'s First Book of VIC	**$12.95***	_____
_____	COMPUTE!'s Second Book of VIC	**$12.95***	_____
_____	COMPUTE!'s First Book of VIC Games	**$12.95***	_____
_____	COMPUTE!'s First Book of 64	**$12.95***	_____
_____	COMPUTE!'s First Book of Atari	**$12.95***	_____
_____	COMPUTE!'s Second Book of Atari	**$12.95***	_____
_____	COMPUTE!'s First Book of Atari Graphics	**$12.95***	_____
_____	COMPUTE!'s First Book of Atari Games	**$12.95***	_____
_____	Mapping The Atari	**$14.95***	_____
_____	Inside Atari DOS	**$19.95***	_____
_____	The Atari BASIC Sourcebook	**$12.95***	_____
_____	Programmer's Reference Guide for TI-99/4A	**$14.95***	_____
_____	COMPUTE!'s First Book of TI Games	**$12.95***	_____
_____	Every Kid's First Book of Robots and Computers	**$ 4.95†**	_____
_____	The Beginner's Guide to Buying A Personal Computer	**$ 3.95†**	_____

* Add $2 shipping and handling. Outside US add $5 air mail; $2
surface mail.
† Add $1 shipping and handling. Outside US add $5 air mail; $2
surface mail.

**Please add shipping and handling for each book
ordered.**   _____

Total enclosed or to be charged.   _____

All orders must be prepaid (money order, check, or charge). All
payments must be in US funds. NC residents add 4% sales tax.
☐ Payment enclosed   Please charge my: ☐ VISA   ☐ MasterCard
☐ American Express . Acc't. No. _____ Expires _____ / ____

Name _____

Address _____

City _____ State _____ Zip _____

Country _____
Allow 4-5 weeks for delivery.

Mapping the VIC is a detailed, comprehensive explana-
tion of the entire memory within the VIC-20 com-
puter. Whether you're programming in BASIC or
machine language, you'll find this an invaluable sourcebook.
Memory locations that you can alter are discussed clearly and
completely, giving you far greater access to your VIC than you
ever had before. You'll even find details here of the BASIC
and operating system routines stored in permanent ROM
memory. This is *the* definitive work on the VIC-20's memory.

This isn't simply a listing of the computer's memory locations,
however. It's also an introduction to the VIC's internal
architecture and memory use. Best of all, it will show you how
to use the computer's powerful, but hidden, features.
Programming hints and tips throughout the book will give you
insights into the computer's capabilities, helping you write
better and faster running programs in BASIC or machine
language.

Mapping the VIC includes such things as:

- An explanation of all Kernal and BASIC subroutines
- Example programs for many of the most useful operat-
  ing routines
- How to create custom characters
- Details on scrolling the screen, bit mapping, and using
  variables
- A program that converts decimal numbers to
  hexidecimal values
- A character set that turns your screen into a 40-column
  display
- An extensive cross reference to memory location labels,
  as well as a detailed subject index

If you're programming in BASIC, you'll return to this book
again and again for instruction, information, advice, and
clarification. If you're already programming in machine lan-
guage, you'll find this book an indispensable guide to the
computer's memory locations and routine. No matter what
your programming experience, *Mapping the VIC* will be an
often-used addition to your resource library.

COMPUTE!'S
MAPPING THE VIC

COMPUTE
Books