MAPPING THE COMMODORE 128



Ottis R. Cowper

A comprehensive memory guide for BASIC and machine language programmers.

A **COMPUTE!** Books Publication

$19.95

MAPPING THE COMMODORE 128

Ottis R. Cowper

COMPUTE! Publications, Inc
Part or ABC Consumer Magazines. Inc.
One of the ABC Publishing Companies
Greensboro, North Carolina

Copyright 1986^ COMPUTE! Publications, Inc. All rights reserved.

Reproduction or translation of any part of this work beyond that permitted by Sections 107 and 108 of the United States Copyright Act without the permission of the copyright owner is unlawful.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-87455-060-2

The author and publisher have made every effort in the preparation of this book to insure the accuracy of the programs and information However, the information and programs in this book are sold without warranty, either express or implied. Neither the author nor COMPUTE! Publications, Inc. will be liable for any damages caused or alleged to be caused direct!)', indirectly, incidentally, or consequentially by the programs or information in this book.

The opinions expressed in this book are solely those of the author and are not necessarily those of COMPUTE! Publications, Inc.

COMPUTE! Publications, Inc., Post Office Boy 5406, Greensboro, NC 27403, (919) 275-9809, is part of ABC Consumer Magazines, Inc., one of the ABC Publishing Companies, and is not associated with any manufacturer of personal computers. Commodore 64 and Commodore 128 are trademarks of Commodore Electronics Limited,

Contents

Preface v
Introduction vii

1. Memory Organization 1
2. Common Working Storage Area 13
3. Bank 0 Working Storage Area 115
4. RAM Usage 181
5. BASIC ROM 195
6. Machine Language Monitor ROM 241
7. Screen Editor ROM 267
8. I/O Chip Registers, Color RAM, and Character ROM 331
9. Kernal ROM 509

Appendices 617
A. Interrupts/Todrf *Heimarck* 619
B. Bugs and Quirks in 128 ROM 625
C. Character, Screen, and Keyboard Codes 629
D. Musical Note Frequencies 645
E. 64/128 Memory Map Cross Reference 649
P. BASIC Keyword Index 657
G. Index of Locations and Routines 663

Preface

The random access memory (RAM), read-only memory (ROM), and interface hardware chips in your Commodore 128 are like postal stations with hundreds of thousands of mailboxes, each of which can hold a single character, or byte of information. This book is a map of all of those memory locations, but it's more than just a list of addresses. It's also a thorough discussion of how the locations are used by the computer, and, more importantly, how you can take advantage of this information to write more powerful programs.

Why a mapping book? The 128's BASIC is the most powerful version yet in a Commodore computer. It could be argued that there's little need to get "under the hood" of the 128, since most of the functions that required lots of PEEKs and POKEs and an intimate understanding of internal hardware functioning in earlier models like the Commodore 64 can now be handled by simple BASIC statements on the 128. While it may be true that the 128's advanced BASIC makes programming easier, complete control over all the computer's features belongs only to those who understand the secrets of how the system operates. The purpose of this book is to unlock those secrets. The information is valuable for both beginning BASIC and advanced machine language programmers.

The standard features provided by the 128 are often plain vanilla, giving only the barest hint of the full capabilities of the computer. Would you like to set up a Dvorak keyboard that will work with almost any program? See the discussion of the keyboard table pointers in Chapter 2. How about an 80-column X 50-line screen display on your RGB monitor? The explanation of the VDC chip registers in Chapter 8 explains the necessary steps. Do you want to learn how the computer sends data over the serial bus? The process is described in Chapter 9. In fact, you'll find here the answers to most of your questions about the 128. And these answers are written in understandable, clear prose.

This book is the result of painstaking disassembly and deciphering of the Commodore 128 ROMs—a task that required gallons of midnight oil. Commodore's BASIC and operating

system are now nearly ten years old. The ROM routines have many twists and turns where various Commodore programmers have made additions and enhancements along the way. Although the 128 is internally quite different from the Commodore 64, there are similarities. As a result, several previous COMPUTE! books for the 64 provided invaluable assistance in attempts to understand some of the intricacies. I'm particularly indebted to Sheldon Leemon for *Mapping the Commodore 64,* and to Dan Heeb for his two volumes of *Commodore 64 and V1C-20 Tool Kit: BASIC* and *Kernal.*

Every effort has been made to insure that the information provided here is accurate, but in a project of this size and scope it is inevitable that some errors will creep in. Please send any corrections you may discover to the attention of the Book Editor at COMPUTE! Publications in Greensboro. You can also send electronic mail messages concerning this book to CompuServe user ID 73317,1143 or to BIX (Byte Information Exchange) user name ottis.

I'd like to salute my wife Gail for moral and logistical support far above and beyond the call of duty. I'd also like to thank the COMPUTE! staff for patience shown when this project dragged on months longer than anticipated. Finally, I'd like to dedicate this book to George and George, departing and arriving as the work took shape.

Introduction

This memory map is a guide to the way a Commodore 128 in 128 mode uses and manipulates its RAM and ROM. No attempt is made here to provide detailed coverage of the 128's 64 mode. A Commodore 128 in 64 mode doesn't just emulate a Commodore 64; for all practical purposes it *is* a 64, with completely separate Kernal and BASIC ROM. The memory map of the Commodore 64 mode (and its BASIC 2.0) is covered in complete detail in COMPUTERS *Mapping the Commodore 64.* However, Appendix E discusses those 128 features available in 64 mode, and provides a cross reference of important memory locations for 64 and 128 modes—information that will be useful in translating Commodore 64 machine language routines for use in 128 mode.

Nor does this book make any attempt to map the way the 128's CP/M mode uses memory. CP/M is a large and complex operating system, and a CP/M mode memory map would easily fill another entire volume. Moreover, the major portion of CP/M is loaded from disk instead of being permanently stored in ROM. As a result, CP/M is subject to more frequent modification; so far, in the short life of the 128, there have been at least three major revisions. Detailed technical information on Commodore 128 CP/M is available in the book *CP/M Plus User's Guide /Programmer's Guide/System Guide,* available directly from Commodore.

Because this book is intended as a reference for intermediate to advanced BASIC and machine language programmers, no attempt is made to provide simple explanations of all the concepts discussed. The discussions assume familiarity with elementary computer concepts such as bits and bytes, and with memory quantity units such as a page (256 bytes) or a K (kilobyte, 1024 bytes). The book also assumes familiarity with the binary and hexadecimal numbering systems, although decimal equivalents are usually provided.

Hexadecimal numbers in the text are always preceded by a dollar sign ($), the standard 8502 nomenclature for hex. Decimal numbers appear without any prefix. When you see a pair of numbers separated by a slash (/), the first number is

decimal and the second is hexadecimal, unless otherwise indicated. This book uses the machine language monitor's convention of preceding binary numbers with a percent (%) sign. For example, %11 indicates the binary value equivalent to decimal 3, not decimal 11.

When you see numbers mentioned in this book, it should be obvious from the context whether the number refers to an address or a value. Where there could have been confusion, the terms *value* and *location* or *address* specify what is meant. In keeping with common practice, only two hexadecimal digits are generally used when discussing addresses in the first page of memory (zero page). That is, addresses 0-255 are usually written as $00-$FF. Four hexadecimal digits are used for all other addresses. For example, location 256 will be written as $0100,

By nature, the computer prefers to deal with whole numbers and doesn't handle fractions easily. Floating point is the method used to manipulate whole and fractional decimal numbers in 128 BASIC. Floating point also enables very large numbers to be handled in only a few bytes. All mathematical operations in BASIC are performed in floating point. (When you specify integer variables in a mathematical operation, the integer value is converted to floating point for the operation; then the result is reconverted to integer format.) However, because floating point is a rather complex subject, it is not explained in detail in this book even though it is mentioned extensively in Chapter 5. If you are interested in the inner workings of floating point, refer to the excellent discussion of the topic in *COMPUTED VIC-20 and Commodore 64 Tool Kit: BASIC,* by Dan Heeb. Although not written specifically for the 128, all the information about floating point applies to BASIC 7.0 as well.

Several terms used freely in this book need clarification. Most locations discussed in Chapters 2 and 3 are either *pointers, vectors,* or *flags.* Pointers and vectors refer to a pair of memory locations that hold an address. Two-byte address values in pointers and vectors are stored in low-byte/high-byte order. That is, the least significant byte of the address should be stored in the first byte of the pointer or vector, and the most significant byte of the address in the second pointer or vector byte.

The difference between pointers and vectors is that a pointer (as the name implies) points to an address from which

data is to be retrieved or in which data is to be stored, whereas a vector points to the address of a routine to be executed.

A flag is a memory location in which individual bits are used to signal particular conditions. A binary bit can have one of two conditions, %0 or %1 (also referred to as clear and set, respectively). The term comes from the analogy of flags, like those on rural mailboxes, that can be either lowered or raised (there's no half-mast in binary). An example is the active screen flag, location 215/$D7. Bit 7 of the location is clear (%0) when the 40-column display is active, or set (%1) when the 80-column display is active. (You'll find that flag locations often use bit 7 because that bit can be tested very easily in machine language with the BMI and BPL instructions.)

Chapter 1 provides a brief introduction to the way the 128 arranges and manages its memory resources. That chapter and Chapter 4 are the only chapters in the book intended to be read from beginning to end. The remaining chapters describe the use or function of various areas of memory and should be used as an encyclopedic reference. The chapters generally cover memory in ascending address order, starting with zero page in Chapter 2 and ending with the Kernal jump table at the very top of memory in Chapter 9. Each entry in Chapters 2-9 consists of the decimal and hexadecimal address of the location or routine; a label, if one is commonly used; a short statement of the function of the location or routine; and a short description of how the location or routine is used.

Memory Organization

The memory arrangement of a Commodore 128 in 128 mode is much different and more complex than that of any of its Commodore predecessors. As a result, it's necessary to understand how the 128 organizes and manages its memory resources before beginning a detailed examination of how those resources are used. Of the computer's three possible personalities, 128 mode is the default. Unless you take some other action—holding down the Commodore key, inserting a Commodore 64 cartridge, placing a CP/M boot disk in the drive—the computer comes up in 128 mode when you turn it on. As the native mode of the system, 128 mode makes the most complete use of the available memory resources.

You might be interested to learn that, while 128 mode is the default operating mode, the computer always starts out in CP/M mode. When you first turn on power, the Z80 microprocessor has control before the 8502 is allowed to take over. There are only a few signs of this: two short routines are copied into bank 0 RAM. One, at 65488/$FFD0, is an 8502 ML routine that surrenders control to the Z80; the other, at 65504/$FFEO, is a Z80 ML routine that surrenders control to the 8502. There are no routines in any of the 128 mode ROMs to perform this initialization. However, once the Z80 completes its initialization sequence, it turns the system over to the 8502 and 128 mode, and does not go back to CP/M mode unless a CP/M disk is booted.

128 Mode

The 128 mode configuration includes 128K of random access memory (RAM) in two 64K blocks, a 28K BASIC interpreter in read only memory (ROM), a 4K machine language monitor in ROM, 4K of screen editor routines in ROM, 8K of Kernal operating system routines in ROM, a 4K character pattern ROM, and 4K of address space for hardware chip registers (with two separate IK banks of color RAM). The design also provides for

up to 32K of additional ROM internally and up to 32K of ROM on cartridge. The operating system can support two additional 64K banks of RAM, although the 128's design makes no provision for adding memory chips. In sum, that's 373K of address space, as illustrated in Figure 1-1.

This entire 373K address space must be manipulated by the 8502 microprocessor that is the brains of the 128, but an 8502 can directly access only 64K of memory at a time. So how is 128 mode even possible?

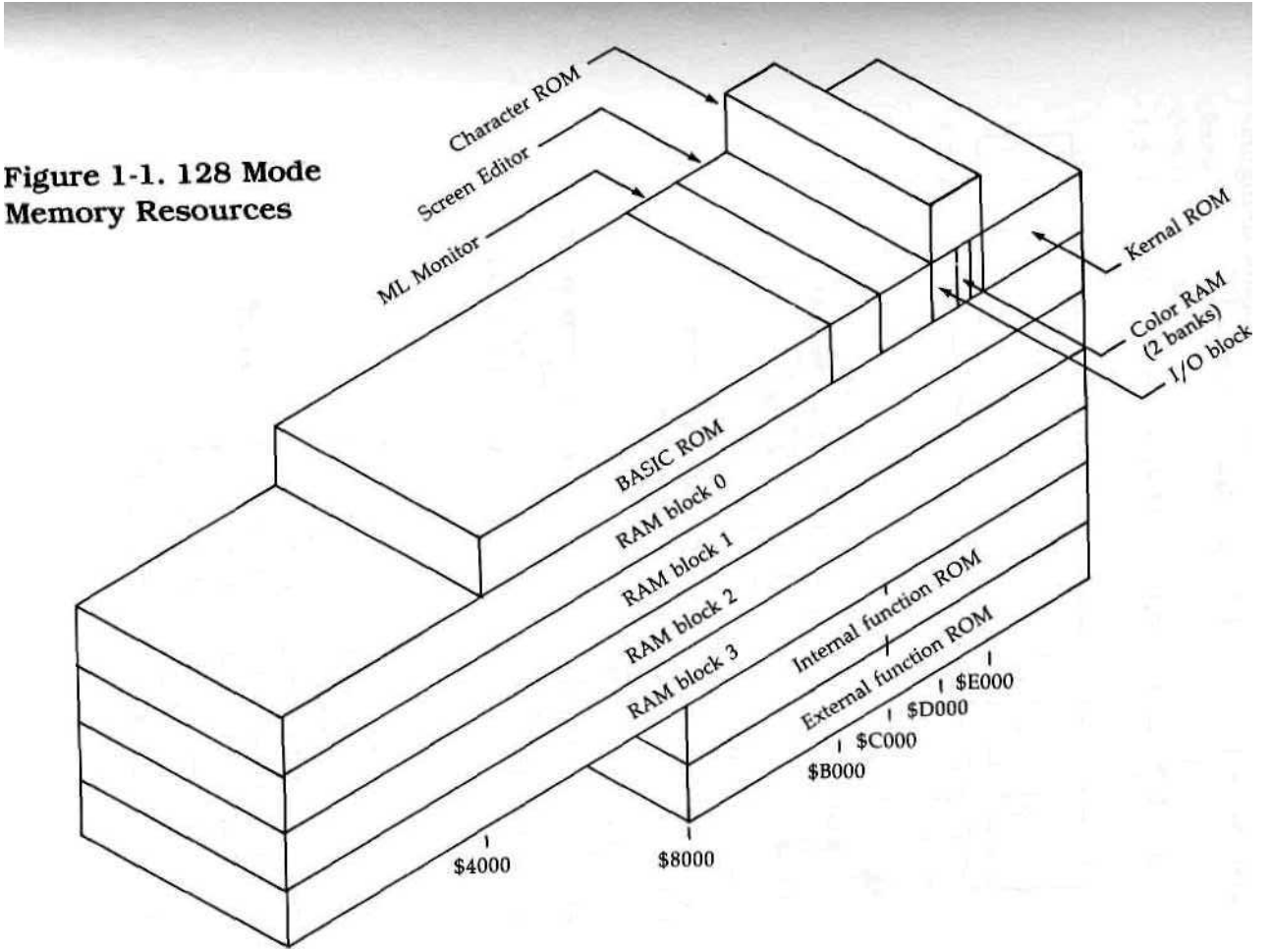
Memory Management Unit

The key is the MMU (Memory Management Unit), a special chip designed by Commodore's engineers to control the memory elements that are visible to the processor. The MMU is assisted by a companion device, the PLA (Programmable Logic Array). The PLA accepts a variety of system timing and control signals and combines them in various ways to create new control signals, taking the place of many separate discrete logic gates. Together, these chips assemble a 64K assortment of RAM, ROM, and I/O chips for the microprocessor to manipulate. The MMU is described in detail in Chapter 8, but the central feature of its memory control system is the configuration register. The value stored in this register, or in a related preconfiguration register, determines what elements the processor sees where. Only the 64K of memory elements defined by the MMU is available to the processor at any given time. Figure 1-2 illustrates the defined function of each bit in the register.

Since the configuration register is a standard eight-bit location, it can hold 256 different values (0-255/$00-$FF); thus, there are theoretically 256 possible configurations of memory resources in a Commodore 128. Fortunately, not all of the possibilities are equally useful, so you don't have to concern yourself with learning them all. The designers of the 128 operating system selected 16 of the most useful arrangements and defined them as banks.

Banks are a central feature of the 128, Banks are not fixed physical arrangements of RAM and ROM. Instead, the 128's banks are illusions created by the MMU to allow the microprocessor to manipulate much more memory than would otherwise be possible. There's nothing particularly sacred about the defined banks—you are free to create your own custom

Figure 1-1. 128 Mode Memory Resources



configurations (see the discussion of the MMU in Chapter 8 for details)—but it is usually more convenient to work in one of the predefined banks. Table 1-1 shows the bank configurations defined by the 128's operating system.

Figure 1-2. MMU Configuration Register

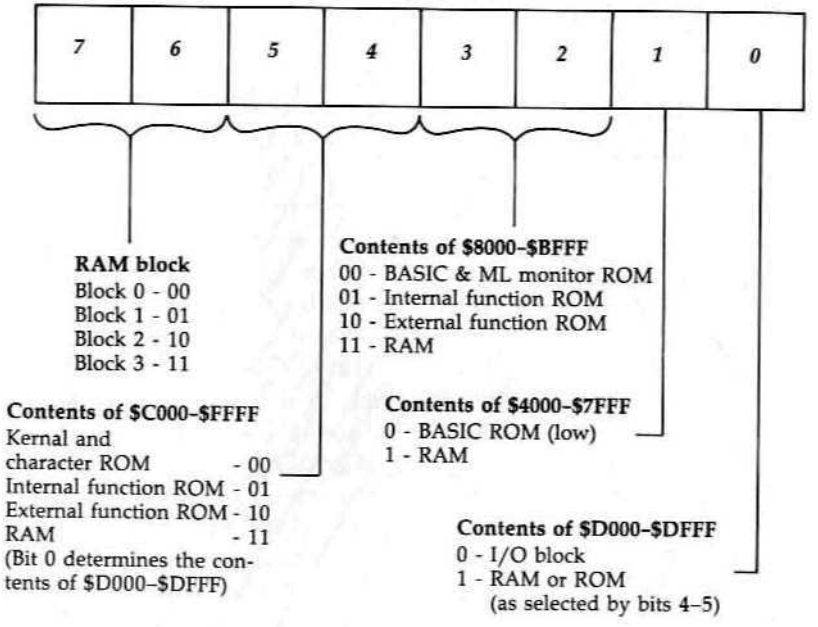


Table 1-1. Standard Bank Configurations

Bank	Configuration Register Setting	Addresses	Contents
0/$00	63/$3F	$0000-$FFFF	RAM from block 0
i/$oi	127/$7F	$0000-$03FF	RAM from block 0
		$0400-$FFFF	RAM from block 1
2/$02	191/$BF	$0000-$03FF	RAM from block 0
		$0400-$FFFF	RAM from block 2
3/$03	255/SFF	$0000-$03FF	RAM from block 0
		$0400-$FFFF	RAM from block 3
4/$04	22/$16	$0000-$7FFF	RAM from block 0
		$8000-$CFFF	Internal function ROM
		$D000-$DFFF	I/O block
		$E000-$FFFF	Internal function ROM
5/$05	86/$56	$0000-$03FF	RAM from block 0
		$0400-$7FFF	RAM from block 1
		$8000-$CFFF	Internal function ROM
		$D000-$DFFF	I/O block
		$E000-$FFFF	Internal function ROM
6/$ 06	150/$96	$0000-$03FF	RAM from block 0
		$0400-$7FFF	RAM from block 2
		$8000-$CFFF	Internal function ROM
		$D00O-$DFFF	I/O block
		$EO00-$FFFF	Internal function ROM
7/$07	214/$D6	$0000-$03FF	RAM from block 0
		$0400-$7FFF	RAM from block 3
		$8000-$CFFF	Internal function ROM
		$D0O0-$DFFF	I/O block
		$E000-$FFFF	Internal function ROM
8/$08	42/$2A	$0000-$7FFF	RAM from block 0
		$8000-$CFFF	External function ROM
		$D000-$DFFF	I/O block
		$E000-$FFFF	External function ROM
9/$09	106/$6A	$0000~$03FF	RAM from block 0
		$0400-$7FFF	RAM from block 1
		$8000-$CFFF	External function ROM
		$D000-$DFFF	I/O block
		$EO00-$FFFF	External function ROM

Bank	Configuration Register Setting	Addresses	Contents
10/$0A	170/$AA	$0000-$03FF	RAM from block 0
		$0400-$7FFF	RAM from block 2
		$8000-$CFFF	External function ROM
		$D0O0-$DFFF	I/O block
		$E000-$FFFF	External function ROM
11/$OB	234/$EA	$0000-$03FF	RAM from block 0
		$0400-$7FFF	RAM from block 3
		$8000-$CFFF	External function ROM
		$D00O-$DFFF	I/O block
		$E000-$FFFF	External function ROM
12/$0C	6/$06	$0000-$7FFF	RAM from block 0
		$8000-$BFFF	Internal function ROM
		$C00O-$CFFF	System ROM (screen editor)
		$DO00~$DFFF	I/O block
		$E00O-$FFFF	System ROM (Kernal)
13/$0D	10/$0A	$0000-$7FFF	RAM from block 0
		$8000-$BFFF	External function ROM
		$CO00-$CFFF	System ROM {screen editor}
		$D0O0-$DFFF	I/O block
		$EO00-$FFFF	System ROM (Kernal)
14/$0E	l/$01	$0000-$3FFF	RAM from block 0
		$4000-$CFFF	System ROM (BASIC 7.0, ML monitor, screen editor)
		$D000-$DFFF	Character ROM
		$E0O0-$FFFF	System ROM (Kernal)
15/$0F	0/$00	$0000-$3FFF	RAM from block 0
		$4000-$CFFF	System ROM (BASIC 7.0, ML monitor, screen editor)
		$D00O-$DFFF	I/O block
		$E00O-$FFFF	System ROM (Kernal)

Exceptions: In all banks, locations $0000 and $0001 are the 8502 processor's on-chip I/O port direction and data registers, and locations $FFOO-$FF04 are MMU configuration and load configuration registers.

This banking system would be too unwieldy to be usable were it not for another capability of the MMU. Notice in the table that the contents of addresses 2-1023/$0002-$03FF are the same in all banks—RAM from block 0. (This particular feature is controlled by the MMU's RAM configuration register rather than by the configuration register.) The common area of RAM is another key to the operation of the 128. Since the area is visible to all banks, a collection of machine language subroutines is copied here from Kernal ROM when the system is initialized. These common subroutines, along with the fact that the MMU makes itself visible in every bank, allow routines in one bank to retrieve, store, and compare data in any other bank; to call subroutines in another bank; or to jump directly to routines in other banks. See the INDFET, INDSTA, INDCMP, JSRFAR, and JMPFAR entries in Chapter 2.

Actually, the operating system's banking scheme promises more than the 128 is able to deliver at this time. Of the four 64K blocks of RAM in the general operating system specification, only two (blocks 0 and 1) are present in the current version of the 128. The operating system was designed to leave open a gateway to future enhanced versions (perhaps a Commodore 256). The circuit board doesn't provide for the addition of RAM chips to populate blocks 2 and 3, nor does the current version of the MMU actually support them (bit 7 of the configuration register has no effect). Thus, banks 2, 3, 6, 7, 10, and 11 can be dismissed outright. If you try to access block 2 RAM (banks 2, 6, or 10), what you'll see is block 0 RAM, so banks 0 and 2, 4 and 6, and 8 and 10 are identical. An attempt to access block 3 will show block 1, so banks 1 and 3, 5 and 7, and 9 and 11 are also identical.

You should be aware that connecting one of the Commodore memory expansion modules (the 1700 for 128K or the 1750 for 512K) won't fill in these missing blocks of RAM. Memory in the expansion modules isn't connected directly to the computer's address lines. Instead, it must be accessed indirectly via the RAM Expansion Controller (REC) chip in the module. See Chapter 8 for more information about the REC and memory expansion modules. Memory in the expansion modules is also arranged in banks, but you shouldn't confuse these with the internal RAM blocks.

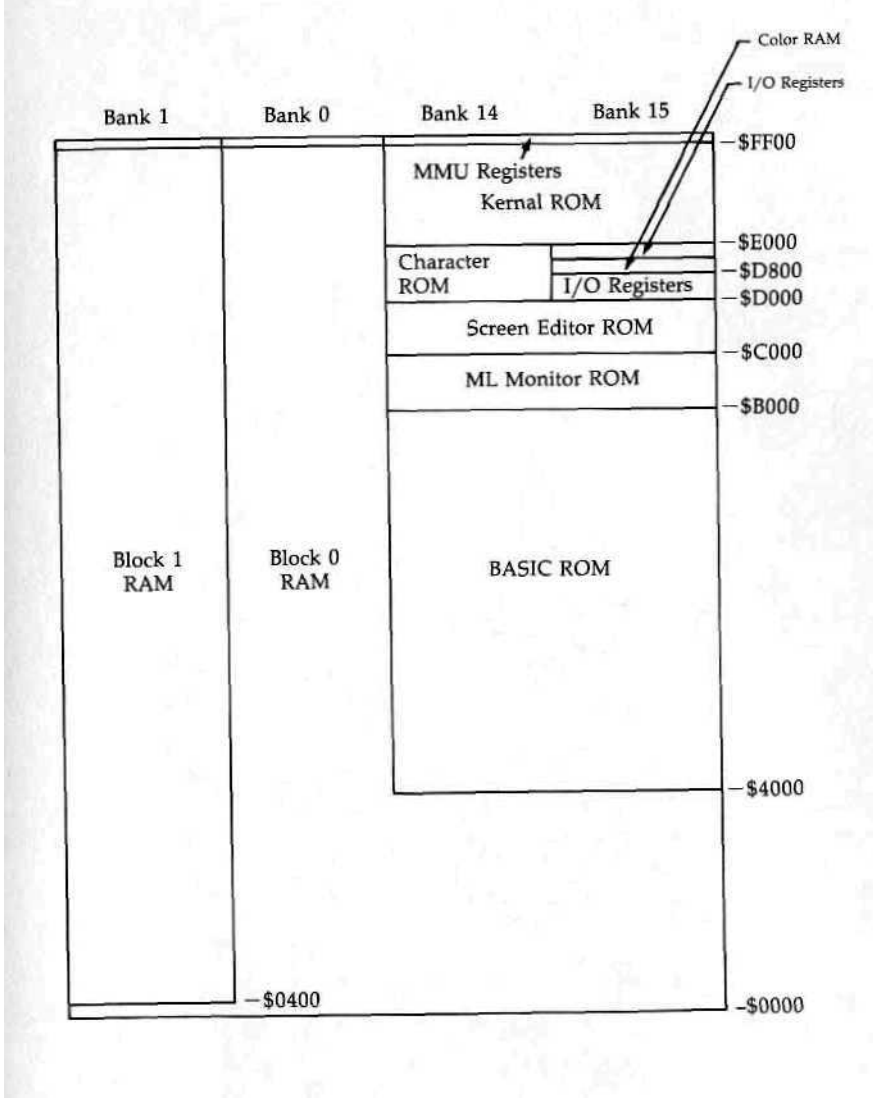
Banks 4, 5, and 12 are useful only if you have a function ROM chip installed in the free socket on the circuit board. Banks

8, 9, and 13 are useful only if you have a 128 ROM cartridge (called an external function ROM) plugged into the expansion port. If you attempt to access one of these ROM areas with no ROM chip installed, you'll get only random, unpredictable data. Since both internal and external function ROMs for the 128 are relatively rare, you can ignore those banks as well, unless you are writing a program specifically to put into ROM.

That leaves only four standard bank configurations which are generally useful: 0, 1, 14, and 15. Figure 1-3 shows the contents of these banks. All the memory areas mapped in this book appear in one or more of these banks. The lower IK of block 0 RAM is the heavily used common area of RAM which appears in every block. It's covered in Chapter 2. The next 7K of block 0 (1024-7167/$0400-$lBFF) is used as working storage by a variety of Kernal and BASIC routines. This area, visible in banks 0, 14, and 15, is covered in Chapter 3. Other RAM usage (banks 0 and 1} is discussed in Chapter 4. Chapter 5 covers BASIC ROM, visible in banks 14 and 15. Chapters 6 and 7 cover the machine language monitor and screen editor, respectively—both also visible in banks 14 and 15. Chapter 8 covers two of the possibilities for addresses 53248-57343/ $D000-$DFFF: the I/O block (including VIC-II chip color RAM) and character pattern ROM. Chapter 9 covers the Kernal ROM seen in banks 14 and 15.

There is one memory selection function not controlled by the MMU. The 128 has two separate IK banks of color RAM, both seen at the same addresses, 55296-56319/$D800-$DBFF in the I/O block. Bits 0 and 1 in the 8502 processor's on-chip data I/O port (location l/$01) determine which block will be seen by the processor and by the VIC chip. See the entry for location l/$01 in Chapter 2 for more information.

Figure 1-3. Normal Bank Configurations



Common Working Storage Area

The 128's memory management hardware has the ability to create common areas of memory—areas where the same memory will be seen regardless of the bank configuration. The system allows up to 16K at both the top and bottom of the processor's address space to be made common. However, the operating system uses only part of this capability, setting up a IK common area at the bottom of memory, locations 0-1023/ $0000-$03FF. No matter what bank configuration you choose, the same block 0 RAM will be seen at these locations. It is this common area, and especially the common routines in page 2, that makes the 128's bank-switching operating system possible.

Zero Page: BASIC and Kernal Working Storage

0-255/$00-$FF

The first 256 memory locations—collectively known as zero page—are special in any computer based on a 6502-family microprocessor like the 128's 8502. The processor has several special addressing modes which use this area. The zero-page addressing modes not only require less memory (two bytes per instruction instead of three); they also execute faster. As a result, system ROM routines make extensive use of these modes. Nearly every address in this page of memory is used by one or more system ROM routines. In fact, you'll notice in the entries for this page that a number of locations have multiple functions, and some have multiple entries.

One of the biggest challenges for machine language programmers is finding sufficient free space in zero page for their programs. Only four locations in the entire page (251-254/ $FB-$FE) are completely unused by any system routine. Most of the locations in the range 10-143/$OA-$8F are used only by BASIC, not by the Kernal. Thus, many of those locations are free for machine language programs that do not require

BASIC. You should be aware that any value stored in zero page will be wiped out during a reset. The RAMTAS routine [$E093], part of the reset sequence, clears locations 2-255/ $02-$FF to zero. (You can prevent this by holding down the RUN/STOP key during the reset which will cause the RAMTAS step of the reset sequence to be skipped. In this case, the system will be left in the machine language monitor after the reset rather than in BASIC.)

Unlike other Commodore computers, the 128 has the ability to make the 8502 see zero page anywhere in memory. The MMU (memory management unit) chip has a feature which allows the processor to exchange zero page with another page so that references to zero page are directed to the alternate page, and references to addresses in the alternate page are directed to zero page. See the discussion of the MMU in Chapter 8 for details. The 128 does not normally make use of this feature; the default position for zero page is at the true zero-page locations.

The first two addresses in this page have a special function. The 8502 processor has a built-in I/O port, and it sees the registers for that port at locations 0-l/$00-$01. References to those addresses always affect the port; the processor will never see the first two bytes of RAM. These locations are not affected by the page-swapping feature. Regardless of where the remainder of zero page is currently seen, locations 0-1 are used exclusively to control the internal port.

0 $00 D8502

Data direction register for processor's on-chip I/O port

Bits 0-6 in this location control the direction of data flow for the seven I/O (input/output) lines on the 8502 microprocessor chip, labeled P0-P6. Setting a bit to %0 makes the corresponding line an input, and its state can be read at the corresponding bit position in location $01. Setting a bit to %1 makes the corresponding line an output, and its state will be controlled by the setting of the corresponding bit position in location $01. The value here is initialized to 47/$2F by the IOINIT routine [$E109], part of both the reset and RUN/STOP-RESTORE sequences. This sets lines 0-3 and 5 for output and lines 4 and 6 for input. Since only seven lines are provided, bit 7 is not used. That bit will retain whatever value is written to it, but its setting has no effect.

1 $01 R85O2

Data register for processor's on-chip I/O port

Each of the seven I/O lines on the 8502 microprocessor has a corresponding bit in this location (bit 7 is unused). The direction of data flow on the lines is controlled by location $00. If an I/O port line is set for input, the corresponding bit here will reflect the state of the input line: %0 if the line is low (0 volts), or %1 if the line is high (+ 5 volts). While a line is set for input, values written to the corresponding bit have no effect. If an I/O port line is set for output, its state will be controlled by the corresponding bit in this location. Storing a %0 in the bit forces the output line to a low (0 volts) state, while storing a %1 in the bit sets the line to a high (+ 5v state).

The I/O lines are connected as follows:

Bits 0-1: The lines connected to these bits control which of the two IK blocks of color memory will be visible at 55296-56319/ $D800-$DBFF when the I/O block is selected. For this purpose, the lines should always be configured as outputs. Unlike in the Commodore 64, these bits have no effect on whether RAM or ROM is selected at a given address. In the 128, memory management is the domain of the MMU chip. See Chapter 8 for more information.

Bit 0 controls which block the processor sees, while bit 1 controls which block the VIC chip sees. Setting either bit to %0 selects block 0, while a setting of %1 selects block 1. The setting of these bits is established during the screen-setup portion of the screen IRQ routine [$C194], That routine sets both bits to %1 for text mode (GRAPHIC 0), or for the text portion of the split-screen modes (GRAPHIC 2 or GRAPHIC 4). For the bitmapped modes (GRAPHIC 1 or GRAPHIC 3) or for the bitmapped portion of the split-screen modes, bit 1 is set to %0. Thus, the VIC sees different blocks of memory for the modes, and drawing on the bitmapped screen will not disturb colors on the text screen. To manipulate these bits in other ways, the screen-setup portion of the IRQ routine must be disabled. Refer to the discussion of the color memory area in Chapter 8 for details on switching color blocks.

Bit 2: The line for this bit, known as the CHAREN line, determines whether the VIC chip will see character ROM in its current video bank. For proper functioning, the line should be configured as an output. While this bit is %0, the VIC chip

will see character ROM beginning at an offset of 4096/$1000 from the start of the bank. The uppercase/graphics set will appear to occupy locations with offsets of 4096-6143/ $1000-$17FF, and the lowercase/uppercase set will appear at offsets of 6144-8191/$1800-$lFFF. The character sets will be visible in all VIC video banks, not just banks 0 and 2 as was the case in the Commodore 64. Only the VIC chip will see the character ROM at these addresses; the processor will still see the locations as RAM or system ROM, depending on the address and bank configuration.

To disable this feature and allow the VIC chip to see RAM at the character set image addresses, the CHAREN bit must be set to %1. However, this cannot normally be done directly because this bit has a shadow at location 217/$D9. During the text mode-setup portion of the screen editor IRQ routine [$C194], the value of bit 2 of the shadow location is copied into this bit. Thus, to change this bit you should set bit 2 of the shadow location instead. If the screen-setup portion of the IRQ routine is disabled (by storing the value 255/$FF in location 216/$D8, for example), the setting of this bit can then be changed directly. The IRQ routine always sets this bit to %1 for bitmapped screen modes or for the bitmapped portion of split-screen modes.

Bit 3: The line for this bit is connected to the CASS WRT (cassette write) line of the cassette port. The setting of this bit determines whether a signal is being written to the tape. For this purpose, the line must be configured as an output. See Chapter 9 for more information about the tape routines.

Bit 4: The line for this bit is connected to the CASS SENSE (cassette button sense) line of the cassette port. If the port line is configured as an input, this bit can be read to determine whether any buttons are currently pressed on the Datassette. When no buttons are pressed (or when no Datassette is connected to the port), this bit will be %1. Pressing any button will change this bit to %0. Unfortunately, the bit merely detects whether buttons are pressed, and cannot indicate which specific buttons. If you press FAST FORWARD when instructed to press PLAY, the 128 won't notice the difference.

Bit 5: The line for this bit controls the CASS MTR (cassette motor) line of the cassette port. When this bit is %1, the power supply to the cassette motor, provided via the CASS

MTR line, is turned off. Setting this bit to %0 turns on the 9-volt power supply to the motor. The setting of this bit is controlled by a shadow location, the cassette motor interlock at 192/SCO.

Bit 6: The line for this bit is connected to the CAPS LOCK key on the keyboard. The line should be configured as an input to read the state of this key. The bit will return a %1 while the key is in the up position (CAPS LOCK off), and a %0 when the key is down (CAPS LOCK on). The status of this bit is read by the SCNKEY routine [$C55D] during each system IRQ, and bit 4 of location 211/$D3 will be assigned the opposite setting of this bit.

Bit 7: There is no I/O port line connected to this bit, so the value here is meaningless. The bit always returns a %0 when read.

2 $02 BANK

Target bank for JMPFAR and JSRFAR

The value here determines the bank to which the JMPFAR routine [S02E3] will jump. Because the JSRFAR routine [$02CD] calls JMPFAR as a subroutine, the value here also determines the destination bank for a JSRFAR. This location should be loaded with the number (0-15) of the desired bank before either JMPFAR or JSRFAR is used.

The BASIC SYS statement is implemented using JSRFAR. In that case, the value here is set from the value in location 981/$03D5, which holds the parameter from the most recent BANK statement (15/$0F by default). The BASIC routine that searches for a token in the runtime stack [$4FAA] also uses location 2/$02 for temporary storage.

When the monitor is entered at the break entry point [$B003], this location is loaded with the bank number in which the system was operating when the BRK opcode was encountered. When the monitor is entered at the cold-start entry point [$B000], as by the BASIC MONITOR command, this location is initialized to 15/$0F (for bank 15). The monitor R command displays the value in this location as the first hexadecimal digit of the PC value. The register change (;) command can be used to alter the value stored here. The value determines the bank for the monitor G (go to routine) and J (jump to subroutine) commands, which use JMPFAR and JSRFAR, respectively.

3-4 $03-$04 PC

Target address for JMPFAR and JSRFAR

The values here determine the address to which the JMPFAR routine [$02E3] will jump. Because the JSRFAR routine [$02CD] calls JMPFAR as a subroutine, the value here also determines the destination address for a JSRFAR. These locations should be loaded with the desired address before either JMPFAR or JSRFAR is used. Contrary to the normal order of address bytes, the high byte of the target address should be stored in location 3/$03 and the low byte in location 4/$04.

When the monitor is entered at the break entry point [$3003], these locations are loaded with the program counter contents stored on the stack when the BRK opcode was encountered. Because of the way the microprocessor handles BRK, this value will be two bytes beyond the address of the BRK ($00) opcode. When the monitor is entered at the cold-start entry point [$B000], as by the BASIC MONITOR command, these locations are initialized to 45056/$B000 (the cold-start entry address). The monitor R command displays the value in these locations as the four rightmost hexadecimal digits of the PC value. The register change {;) command can be used to alter the value stored here. The value determines the target address for the monitor G (go to routine) and J (jump to subroutine) commands, which use JMPFAR and JSRFAR, respectively.

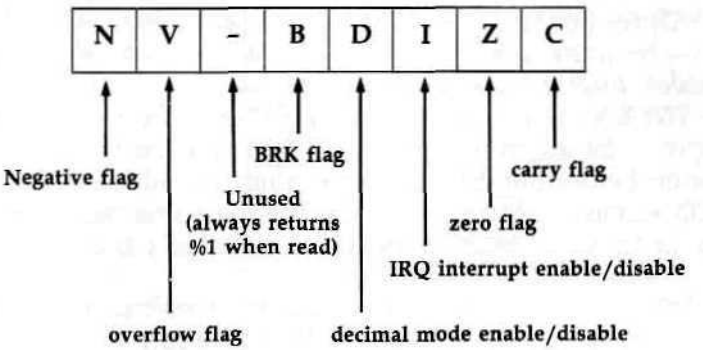
5 $05 S-REG

Status register storage for JMPFAR and JSRFAR

The value in this location is transferred to the processor's status register when a routine is called with JMPFAR [$02E3]. Because JSRFAR [$02CD] also uses JMPFAR, the value here will also determine the initial status register value for a routine called with JSRFAR. You can use this location to set up particular entry conditions for the target routine. For example, certain system routines behave differently depending on whether the carry bit, bit 0 of the status register, is clear (%0) or set (%1) when the routine is called. You can specify the entry setting of the carry bit by setting bit 0 of this location. Figure 2-1 shows the function of the various status register bits. If you don't need any special entry conditions, it's best to set this location to 0/$00.

The contents of the status register upon return from the target routine are stored in this location before return from JSRFAR, so you can read this location to determine the exit status of the routine. This is useful because system routines often use status register bits, particularly carry, to return information about the success of the operation performed by the routine.

Figure 2-1. 8502 Processor Status Register



The BASIC 7.0 version of the SYS statement allows you to specify a status register value, which will be placed in this location before the JSRFAR to the specified address. The RREG statement can be used to read the value here. (The status register value returned by RREG is actually the contents of this location.)

When the monitor is entered at the break entry point [$B003], this location is loaded with the status register contents stored on the stack when the BRK opcode was encountered. When the monitor is entered at the cold-start entry point [$BO00], as by the BASIC MONITOR command, this location is initialized to zero. The monitor R command displays the value in this location under the heading SR. The register change (;) command can be used to alter the value stored here. The value determines the status register contents for the monitor G (go to routine) and J (jump to subroutine) commands, which use JMPFAR and JSRFAR, respectively.

6 $06 A_REG

Accumulator storage for JMPFAR and JSRFAR

The value in this location is transferred to the processor's accumulator {A register) when a routine is called with JMPFAR [$02E3J. Because JSRFAR [$02CD] also uses JMPFAR, the value here will also determine the initial accumulator value for a routine called with JSRFAR. You can use this location to set up a particular entry value for the target routine. The contents of the accumulator upon return from the target routine are stored in this location before return from JSRFAR, so you can read this location to determine the exit accumulator value. The JSRFAR routine itself uses the accumulator after return from the target routine, so you must look to this location for the accumulator value from the target routine.

The BASIC 7.0 version of the SYS statement allows you to specify an accumulator value, which will be placed in this location before the JSRFAR to the specified address. The RREG statement can be used to read the value here. (The accumulator value returned by RREG is actually the contents of this location.)

When the monitor is entered at the break entry point [$B003], this location is loaded with the accumulator contents stored on the stack by the IRQ/BRK handler [$FF17]. When the monitor is entered at the cold-start entry point [$B000], as by the BASIC MONITOR command, this location is initialized to zero. The monitor R command displays the value in this location under the heading AC. The register change {;) command can be used to alter the value stored here. The value determines the accumulator contents for the monitor G (go to routine) and J (jump to subroutine) commands, which use JMPFAR and JSRFAR, respectively.

7 $07 X_REG

X register storage for JMPFAR and JSRFAR

The value in this location is transferred to the processor's X register when a routine is called with JMPFAR [S02E3J. Because JSRFAR [$02CD] also uses JMPFAR, the value here will also determine the initial X register value for a routine called with JSRFAR, You can use this location to set up a particular entry value for the target routine. The contents of the X register upon return from the target routine are stored in this location before return from JSRFAR, so you can read this location

to determine the exit X register value. The JSRFAR routine itself uses the X register after return from the target routine, so you must look to this location for the X register value from the target routine.

The BASIC 7.0 version of the SYS statement allows you to specify an X register value, which will be placed in this location before the JSRFAR to the specified address. The RREG statement can be used to read the value here. (The X register value returned by RREG is actually the contents of this location.)

When the monitor is entered at the break entry point [$B003], this location is loaded with the X register contents stored on the stack by the IRQ/BRK handler [$FF17]. When the monitor is entered at the cold-start entry point [$B000], as by the BASIC MONITOR command, this location is initialized to zero. The monitor R command displays the value in this location under the heading XR. The register change {;) command can be used to alter the value stored here. The value determines the X register contents for the monitor G (go to routine) and J (jump to subroutine) commands, which use JMPFAR and JSRFAR, respectively.

8 $08 Y_REG

Y register storage for JMPFAR and JSRFAR

The value in this location is transferred to the processor's Y register when a routine is called with JMPFAR [$02E3]. Because JSRFAR [$02CD] also uses JMPFAR, the value here will also determine the initial Y register value for a routine called with JSRFAR. You can use this location to set up a particular entry value for the target routine. The contents of the Y register upon return from the target routine are stored in this location before return from JSRFAR, so you can read this location to determine the exit Y register value.

The BASIC 7.0 version of the SYS statement allows you to specify a Y register value, which will be placed in this location before the JSRFAR to the specified address. The RREG statement can be used to read the value here. (The Y register value returned by RREG is actually the contents of this location.)

When the monitor is entered at the break entry point [$B003], this location is loaded with the Y register contents stored on the stack by the IRQ/BRK handler [$FF17]. When

the monitor is entered at the cold-start entry point [$B000], as by the BASIC MONITOR command, this location is initialized to zero. The monitor R command displays the value in this location under the heading YR. The register change (;) command can be used to alter the value stored here. The value determines the Y register contents for the monitor G (go to routine) and J (jump to subroutine) commands, which use JMPFAR and JSRFAR, respectively.

9 $09 STKPTR

Stack pointer storage for JSRFAR and monitor

This location is used in the JSRFAR routine [$02CD] to record the value in the stack pointer upon return from the target routine. The value here doesn't affect the setting of the stack pointer; it merely records the exit value.

When the monitor is entered via either the cold-start entry point [$B000] or the break entry point [$B003], the current stack pointer value is stored in this location. The monitor R command displays the value in this location under the heading SP. The register change (;) command can be used to alter the value stored here. The value here is restored to the microprocessor's stack pointer before the JMPFAR in the G (go to routine) command routine. This location will hold the stack pointer value after a J (jump to subroutine) command, since that routine uses JSRFAR.

9 $09 CHARAC or INTEGR

Working storage for various routines

This location is used for several different purposes by a variety of BASIC routines. It serves as temporary storage in the routine which interprets ASCII characters as numeric values [$50A0]. It holds the value of the desired search character in the routine which searches for a particular character in a BASIC program line [$52A2], and in the routine that puts a string into the string pool [$869A]. It holds the low byte of the integer value generated in the BASIC INT routine [$8CFB]. It is also used for temporary storage of intermediate values while performing BASIC AND or OR operations [$4C86],

10 $0A ENDCHR

Working storage for various routines

This location is used for several different purposes by a variety of BASIC routines. It serves as a counter of the number of digits in the ASCII representation of a number during the routine which interprets the characters as a numeric value [$50A0]. It holds the value of the character which terminates the search in the routine which looks for a particular character in a BASIC program line [$52A2], and in the one that puts a string into the string pool [$869A]. It is also used for temporary storage of intermediate values while performing BASIC AND or OR operations [$4C86].

11 $0B TRMPOS

Current screen column for TAB and SPC calculations

The value in this location is used during the portion of the BASIC PRINT routine [$5554] that handles the TAB and SPC functions. In the computation of the target column for the TAB or SPC, this location will hold the current cursor column value.

12 $0C VERCK

BASIC LOAD/VERIFY flag

The same routine is used to perform both the load and verify operations, so this flag indicates which is being performed, A zero value here indicates a load operation, and a nonzero value indicates verify. The value here is set during the LOAD/VERIFY [$9129] and DLOAD/DVERIFY [$A1A4] routines. Both operations use the Kernal LOAD routine [$F265], which has its own load/verify flag at location 147/$93.

13 $OD COUNT

Working storage for various routines

This location is used for different purposes by several BASIC routines. It holds the most recently found token during program tokenization [$430A]. In the routine that adds or deletes BASIC program lines [$4DE2], this location holds the length of the current line. It is also used as a counter in the RREG routine [$50BD], and as a counter in the subroutines that find or create array-variable elements.

14 $0E DIMFLG

Array dimension flag

This location is used during the routines that create array variables to indicate whether the routines are being called to as the result of a DIM statement. For a DIM statement, this location will contain a nonzero value; otherwise it will be set to 0/$00. This flag is used in testing for the REDIM'D ARRAY ERROR condition.

15 $OF VALTYP

Variable type flag

This location is used to indicate the type of variable currently being evaluated. A value of 0/$00 indicates that the variable is numeric. A nonzero value indicates that the variable is string type. During the routine that finds or creates a variable [$7AAF], this location is set to 0/$00 if the variable is numeric type, or to 255/$FF if it is string type.

16 $10 INTFLG

Numeric variable type flag

If the variable currently being evaluated is numeric (see the entry for location 15/$0F above), bit 7 of this location will be used to indicate the numeric type. If that bit is %0, the variable is standard (floating point) type. If the bit is %1, the variable is integer type. During the routine that finds or creates a variable [$7AAF], this location will be set to 0/$00 for floating-point variables or 128/$80 for integer variables.

17 $11 GARBFL

Working storage for various routines

This location is used for different purposes in several BASIC routines. During string evaluation, it is used as a garbage-collection flag. A zero value indicates that no garbage collection has been performed, while a nonzero value (1/S01) indicates that garbage collection has taken place. The location is also used as a quote mode flag during LIST; a value of 0/$00 indicates that quote mode is off, while a nonzero value (l/$01) indicates that quote mode is in effect. In addition, this location is used as temporary storage for the high byte of the disk status variable during the evaluation of the reserved variable DS.

18 $12 SUBFLG

Integer/subscript prohibit flag

This location is used during the routine to find or create a variable [$7AAF] to specify whether integer or subscripted (array) variables are allowed. While the value here is zero, the variable being evaluated can be of any type. The FOR and DEF routines store the value 128/$80 here. For FOR, this prevents the use of integer or array variables as loop indexes. For DEF, this restricts the function definition to floating point variables and also prevents the parentheses in the function definition from being interpreted as indicating an array variable. This location is reset to zero after each variable is evaluated, and also during CLR [$51F8].

19 $13 INPFLG

Input source flag

BASIC uses a common input handling routine [$56B2] for READ, GET (including GETKEY and GET#), and INPUT (including INPUT#). This location is used to indicate which operation is being performed. The value here will be 152/$98 for a READ operation, 64/$40 for a GET, or 0/$00 for an INPUT.

20 $14 TANSGN

Comparison type flag

Tangent sign flag

The value in this location is used during the string and number comparison routine [$4CB6] to specify the type of comparison being performed, A value here of 1 indicates greater than (>), 2 indicates equal (—), and 4 indicates less than (<). The values are cumulative, so a test for greater than or equal (> —) would result in a value here of 3 (1 + 2). This location is also used during the TAN function routine [$9459] to indicate the sign of the resulting value.

21 $15 CHANNL

Logical file number for BASIC input and output

The value in this location specifies the logical file from which BASIC will receive input and to which BASIC will direct output. The default value is 0/$00, which indicates input from the keyboard and output to the screen. (Logical file 0 is reserved for the system's use; you cannot open logical file 0.)

Statements which get input or send output to other devices, such as GET#, INPUT#, and PRINT*, will temporarily change the value here to the channel number specified in the statement.

The CMD statement can also be used to change the value here and direct all output to a specified logical file. However, you can't depend on CMD (or POKEing a value here) to keep all output flowing to the specified logical file. A number of other BASIC statements reset the value here to 0/$00 each time they are executed, restoring default input and output devices. These statements include GET (and GET# and GETKEY), INPUT#, and PRINT*.

22-23 $16-$17 LINNUM

Integer value of ASCII digit string

These are very busy locations, since the routine which reads ASCII characters from program text and converts the result to a two-byte line number value [$50A0] stores its results here. Other routines which manipulate program lines, such as the one which adds or deletes program lines, will use these locations to hold the line number. Any statement which reads a line number, including GOTO, GOSUB, LIST, and so on, will expect to find the target line number in these locations. The TRAP destination line number is held here during the ERROR routine [$4D3C], and the COLLISION target line number is held here during the GONE routine [$4A9F].

Machine language programmers can store line number values in these locations, then jump into a BASIC routine at a point beyond the line number evaluation step. For example, a machine language program can enter a BASIC program at any line number by jumping into the GOTO routine with the target line number in these locations. The following section of code performs the equivalent of GOTO 100:

```
LDA  #$64    ;Place line number in $16-$17.
STA  $16
LDA  #$00
STA  $17
LDA  #$0F    ;Bank number for BASIC ROM (15).
STA  $02
LDA  #$59    ;Enter GOTO routine at $59FB.
STA  $03
LDA  #$FB
STA  $04
JMP  $02E3   ;Use JMPFAR to call routine.
```

24 $18 TEMPPT

Pointer into temporary string descriptor stack
The value in this location points to the next available slot in the temporary string descriptor stack at 27-35/$lB-$23. This location can have the following values:

Value	Meaning
27/$lB	no entries (stack empty)
3O/$1E	one entry
33/$21	two entries
36/$24	three entries (stack full)

25-26 $19-$1A LASTPT

Pointer to most recent descriptor stack entry
These locations hold the address of the most recent entry in the temporary string descriptor stack at 27-35/$lB-$23. Location 25/$19 will hold the equivalent of the value in 24/$18 less three, and location 26/$lA will hold zero (it is assigned this value during the BASIC cold-start sequence). For example, when there are two entries on the stack, 24/$18 will hold $21, while these two locations will hold $1E and $00, corresponding to address $001E, the address of the second entry in the stack.

27-35 $lB-$23 TEMPST

Temporary string descriptor stack
The three 3-byte entries here hold descriptors (length plus a 2-byte pointer to the starting address of the string in the string pool) for strings being evaluated or assembled. For strings being assigned to variables, the descriptor value generated here will be transferred to the variable table entry for that string,

36-37 $24-$25 INDEX

Multipurpose address pointer
These locations are used as an address pointer by several BASIC routines, including the one at 927/$039F, which retrieves characters from bank 0 (BASIC program text), and the one at 951/$03B7, which retrieves characters from bank 1 (BASIC string storage). Numerous BASIC routines call those character retrieval routines, including the one which inserts or deletes program lines [$4DE2] and the one which updates variable tags while making space for a new variable. The

pointer is also used in the LIST routine to read characters from the keyword table, and in the floating-point routines to copy floating values to and from the variable storage area in bank 1. In addition, location 36/$24 is used for temporary storage during formula evaluation, and location 37/$25 is used as a pointer into the ROM keyword tables when tokenizing program lines [$43E2] or listing (detokenizing) program lines [$5123].

38-39 $26-$27 INDEX2

Multipurpose address pointer

These locations are used as an address pointer by the routine at 960/$03C0 which fetches characters from BASIC program text in bank 0. That routine is called by several other BASIC routines, including the one which adds or deletes program lines. These locations are also used by the ERROR routine [$4D3C] as a pointer to the specified error message in the message table in ROM.

40-44 $28-$2C RESHO

Temporary storage area for multiplication and division

This area is used to hold intermediate values during the BASIC routines that perform floating-point multiplication and division.

45-46 $2D-$2E TXTTAB

Start-of-BASIC-program pointer

The value in these locations points to the first address block 0 RAM used for BASIC program text. The value here is initialized to 7169/S1C01 during the BASIC cold-start sequence. In the Commodore 64, the value here was initialized to the value in the Kernal MEMSTR pointer, the bottom of memory established during the Kernal reset sequence. However, the 128 always initializes the same value here, without regard for the value in MEMSTR (2565-2566/$0A05-$0A06).

The only Kernal routines that change the value here are the ones that allocate or de-allocate a bitmapped graphics area for the GRAPHIC statement. When a bitmapped graphics area is allocated, BASIC program text is moved upward to start at 16385/$4001, above the bitmapped graphics area at 7168-16383/$1COO-$3FFF. In this case, the values in these pointers will be adjusted accordingly. The value here will be reset to

7169/$1CO1 when the graphics area is de-allocated and the BASIC program text is moved back down to its original position.

During the NEW and RUN routines, the CHRGET pointer (61-62/$3D-$3E) is initialized with a value one less than the address in these locations. You can store new values in these locations to change the starting position of BASIC program text—for example, if you wish to reserve free memory space in block 0 RAM below the program. However, two other steps are required to properly initialize the system to use the new starting position: You must also store the value 0/$00 in the location immediately before the address specified here (BASIC requires that program text be preceded by a zero byte), and you must perform a NEW to reset other pointers to reflect the new start-of-BASIC position.

During execution of BASIC'S SAVE and DSAVE routines, the value here determines the starting address of the data to be saved.

47-48 $2F-$3O VARTAB

Start-of-variables pointer

The value in these locations points to the first address in block 1 RAM used for scalar (nonarray) variable storage. The value here is initialized to 1024/$0400 during the BASIC cold-start sequence, and no other system routine changes that setting. You can store new values in these locations to change the starting position of the variable table—for example, if you wish to reserve free memory space for data storage in block 1 RAM below the variables. However, to properly initialize the system to use the the new starting position, you must perform a CLR to reset other pointers to reflect the new start-of-variables position. During the CLR routine [$51F8] {which is also performed during NEW and BASIC cold start), the start-of-arrays pointer (49-50/$31-$32) and the end-of-arrays pointer (51-51/$33-$34) are also set to the value in these locations.

49-50 $31-$32 ARYTAB

Start-of-arrays pointer

The value in these locations points to the first address in block 1 RAM used for the storage of array variables, which is also one location above the last address used for array variables. The value here is initialized to the start-of-variables value in

locations 47-48/$2F-$30 during the CLR routine [$51F8] (which is also performed during NEW and BASIC cold start).

51 -52 $33-$34 STREND

Start-of-free-memory pointer

The value in these locations points to the lowest address in block 1 RAM available for the storage of strings, which is also one location above the last address used for array variables. The value here is initialized to the start-of-variables value in locations 47-48/$2F-$30 during the CLR routine [$51F8] (which is also performed during NEW and BASIC cold start). When the value here equals the value in location 49-50/ $31-$32, no arrays are being used. The function FRE(l) will return the difference between the value here and the one in locations 53-54/$35-$36, representing the remaining amount of memory available for string storage. When the value in 53-54/$35-$36 (the FRETOP pointer) reaches the value here, garbage collection is performed. If garbage collection cannot remove enough unused strings to create free space between the address here and the one pointed to by FRETOP, an OUT OF MEMORY error occurs.

53-54 $35-$36 FRETOP

Bottom-of-string-space pointer

The value in these locations points to the lowest address in block 1 RAM used for the string pool. All character strings used in a BASIC program are stored in the area of block 1 between the address pointed to in 57-58/$39-$3A and the address pointed to here—an area called the string pool. Each active string here will have a descriptor in the variable array table areas at the bottom of block 1, or in the temporary descriptor stack at 27-35/$lB-$23. The pool may also contain inactive strings that the program is no longer using. The value here is initialized to the top-of-memory value in locations 57-58/$39-$3A during the CLR routine [$51F8] (which is also performed as part of NEW and the BASIC cold-start sequence).

When the value here equals the value in location 57-58/$39-$3A, no strings have yet been used. Strings are added from the top of memory downward. When the value here reaches the value in 51-52/$33-$34, garbage collection is

performed to remove inactive strings. If garbage collection cannot remove enough unused strings to create free space between the address here and the one in 51-52/$33-$34, an OUT OF MEMORY error occurs. The function FRE(l) will return the difference between the value here and the one in locations 51-52/$33-$34, the amount of free memory remaining for string storage.

55-56 $37-$38 FRESPC

Temporary pointer into the string pool

These locations are used by the routines that add strings to the string pool as a pointer to the currently referenced string, and as a pointer to the current string during the garbage collection routines.

57-58 $39-$3A MAX_MEM_1

Top-of-memory pointer

The value in these locations determines the highest address in block 1 RAM available for the string pool. (Actually, the address value here will be one location beyond the highest location used for the string pool.) The string pool is filled downward from the address specified here. The value in locations 53-54/$35-$36 specifies the address of the bottom of the pool. When the value in those locations equals the value here, the pool is empty. The BASIC cold-start routine initializes these locations to 65280/$FFO0, one location beyond the highest contiguous address in block 1 RAM (MMU registers are seen at 65280-65284/$FF0O-$FFO4 in all memory configurations). You can reduce the value here to reserve memory at the top of block 1 for other purposes such as data storage. However, when you change the value here you should also execute a CLR statement [$51F8] to reset the other string pool pointers,

59-60 $3B-S3C CURLIN

Current BASIC line number

These locations hold the line number of the BASIC program line currently being executed. After each program line is executed, the routine which executes BASIC program lines [$4AF3] will load these locations with the number of the next line to be executed. The value here is used by various other

BASIC routines that need to know which line is currently being executed, The value here is stored in locations 4608-4609/$1200-$1201 by the routine that processes STOP or END [$4BCA]. The value stored in those locations will be transferred back here by the CONT routine [$5A60]. The value here will be stored in locations 4617-4618/$1209-$120A when an error is processed by the ERROR routine [$4D3C]. The value in those locations will be transferred back here by the RESUME routine [$5F62],

61-62 $3D-$3E TXTPTR

Pointer for main BASIC character retrieval routine

These locations serve as the pointer into BASIC text for the CHRGET routine, BASIC'S primary character retrieval routine. In earlier Commodore computers, the entire CHRGET routine was in zero page. The 128's CHRGET is located higher in the common area, beginning at address 896/$0380, and only the pointer is kept in zero page. CHRGET is designed to retrieve the next nonspace character of BASIC text, so the first step in CHRGET is to increment the address here. The routine also has an alternate entry point called CHRGOT at 902/$0386, which retrieves the current character (the one at the address here) without incrementing the pointer.

The NEW, RUN, and LOAD routines all call the subroutine [$5254] which initializes this pointer to one byte before the start-of-BASIC value in locations 45-46/$2D-$2E. Because the CHRGET routine is so heavily used, many BASIC routines affect the value here. For example, any of the routines which send the program to another line, such as GOTO, GOSUB, THEN, and so on, must replace the current value here with the address of the target line. The value here is stored in locations 4610-4611/$1202-$1203 by the routine that processes STOP or END [$4BCA]. The value stored in those locations will be transferred back here by the CONT routine [$5A60]. The value here will be stored in locations 4622-4623/$120E-$120F when an error is processed by the ERROR routine [$4D3C], The value in those locations may be transferred back here by the RESUME routine [$5F62].

The value here is also used as a pointer for the alternate character retrieval routine at 969/$03C9, which fetches the current text character without CHRGET's test for character type.

63-64 $3F-$40 FNDPNT

Working pointer for various routines

These locations are used as a working pointer into the runtime stack at 2048-2559/$0800-$09FF by the routines that search for tokens in the stack. The RENUMBER routine [$5AF8] uses these locations as an end-of-program pointer. The PRINT USING routine [$9520] uses the routine at 939/$03AB (which uses these locations as a pointer) to retrieve characters from the template pattern string in block 1 RAM.

65-66 $41-$42 DATLIN

Line number of current DATA statement

These locations hold the line number of the BASIC program line containing the DATA statement from which DATA items are currently being read. These locations are updated by the subroutine that searches for the start of the next DATA statement: [$57CA], called during execution of the READ statement. The value here isn't used by any system routine, but it can be very helpful when you're debugging a program containing DATA statements. Whenever a program stops with an ILLEGAL QUANTITY or TYPE MISMATCH error message in a line containing a READ statement, it's very likely that the error is actually in the DATA line rather than the line specified in the error statement (the one which contains READ). You can find the line number from which the last, possibly erroneous, DATA item was read using PRINT PEEK(65) + 256 * PEEK(66).

67-68 $43-$44 DATPTR

Pointer to next DATA item

These locations are used as a pointer to the address at which the search for the next available DATA item will begin. The subroutine that searches for the next DATA item [$57CA], called during execution of the READ statement, will update the value here to point to the start of the next DATA item. The RESTORE statement, when used without a line number parameter, resets the value here to the starting address of BASIC program text (from locations 45-46/$2D-$2E), That RESTORE subroutine is also called as part of the CLR routine, which in turn is called as part of RUN. Thus, the search for DATA items normally begins at the first program line. The

RESTORE statement can be used with a line number parameter to change the value here. In that case, the pointer value will be reset to the starring address of the specified line. The specified line need not contain a DATA statement. It merely specifies the line from which the search for the next DATA statement will begin.

69-70 $45-$46 INPPTR

Text pointer for input

The common input routine [$56B2], used in the execution of the GET, GETKEY, GET#, INPUT, INPUT*, and READ statements, uses these locations as a pointer to the characters to be read as input. The value here will be transferred into the CHRGET pointer at 61-62/$3D-$3E so that CHRGET can be used to retrieve characters from the input. The GET, GETKEY, and GET* statements will initialize the value here to 513/$0201, an input buffer location set to 0/$00 to cause the input routine to read the next character. The INPUT and INPUT* statements will initialize the value here to 511/$01FF, a location immediately before the input buffer set to 44/$2C, the code for the comma character. The actual input will be in the input buffer beginning at 512/$0200. The READ statement will initialize these locations with the starting address of the next DATA item (from locations 67-68/$43-$44).

71-72 $47-$48 VARNAM

Current variable name

These locations are used during the routine to find or create a variable [$7AAF] to hold the compressed (two-byte) form of the specified variable name. This compressed form will then be used as a search pattern to check whether a variable of the same name and type currently exists. If not, the characters here will be used as the name for the new variable.

73-74 $49-$4A VARPNT

Pointer to variable descriptor

These locations are used as a pointer to the first byte of the descriptor for the variable—the address of the location just beyond the two-character name in the variable table entry for the variable. The value here is set upon exit from the routines to find [$7AAF] or create [$7B90] a variable. The FN (user-

defined function) routine will load these locations with the address of the descriptor for the dummy variable in the function definition,

75-76 $4B-$4C FORPNT

Variable descriptor pointer and working storage
These locations are used during the routine that assigns variable values (LET [$53C6]) as a pointer to the variable value or string descriptor. For numeric variables, the address here will be the location in block 1 RAM where the value will be stored. For string variables, the address here will be the location in block 1 RAM where the length and pointer into the string pool for the string will be stored. The FOR statement uses the value here to find the address of the value for the loop index variable.

For the WAIT statement [$6C2D], location 75/$4B holds the test byte pattern and location 76/$4C holds the mask byte pattern. Location 75/$4B is also used as an index into the current line during the routine to list BASIC program lines [$5123],

77-78 $4D-$4E VARTXT

Temporary storage for text pointer
These locations are used for temporary storage for the CHRGET pointer value from 61-62/$3D-$3E during the common input routine [$56B2], which uses CHRGET to retrieve characters from the input source location. Location 77/$4D is also used during the numeric expression evaluation routine [$77EF] as a flag to indicate when the end of the expression has been reached.

79 $4F OPMASK

Relational operator flag
When the main expression evaluation routine [$77EF] finds a relational operator (<, =, or >) in the current expression, it stores a value here indicating which operator has been found. For greater than (>) operations, the value here will be 1. For equals (=), the value will be 2; for less than (<) it will be 4. When the expression is evaluated, this value will be transferred to location 20/$14.

80-81 $50-$S1 DEFPNT

Defined function pointer and working pointer
These locations are used by the routine that retrieves bytes from the variable table entry for a function definition (FN). That routine [$42CE] uses these locations as a pointer for one of the bank 1 character retrieval subroutines [$03AB]. These locations are also used as a working pointer by one of the routines that reads values during garbage collection. That routine [$42FB] also uses a bank 1 data retrieval subroutine [$03AB]. The routine that allocates the bitmapped graphics area [$9F4F] uses these locations to hold the number of bytes that must be moved upward to make room for the graphics area.

80-84 $50-$54 TEMPF3

Temporary storage for floating-point value
These locations are used to temporarily hold the floating-point value of the exponent during the routine to handle the exponentiation (T) operator [$8FC1].

82-83 $52-$53 DSCPNT

Variable address storage and working pointer
The routine that creates space in the string poo] for a new string variable uses these locations to temporarily store the address of the variable table entry. These locations are also used as a pointer by the routine that retrieves characters from the string poo] for the LEFT$, RIGHT$, and MID$ functions [$42D8].

85 $55 HELPER

HELP flag
Bit 7 of this location is tested in the routine which lists BASIC program lines [$5123] to determine whether the line is being displayed by LIST or by HELP. When the bit is %1, the subroutine at 22956/$59AC will be called to highlight the portion of the line where the most recent error occurred. The HELP statement routine [$5986] sets bit 7 to %1 before it calls the line-listing routine, and clears it to %0 afterwards.

86-88 $56-858 JMPER

BASIC function execution vector
This vector is used to execute the routines that handle BASIC functions. Location *S6/$56* is initialized during the BASIC

cold-start sequence with the value 76/$4C, the 8502 JMP opcode. The function dispatch routine [$4BF7] loads 87-88/$57-$58 with the address of the routine that performs the desired function operation. A JSR $0056 instruction then executes the function-handling routine.

89-93 $59-$5D TEMPF1

Floating-point work area
These locations are used as a temporary floating-point work area during the series evaluation routine [$9086] for the LOG, SIN, COS, TAN, and ATN functions. Location 89/$59 is also used for temporary storage during the routine [$9D7C] which subtracts the contents of one pair of bitmapped graphics storage locations from the contents of another pair of locations.

90-91 $5A-$5B ARRYPNT

Multipurpose working pointer
These locations are used as a pointer to the destination of text being moved in the routine to add new BASIC program lines to memory [$4DE2] and as a pointer into array space when making room for a new variable [$7B90]. They are also used to hold the line link value during RENUMBER [$5AF8].

92-93 $5C-$5D HIGHTR

Multipurpose address pointer
These locations serve as a pointer for the routine that reads the source text being moved in the routine to add new BASIC program lines. This routine [$42DD] uses one of the common bank 0 character retrieval routines [$039F]. The locations serve as a pointer in the routine to read source bytes when creating space for new variables. This routine [$42E2] uses one of the common bank 1 character retrieval routines [$03AB]. The locations are used during the RENUMBER routine [$5AF8] to hold the number of the line currently being renumbered.

93-95 $5D-$5F STR1

String length and pointer for MID$
When MID$ is used as a statement [$5901] (to add characters to a string), these locations hold the descriptor of the original string. Location 93/$5D holds the length, and locations 94-95/$5E-$5F hold the address and are used as a pointer.

94-98 $5E-$62 TEMPF2

Temporary storage for floating-point value
These locations are used to store an intermediate value from floating-point accumulator #1 (FAC1) during the series evaluation routine [$909C] for the EXP function.

94-95 $5E-$5F

Working pointer for garbage collection
These locations are used as a pointer to the tag bytes for the current string during the routine that performs string pool garbage collection [$92EA].

95 $5F DECCNT

Decimal point position
This location is used during the routine [$8E42] that creates a character string representing the value in floating-point accumulator #1 (FAC1) to hold the position within the string for the decimal point. The location is also used as a loop counter in the routine [$7E3E] to calculate the amount of memory needed for an array.

96-98 $60-$62 STR2

Substring length and pointer for MID$
When MID$ is used as a statement [$5901] (to add characters to a string), these locations hold the descriptor of the substring to be added. Location 96/$60 holds the length, and locations 97-98/$61-$62 hold the address and are used as a pointer.

96-104 $60-$68 T0-T2

Monitor zero-page pointers and working storage
These locations are used by many routines in the monitor. The monitor routine [$B7CE] that determines the numeric value of a parameter in the input buffer leaves the value in locations 96-98/$60-$62 (in low- to high-byte order), so any numeric value in a monitor command is at least initially held there. For monitor commands that accept two or more address parameters, the first address is transferred into locations 102-104/$66-$68, and the value there is then used as a working pointer to the byte to be read or written. (The monitor's indirect fetch [$B11A], indirect store [$B12A], and indirect compare [$B13D] routines use 102-103/$66-$67 for the address pointer

and 104/S68 for the bank value.) The starting address is subtracted from the ending address, and the result is transferred to 99-101/$63-$65. The value in those locations is then used as a count of bytes to be affected by the operation. The compare/transfer routine [$B231], which accepts three address parameters, uses 102-104/$66-$68 as the pointer to the source address for the compare or transfer and 96-98/$60-$62 as the pointer to the destination address.

Some monitor routines also make alternate use of some of these locations. The memory display routine [$B152] uses 96-98/$60-$62 as a count of lines to be displayed. During assembly [$B406], 99/$63 holds the length of the current instruction, and location 100/$64 holds the addressing mode type. Locations 99-100/$63-$64 are used to unpack mnemonics during disassembly [$B6A1], and 99/$63 serves as a counter during directory display [$BB03].

97-98 $61-$62 LOWTR

Multipurpose address pointer

A wide variety of BASIC routines use these locations as a pointer. They serve as the pointer for a heavily used routine [$42EC] to read characters from BASIC program text. (That routine uses one of the common bank 0 character retrieval subroutines [$039F].) The routine is called by the routine which adds or deletes program lines [$4DE2], the one which searches for a line number [$5064] (in which case the starting address of the line is returned in these locations), LIST [$50E2], and DELETE [$5E87]. These locations also serve as the pointer for a routine [$4300] to read values from the variable table. (That routine uses one of the common bank 1 character retrieval subroutines [$03AB].) The routine is called by the routine [$7AAF] which searches the variable table to check whether a variable with a specified name already exists, and the one [$7CAB] which performs a similar search for array names. If an existing name is found, the address of the table entry for the variable or array will be returned in these locations.

99-103 $63-$67 FAC1

Floating-point accumulator 1

These locations are the primary work area for all routines which use floating-point math, which includes all of BASIC'S

mathematical functions. Any numerical value used in a BASIC program will be converted to a floating-point value here for further processing. The result of any BASIC operation which performs a numerical calculation will be held in these locations.

Floating-point notation is rather complicated. This method of representing numbers has three components: the mantissa, the base, and the exponent. You may be familiar with BASIC'S scientific notation. For example, the value 73,500 will be represented as 7.35E4, BASIC'S shorthand for 7.35 X 10^4. In this format, the 7.35 is the mantissa, the 10 is the base, and the 4 is the exponent. In BASIC'S internal floating-point format, the base value is 2; location 99/$63 holds the exponent, and locations 100-103/$64-$67 hold the mantissa. The exponent is held in excess-128 format, meaning that 129 has been added to the exponent value. (This is a little trick to avoid having to deal with negative exponents.) To get the true exponent value, subtract 129. Only the lower 31 bits of the four-byte mantissa value are used, and the mantissa is normalized—adjusted so that its value is always effectively in the range 1-1.9999. Thus, the formula for converting the FAC1 contents into a decimal value is:

value = (1 + (mantissa / (2 T 31))) * 2 t (exponent - 129)

The routine which converts the contents of FAC1 into a two-byte integer value will leave the results of the conversion in locations 102-103/$66-$67. Some routines which don't use floating-point math use these locations for other purposes. Locations 102-103/$66-$67 are used as a pointer by the routine [$42E7] that reads values from the variable table. That routine uses one of the common bank 1 character retrieval subroutines [$03AB].

104 $68 FACSGN

Sign of FAC1

Bit 7 of this location is used to indicate the sign of the value in FAC1. The value here will be 0/$00 for positive values in FAC1 and 255/$FF for negative values. As long as the floating-point value is held in the accumulator, this location will be used to indicate its sign. When the floating-point value is stored in a variable, the setting of this bit will be copied to the highest bit of the mantissa. Likewise, when a value is copied from a variable into the accumulator, the setting of bit 7 of the most significant byte of the mantissa is copied here.

105 $69 SGNFLG

Sign flag during conversion
Count of terms in series evaluation

This location is used as a flag during the routine [$8D22] that calculates the floating-point value equivalent of a string of digit characters to indicate whether the string being converted has a leading negative sign. During the series evaluation routine [$909C], this location holds the number of terms in the series.

106-110 $6A-$6E FAC2

Floating-point accumulator 2
These locations are the second floating-point accumulator area, used in those operations that require a second floating-point value. Location 106/$6A is the exponent and locations 107-110/$6B-$6E are the mantissa. All operations that involve both accumulators will transfer the results to FAC1.

111 $6F ARGSGN

Sign of FAC2
Bit 7 of this location indicates the sign of FAC2, just as location 104/$68 does for FAC1.

112 $70 ARISGN

Sign comparison flag
The routines that load values into FAC1 [$8A89] and FAC2 [$8AB4] perform an exclusive-OR of the values in locations 104/S68 and 111/$6F—the signs of the values in the respective floating-point accumulators. Thus, this location will hold 0/$00 if both signs are positive or both are negative, or 255/$FF if the signs are different.

112-113 $70-$71 STRNG1

Multipurpose address pointer
These locations are used as an address pointer by the routine [$42F1] which loads characters from strings in BASIC program text for transfer into the string pool. (That routine uses a common bank 0 character retrieval subroutine [$039F].) The locations are also used as a pointer by the routine [$42F6] that reads characters from the first string in a concatenation operation. (That routine uses a common bank 1 character retrieval subroutine [$03AB].)

113 $71 FACOV

Rounding flag for FAC1

When a pair of floating-point mantissas are adjusted for a math operation, any extra bits that must be shifted out of the smaller mantissa are held here and used to round the final result to extend the accuracy of the operation.

114-115 $72-$73 STRNG2

Multipurpose address pointer and working storage

In the series evaluation routine, these locations are used as a pointer to the constant values used in the series evaluation. In the VAL routine [$804A], these locations are used as a pointer into the character string to be translated into a floating-point value. These locations are used as working storage in the routines that calculate the amount of memory required for an array. In the DEC routine [$8072], these locations are used as a work area for converting the hexadecimal string characters into a two-byte integer value.

116-117 $74-$75 AUTINC

Step value for autoincretnent

These locations hold the step value for automatically incrementing the line number if autoincrement mode is active. After each BASIC program line is entered while this mode is active, the value here will be added to the previous line number and the resulting new line number will be printed on the screen. Autoincrement mode will be active whenever these locations contain a nonzero value. These locations are reset to 0/$00 during the BASIC cold-start sequence, and during the RUN subroutine [$5A81] that resets flags. The value here can be set using the AUTO statement.

118 $76 MVDFLG

Graphics area flag

The value here indicates whether the bitmapped graphics color and screen area has been allocated at 7168-16383/ $1COO-$3FFF, in which case the start of the BASIC program area will have been moved to 16384/$4000. A value of 0/$00 here indicates that no graphics area is allocated, while a nonzero value indicates that the area has been allocated and the BASIC program area has been moved. This location is initial-

ized to 0/$00 (no graphics area) during the BASIC cold-start sequence. When the graphics area is allocated, this location is decremented (to 255/$FF). The only BASIC statement that resets this location is GRAPHIC CLR—the value here isn't affected by NEW or CLR—so once a graphics area is allocated it will remain allocated until the computer is reset or a GRAPHIC CLR statement is executed.

119 $77 Z_P_TEMP_1

General-purpose working storage

This location is used for temporary storage by a variety of BASIC routines.

120 $78 HULP

String offset pointer

This location is used during the routine [$5901] that handles MID$ as a statement to hold the offset from the start of the string to the substring being replaced. It's also used during the PLAY string-processing subroutine [$6DE1] to hold the offset to the next character waiting to be processed in the string.

121 $79 SYNTMP

Multipurpose temporary storage

This location is used for temporary storage by a number of different BASIC routines.

122 $7A MTXTPTR

Index into input buffer for monitor

The monitor uses this location to store the position of the next character to be read from the input buffer (512-672/ $0200-$02A0).

122-124 $7A-$7C DSDESC

Descriptor for disk error string DS$

These locations are used as the descriptor for the disk status string provided by the reserved variable DS$. Location 122/$7A will hold the length of the string, and locations 123-124/$7B-$7C will hold the address of the string. The length value is initialized to 0/$00, effectively emptying the string, during the CLR routine [$51F8], which is also part of NEW and RUN, The routine to generate the error string [$A778] will set the values here whenever the DS or DS$ variables are used.

125-126 $7D-$7E TOS

BASIC runtime stack pointer

These locations are used as the pointer into the BASIC runtime stack at 2048-2559/$0800-$09FF. This stack area is used to hold information for FOR, GOSUB, and DO statements (see Chapter 3 for details). The value here is the address of the next free location in the stack, which is filled from top to bottom—from 2559/$09FF down to 2048/$0800. Unlike the processor's stack with its automatic pointer, the pointer into this stack must be updated explicitly. The pointer value is reset to 2559/S09FF during the CLR routine [$51F8], which is also part of NEW and RUN. Each time an entry is placed on the stack, the pointer value here is decremented by the number of bytes in the stack entry. Whenever an entry is retrieved from the stack, the value is incremented by the number of bytes to be removed.

127 $7F RUNMOD

RUN mode flag

This location is used to indicate the current operating mode of the computer. When the value here is 0/$00, BASIC is in immediate mode. No program is executing; BASIC is waiting for a command or a program line to be entered. When bit 6 is set to %1 (flag value of 64/$40), a program is being loaded for execution (the RUN *"filename"* statement has been used). When bit 7 is set to %1 (flag value 128/$80), a BASIC program is being executed. The value here is reset to 0/$00 during the step of the main BASIC loop that displays the READY prompt. The RUN subroutine [$5A81] to set flag values will set this location to 128/$80, unless the option to load and run a disk file has been used. In that case, the flag will be set to 64/$40 while the file is loading, then to 128/$80 when it begins running.

128 $80 POINT

Decimal point position

This location is used during the PRINT USING routine [$9] to hold the number of digits to be printed before the decimal point.

128-129 $80-$81 PARST

Parameter flags for DOS support commands
The various disk command routines set these locations before calling the routine [$A32F] that processes parameters for disk commands. The values here indicate which parameters are valid for the command being processed. When a bit is %1, the parameter string for the command can include the corresponding element:

Location	Bit	Parameter element
128/$80	0	source filename
	1	destination filename (following TO)
	2	logical file number (#)
	3	device number (U)
	4	source drive number (D)
	5	destination drive number (D following TO)
	6	file type parameters (L or W)
	7	save-with-replace indicator (@)
129/$81	0	bank number <B)
	1	starting address (P)
	2	ending address (P following TO)

130 $82 OLDSTK

Storage for processor stack pointer
This location is used to store the current processor stack pointer value before a BASIC program line is executed [$4AF3]. If an error occurs while the line is being executed, the value here will be restored to the stack pointer during the error-handling routine [$4D3C],

131 $83 COLSEL

Color source for current graphics command
The first parameter in graphics commands such as DRAW, CIRCLE, BOX, and so on, is the color source number. That value is held here after the parameter is evaluated. For the standard bitmapped (GRAPHIC 1) screen, valid values are 0 (background) and 1 (foreground). For multicolor bitmapped (GRAPHIC 3) screens, values of 2 and 3 are also valid to select the additional multicolor sources. If the parameter is omitted, the value here will default to 1 (foreground).

132 $84 MULTICOLOR_1

Color source 2 storage

This location holds the current color number for color source 2. The value here doesn't have any immediate effect on the screen display, but whenever the COLOR statement routine [$69E2] is executed, the lower four bits of the value here will be copied into the lower four bits of the multicolor video matrix fill byte at 995/$03E3. That value will be used to fill the video matrix area when the multicolor bitmapped (GRAPHIC 3) screen is cleared. Thus, the value here eventually determines the color of multicolor bitmapped pixels represented by %10 bit pairs. This is the bit pattern that will be produced for lines drawn when color source 2 is specified. During the BASIC cold-start sequence, this location is initialized to 1 (white). The value here can be changed using the statement COLOR *2,n,* where n is a standard BASIC color number (1-16). Remember that the color change is effective only after the multicolor bitmapped screen is cleared.

133 $85 MULTIC0L0R-2

Color source 3 storage

This location holds the current color number for color source 3. The value here doesn't have any immediate effect on the screen display, but whenever the bitmapped screen is cleared, block 0 of color memory (55296-56319/$D800-$DBFF) is filled with the value here. Thus, the value here eventually determines the color of multicolor bitmapped pixels represented by %11 bit pairs. This is the bit pattern drawn for lines when color source 3 is specified. During the BASIC cold-start sequence, this location is initialized to 2 (red). The value here can be changed using the statement COLOR 3,Ji, where *n* is a standard BASIC color number (1-16). Remember that the color change is effective only after the multicolor bitmapped screen is cleared.

134 $86 FOREGROUND

Current foreground color (source 1) storage

This location holds the current color number for color source 1. The value here doesn't have any immediate effect on the screen display, but whenever the COLOR statement routine [$69E2] is executed, the lower four bits of the value here will be copied into the upper four bits of both the standard video

matrix fill byte at 994/$03E2 and the multicolor video matrix fill byte at 995/$03E3. One of these values, depending on the display mode, will be used to fill the video matrix area when the bitmapped screen is cleared. Thus, the value here eventually determines the color of standard bitmapped pixels represented by %1 bits or of multicolor bitmapped pixels represented by %10 bit pairs. During the BASIC cold-start sequence, this location is initialized to 13/$0D (light green). The value here can be changed using the statement COLOR \,*n*, where *n* is a standard BASIC color number (1-16). Remember that the color change is effective only after the screen is cleared.

135-136 $87-$88 SCALE_X

Horizontal scaling factor

These locations hold the horizontal scaling factor for BASIC graphics routines. If scaling is in effect (indicated when the scaling flag at 4458/$116A holds a nonzero value), the specified horizontal *(x)* coordinates for all graphics routine parameters will be adjusted according to the value here to get the true bitmap coordinates. The value here can be changed using the SCALE statement. If the first parameter in the statement is 1 (scaling on), the factor here is calculated from the second parameter according to the following formula:

scaling factor = (65535 * 320)/scaling parameter

If the parameter is omitted, a default value of 20480/$5000 (for a bitmapped screen) or 10240/S2800 (for a multicolor bitmapped screen) is supplied. This allows a scaled screen of 1024 horizontal positions *(x* coordinates 0-1023).

137-138 $89-$8A SCALE-Y

Vertical scaling factor

These locations hold the vertical scaling factor for BASIC graphics routines. If scaling is in effect (indicated when the scaling flag at 4458/$116A holds a nonzero value), the specified vertical *{y)* coordinates for all graphics routine parameters will be adjusted according to the value here to get the true bitmap coordinates. The value here can be changed using the SCALE statement. If the first parameter in the statement is 1 (scaling on), the factor here is calculated from the third parameter according to the following formula:

scaling factor = (65535 * 200)/scaling parameter

If the parameter is omitted, a default value of 12800/13 200 is supplied. This allows a scaled screen of 1024 vertical positions (*y* coordinates 0-1023).

139 $8B STOPNB

PAINT mode flag

Bit 7 of this location is used during the PAINT statement routine [$61A8] to specify whether the fill stops at pixels where the source color is encountered (indicated when the bit is %0) or whether all nonbackground pixels will be filled (indicated when this bit is %1). This location is normally set according to the fourth parameter of the PAINT statement, to 0/$00 if the parameter is 0 or omitted, or to 128/$80 if the parameter is 1.

140-141 $8C-$8D GRAPNT

Address pointer for graphics routines

These locations are used as an address pointer by several BASIC graphics routines. The value here points to the address within the bitmap where a character pattern will be copied during CHAR [$67D7]. The locations serve as a pointer to the area being filled during the SCNCLR [$6A79]. In the general pixel-drawing routine, these locations point to the bitmap address where the pixel will be drawn.

142-143 $8E-$8F VTEMP

Temporary storage for graphics routines

These locations are used for temporary storage by a variety of BASIC graphics routines.

144 $90 STATUS

Status flag for tape and serial bus operations

This location records the status of the most recent tape or serial bus operation. In general, when the operation has been successful the value here is 0/$00, while a nonzero value indicates that an error has occurred or that the end of the file has been reached. The value here is reset to zero at the beginning of any load or save, or whenever a file is opened to tape or a serial device. Various error conditions are indicated by setting particular bits to %1. The bits are used as follows:

Bit	Bit value	Serial bus	Tape
0	l/$01	timeout during write	
1	2/S02	timeout during read	
2	4/$04		short block (leader read where data expected)
	8/$08		long block (data read where leader expected)
	16/$10	mismatch during verify	unrecoverable read error {or mismatch during verify)
	32/$20		checksum mismatch for block
	64/$40	EOI (end of file)	end-of-file marker read
	128/$80	device not present	end-of-tape marker read

In BASIC, the reserved variable ST returns the value here when the current I/O device is tape (1) or serial (4 or larger). For RS-232 operations, the status is instead recorded in location 2580/S0A14.

145 $91 STKEY

Scan value of STOP key column

This location holds the current status of the keyboard column containing the RUN/STOP key. The Kernal UDTIM routine [$F5F8], which is part of the system jiffy IRQ sequence, includes a section which reads the current column of the keyboard matrix. (See Figure 7-1 in Chapter 7 for a diagram of the keyboard matrix.) The current state of that column is stored in this location (unless the key connected to row 7 of the column has been pressed at the same time as some key in columns 1 or 6, which contain the SHIFT keys). The proper functioning of this routine depends on the fact that the SCNKEY routine [$C55D], normally performed earlier in the IRQ sequence, leaves the system set to scan column 7, the one containing the RUN/STOP key (in row 7). When the Kernal STOP routine [$F66E] is called to determine whether the RUN/STOP key is currently pressed, it checks this location rather than actually reading the keyboard.

This location can also be used to read any of the other keys in column 7. The value here will be 255/$FF when no key in that column is pressed. Pressing a key sets a corresponding bit here to %0. The values here when the respective keys are pressed are as follows:

Key	Bit	Value	Key	Bit	Value
	0	254/$FE	space	4	239/$EF
*	1	253/$FD	Commodore	5	223/$DF
CONTROL	2	251/$FB	Q	6	191/$BF
2	3	247/$F7	RUN/STOP	7	127/$7F

146 $92 SVXT

Tape-timing baseline adjustment factor

This location is used during routines which read from tape to hold a value representing the difference between the actual time required to read a bit from tape and the standard rime. This value is used to adjust other timing constants to compensate for minor variations in tape motor speeds.

147 $93 VERCK

Kernal load/verify flag
Monitor operation flag

The same Kernal routine is used to perform both load and verify operations. This location is used during the routines which read data from tape and disk to specify which operation has been called for. The value in the accumulator upon entry to the Kernal LOAD routine [$F265] will be stored here.

The monitor compare/transfer routine [$B231] uses this location as an operation flag. A value of zero here indicates that a compare operation is being performed, while a value of 128/$80 indicates a transfer operation. The monitor byte-pattern search routine [$B2CE] stores the number of characters in the search buffer here. The monitor load/save/verify setup routine stores the character code of the current command (L, S, or V) here.

148 $94 C3P0

Serial deferred character flag

This location is used to indicate whether a character is waiting in the one-byte character buffer at 149/$95. Bit 7 of this location will be %0 if no character is waiting, or %1 if the buffer contains a byte awaiting transmission.

149 $95 BSOUR

Serial character buffer

This location is used as a buffer for bytes sent over the serial bus. The operating system maintains this buffer so that the last byte of a file can be sent with the EOI handshake to identify it as the final byte. Location 148/$94 is used to indicate whether the current value here represents a character awaiting transmission. It's very important to close serial bus files opened for writing; otherwise, the final byte with the end-of-file handshake won't be sent.

150 $96 SYNO

Cassette block synchronization count

This location is used during routines which read from tape to indicate when the system has read leader bytes and is waiting for the end of the leader segment.

151 $97 XSAV

Temporary register storage

This location is used for temporary storage of the Y register value during the Kernal GETIN subroutine for RS-232, and for temporary storage of the X register value during the Kernal BASIN routine for tape.

152 $98 LDTND

Number of files currently open

This location records the number of active files—the number of files which have been opened but not yet closed. This value also serves as an index to the next available entry in the logical file number, device number, and secondary address tables at 866-895/$0362-$037F. The value here is reset to 0/$00 (no files open) when zero page is cleared during the reset sequence. The Kernal CLALL routine [$F222] will also reset this location to 0/$00. The value here is incremented each time a logical file is opened, and decremented each time one is closed. An attempt to open an additional file when this location contains 10/$0A, indicating that the maximum 10 files are already open, will result in a TOO MANY FILES error.

153 $99 DFLTN

Current input device

The value here specifies the current input device number for the Kernal GETIN and BASIN routines, When a logical file is selected for input by the CHKIN routine, the device number value for the file is read from that file's entry in the device number table at 876-885/$036C-$0376 and stored here. The CLRCH routine will reset the value here to 3/$03, to make the keyboard the default input device.

154 $9A DFLTO

Current output device

The value here specifies the current output device number for the Kernal BSOUT routine. When a logical file is selected for

output by the CKOUT routine, the device number value for the file is read from that file's entry in the device number table at 876-885/$036C-$0376 and stored here. The CLRCH routine will reset the value here to 0/$00, to make the screen the default output device.

155 $9B PRTY

Tape character parity
This location is used during routines which read from tape to calculate the parity of the byte currently being read. Bytes stored on tape have an extra parity bit added to make an odd total number of %1 bits in the combined character (eight data bits plus parity). This location is used to make sure that an odd total number of bits is read back for each character.

156 $9C DPSW

Tape dipole received flag
This location is used when a byte is being read from tape to indicate whether all bits of the byte have been received (indicated by a nonzero value), or whether bits are still being read (indicated by a value of 0/$00).

157 $9D MSGFLG

Kernal message control flag
This location controls whether Kernal messages will be displayed. The Kernal routines have two types of messages: control messages (PRESS PLAY ON TAPE, SEARCHING FOR, and so on) and error messages (I/O ERROR # followed by a number). This location controls which types of messages, if any, will be displayed. When the value here is set to 0/$00, no Kernal messages are displayed. Setting bit 6 to %1 enables error messages, while setting bit 7 to %1 enables control messages. The value here can be set using the Kernal SETMSG routine [$F75C]. The BASIC routine MAIN [$4DB7], which is responsible for the READY prompt, sets this flag to 128/$80 (control messages only), since BASIC provides its own error messages. When the RUN routine is executed to run a program, the value here is reset to 0/$00 (no messages). The monitor changes the setting to 192/$C0 (all messages).

158 $9E PTR1

Tape pass 1 error-log pointer

The Commodore tape system records two copies of each block of data written to tape. If errors are detected while the first copy is being read, the address where the erroneous byte is located is stored in the tape error-log area at the bottom of page 1, This location is used as an offset to the next available two-byte address slot in the error log. The value here is reset to 0/$00 at the beginning of the operation. An unrecoverable error occurs if the value here exceeds 60/$3C, indicating that more than 31 errors have been logged.

This location is also used to hold the offset into the specified filename during the routine which checks to determine whether a particular tape header has been found, and for temporary storage of the type identifier byte when header blocks are being written to tape.

159 $9F PTR2

Tape pass 2 error-log pointer

This location is used during the routine which reads the second copy of each tape data block to indicate the offset to the next slot in the tape error log. That slot will contain the address of the next byte that needs correcting in the second pass. This location is also used to hold the offset into the filename in the tape header when the routine is checking whether a particular tape header has been found.

The monitor assemble routine [$B406] also uses this location to store the position of the next character to be processed from the instruction address buffer (2720-2729/ $OAA0-$0AA9).

160-162 $A0-$A2 TIME

Software jiffy clock

These three bytes comprise the jiffy clock, a counter maintained by the operating system. Location 160/$A0 is the high byte, 161/$A1 the middle byte, and 162/$A2 the low byte. The UDTIM routine [$F5F8], called during each system jiffy IRQ interrupt sequence, will increment this counter 60 times per second. (UDTIM checks and compensates for PAL video

systems, so these locations are incremented 60 times per second regardless of whether interrupts occur at the North American rate of 60 times per second or the European rate of 50 times per second.) Thus, location 162/$A2 will be incremented every 1/60 second; location 161/$A1 every 1/60 * 256 = 4.27 seconds; and location 160/$A0 every 4.27 * 256 = 1092 seconds, or every 18.2 minutes. All three locations (along with the rest of zero page) are reset to 0/$00 during the reset sequence. The UDTIM routine will also reset the locations to 0/$00 if the value here reaches $4F1AO1, corresponding to 24 hours after the start of the count. The Kernal RDTIM routine [$F65E] can be used to read these locations, and the SETTIM routine [$F665] can be used to change the value here. From BASIC, the reserved variables TI and TI$ can be used to read the values here (TI$ converts the value to hours:minutes:seconds format). TI$ can also be used to change the value here.

Although this timer is easy to use, especially from BASIC with TI and TI$, it's not particularly accurate for timekeeping applications. These locations depend on the system IRQ interrupt, which is affected by a number of operations. For example, the system interrupt is turned off during loads and saves to tape or disk, effectively stopping the clock. The more tape or disk operations you perform, the more inaccurate your clock time becomes. If you need more reliable timekeeping, refer to the discussion of the CIA chips' time-of-day clocks in Chapter 8.

163 $A3 PCNTR/R2D2
Tape: Count of bits to be read or written
Serial: EOI flag

When characters are being read from or written to tape, this location is used as a countdown for the number of bits remaining to be received or sent.

When characters are being sent over the serial bus, this location is used to indicate when an EOI (end or identify) handshake should be performed to mark the end of the file. The EOI sequence is added when bit 7 of this location is set to %1.

164 $A4 FIRT/BSOUR1

Tape: Half-cycle indicator
Serial: Byte received

When bits are being read from or written to tape, this location is used to indicate which half-cycle for the bit is currently being received or sent.

When characters are being received over the serial bus, this location is used to assemble received bits into complete bytes.

165 $A5 CNTDN/COUNT

Tape: Leader synchronization countdown
Serial: Count of bits to send / burst mode byte count

During the routines which write blocks of data to tape, this location is used to provide the countdown characters that come at the end of each leader segment. The value here is initialized to 9; it will then be repeatedly written to tape and decremented until the value reaches zero.

When characters are being sent over the serial bus, this location is used as a countdown of bits to be sent. The value here is initialized to 8 for each byte and decremented each time a bit is sent. When bytes are being read from the serial bus, this location is used to indicate whether an EOI handshake has been detected. The value is initialized to 0/$00, then incremented after the first EOI is received. During high-speed burst mode loads, this location is used as a count of the number of bytes read from the current disk sector.

166 $A6 BUFPT

Pointer into cassette buffer

This location is used during the tape BASIN routine to hold the offset to the next character to be read from the cassette buffer. This location is incremented after each character is read from the buffer. When the value here reaches 192/$C0, all characters have been read from the buffer, so another block of data will be read into the buffer (if another is available) and the value here will be reset to 0/$00. During the tape BSOUT routine, this location holds the offset of the next available position in the cassette buffer. This location is incremented each time a character is added to the buffer. The buffer is considered filled when the value here reaches 192/$C0, at which the

block of data will be written to tape and the value here will be reset to 0/$00.

167 $A7 SHCNL/INBIT
Tape: Leader dipole count / block indicator
RS-232: Current bit received

During the routines which write to tape, this location is used as one counter in a timing loop to specify the number of leader dipoles to be written. When reading from tape, this location is used to indicate which block is being read.

When characters are being received over the RS-232 interface, this location holds the most recently received bit.

168 $A8 RER/BITCI
Tape: Half-cycle indicator for writing / error flag for reading
RS-232: Count of bits remaining to be received

When bits are being written to tape, this location is used to indicate which half-cycle of the dipole for the bit is currently being written. When characters are being received from tape, this location is used as a flag to indicate an error in the received byte.

When characters are being received over the RS-232 interface, this location is used as a countdown for the number of bits to be received for the current character. The value here will be initialized from 2581/S0A15 for each character.

169 $A9 REZ/RINONE
Tape: Word marker flag / half-cycle flag
RS-232: Start bit received flag

When characters are being written to tape, this location is used to indicate whether a word marker dipole has yet been written for the current character. When characters are being read from tape, this location is used to indicate whether the next half-cycle should be a long or short one.

When characters are being received over the RS-232 interface, this location is used to indicate whether a start bit has been received yet, A nonzero value here indicates that the system is still waiting for a start bit, while a value of 0/$00 means that a start bit has been received.

170 $AA RDFLG/RIDATA

Tape; Read phase flag
RS-232: Assembly byte for received bits

During the routines which read from tape, this location indicates the current stage of the operation. When the value here is 0/$00, the reading routine is waiting for the synchronization countdown characters to be read. Nonzero values less than 64/$40 indicate that block countdown characters are being read. A value of 64/$40 indicates that the first copy of the data block has been read, while a value of 128/$80 indicates that all characters from the first block have been read and the routine is waiting for the second copy.

When characters are being received over the RS-232 interface, the bits received are shifted into this location until a full byte has been assembled.

171 $AB SHCNH/RIPRTY

Tape: Leader dipole counter / checksum work byte
RS-232: Received byte parity

During the routines which write to tape, this location is used as one counter in a timing loop to specify the number of leader dipoles to be written. During the routines which read from tape, this location is used for computing the checksum for the block being read.

When characters are being received over the RS-232 interface, this location is used to indicate whether an odd or even number of %1 bits have been received, to determine the parity of the received bit.

172-173 $AC-$AD SAL-SAH

Kernal working address pointer

These locations are used as a pointer to the address of the current byte to be written to tape or saved to disk, or the address where the byte read from tape or from a disk boot sector is to be stored. The Kernal has several routines to service this pointer, including one [$ED51] to load this pointer with the operation starting address in 193-194/$C1-$C2, one [$EEC1] to increment the address here, and one [$EEB7] to compare the address here against the operation ending address at 174-175/$AE-$AR There is also a routine [$F7CC] to retrieve the character at the pointer address from the bank specified in 198/$C6,

and one [$F7BC] to store the current accumulator contents at the pointer address in the bank specified in 198/$C6.

172-175 $AC-$AF

Work area for disk booting

The Kernal BOOT-CALL routine [$F890] uses locations 172-173/$AC-$AD to hold the address at which the contents of additional boot sectors are to be stored. Location 174/$AE holds the bank number for the additional data. Location 175/ $AF holds the number of disk sectors to be loaded during the boot process.

174-175 $ AE-$ AF EAL-E AH

Kernal address pointer

This location is used during the routines which read from or write to tape, or in saving to disk, to hold the ending address for the operation. For loading from disk, this location is used as a working pointer to the address where data is stored. After all bytes have been loaded, the locations will hold the ending address. (Actually, in all cases the pointer will hold the address of the location immediately following the last one involved in the operation.) The Kernal SAVF routine [$F53E] initializes these locations with the contents of the X and Y registers when the routine is called. The Kernal provides a routine [$F7C9] to retrieve the character at the pointer address from the bank specified in 198/$C6, and one [$F7BF] to store the current accumulator contents at the pointer address in the bank specified in 198/$C6.

176 $BO CMPO

Tape adjustable baseline compensation factor

This location is used during tape routines to indicate whether the current baseline time (the time allotted for a particular type of dipole) needs to be slightly increased or decreased. This allows the computer to compensate for slight variations in tape speed.

177 $B1 TEMP

Working storage for compensation factor computation

This location is used as a work byte for computing the baseline compensation factor at 176/$B0.

178-179 $B2-$B3 TAPE1

pointer to cassette buffer

These locations hold the starting address of the 192-byte cassette buffer. The value here is initialized to 2816/$0B0O by the Kernal RAMTAS routine, part of the reset sequence. No Kernal routine changes this default setting. The routines that read and write data to tape test these locations to insure that the address is greater than 512/$0200.

180 $B4 SNSW1/BITTS

Tape: leader/data flag

RS-232: Count of bits transmitted

During routines which read from tape, this location is used to indicate whether the routine is currently waiting for the start of a data block (indicated by a value of 0/$00 here) or reading data from a block (indicated by a nonzero value here).

When bytes are being sent over the RS-232 interface, this location holds the count of bits sent for the current character.

181 $B5 DIFF/NXTBIT

Tape: Leader completed flag

RS-232: Next bit to send

During routines which read from tape, this location is used to indicate when the end of a leader segment has been reached. The value here is set to 0/$00 when the word marker at the end of a leader is read.

When bytes are being sent over the RS-232 interface, bit 2 of this location is used to hold the setting of the next bit to be sent.

182 $B6 PDP/RODATA

Tape: Error flag / end of block flag

RS-232: Character being sent

When an error is detected while a character is being read from tape, this location is set to a nonzero value to indicate that the character has not been read successfully. During routines which write to tape, this location is used as a flag to indicate when end-of-block processing should be performed.

When bytes are being sent over the RS-232 interface, this location holds the character being sent. Bits are pulled off one at a time from right to left.

183 $B7 FNLEN

Length of current filename

This location holds the length of the filename for the current I/O operation. The value here can be set using the Kernal SETNAM routine [$F731]. The starting address for the filename is held in locations 187-188/$BB-$BC, and the bank number where the filename is found is held in location 199/$C7.

184 $B8 LA

Logical file number

This location holds the logical file number for the current I/O operation. The value here can be set using the Kernal SETLFS routine [$F738]. When a file is opened, the value here will be transferred into the logical file number table at 866-875/ $0362-$036B.

185 $B9 SA

Current secondary address

This location holds the secondary address for the current I/O operation. The value here can be set using the Kernal SETLFS routine [$F738]. When a file is opened, the value here will be transferred into the secondary address table at 886-895/ $0376-$037F.

186 $BA FA

Current device number

This location holds the device number for the current I/O operation. The value here can be set using the Kernal SETLFS routine [$F738], When a file is opened, the value here will be transferred into the device number table at 876-885/ $036C-$0375.

187-188 $BB-$BC FNADR

Pointer to start of filename

These locations hold the starting address of the filename for the current I/O operation. The value here can be set using the Kernal SETNAM [$F731]. Location 183/$B7 holds the length of the filename, and location 199/$C7 holds the bank number in which the filename is located.

189 $BD OCHAR/ROPRTY

Tape: Byte read from tape / byte to be written to tape
RS-232: Parity calculation working storage
Serial: Current byte during burst mode load

For tape operations, this location holds the byte most recently read, or the byte currently being written.

When bytes are being sent over the RS-232 interface, this location is used to indicate whether an even or odd number of %1 bits have been sent in the current character. This information is used to determine the value of the parity bit if one is to be sent.

During high-speed burst mode loads from disk, this location holds the byte most recently received from the drive.

190 $BE FSBLK

Block count
This location is used during routines which read from or write to tape to specify which of the two images of the current block is currently being read or written.

191 $BF MYCH/DRIVE

Tape: Assembly area for byte being read
Disk: Default drive number for booting

When characters are being read from tape, the bits read are assembled in this area until a complete byte is formed; then the value is transferred to location 189/$BD for evaluation or storage.

During the BOOT_CALL routine [$F890], this location is used to hold the character code for the specified drive number. The contents of the accumulator when the routine is called will be stored here.

192 $CO CAS1

Tape motor interlock
This location is used to control bit 5 of the processor I/O port at location l/$01. The system jiffy IRQ sequence includes a subroutine [$EED0] which tests bit 4 of the processor port to determine whether any Datassette buttons are pressed. If no buttons are pressed, this location is set to 0/$00 and bit 5 of the port is set to %1 to turn off power to the cassette motor. When a button is pressed, this location is checked. If it con-

tains a 0/$00, the port bit is set to turn off the power. Thus, the cassette motor can't be powered when no button is pressed or while this location contains 0/$00. When this location is set to any nonzero value, the setting of the port bit is not affected by the IRQ subroutine, so—as long as a button is pressed—the motor can be turned on and off, changing the setting of the port bit.

193-194 $C1-$C2 STA

Kernal address pointer

These locations are used by the Kernal SAVE routine [$F542] to hold the starting address of the area of memory to be saved to disk or tape, The value is loaded from the zero-page pointer specified in the accumulator upon entry to the routine.

These locations are also used by the Kemal BOOT_CALL routine. Location 193/$C1 holds the track number and location 194/$C2 holds the sector number for the block currently being read from disk.

195-196 $C3-$C4 TMP2/MEMUSS

Kernal address pointer

The contents of the X and Y registers upon entry to the Kemal LOAD routine [$F265] are stored here. If the secondary address that preceded the LOAD was 0/SOO, a relocating load was specified, so this address is used as the starting address for the loaded data.

These locations are also used as a working pointer in the routine [$E1FO] to initialize the soft reset vector,

197 $C5 DATA

Bit read from tape / checksum of block written to tape

During routines which read from tape, this location is used to indicate the value of the bit most recently read. During routines which write to tape, this location is used for working storage of the checksum being calculated for the block.

198 $C6 BA

Bank where data for save, load, or verify is found

This location holds the bank number from which data will be saved by the Kernal SAVE routine or to which data will be loaded or verified by the Kernal LOAD routine. The value

here doesn't affect the current system configuration; it only specifies the bank for load, save, or verify operation data. The Kernal SETBANK routine [$F73F] can be used to set the value here.

199 $C7 FNBANK

Bank where filename for open, save, load, or verify is found
This location holds the bank number in which the filename for the current I/O operation is found. The value here can be set using the Kernal SETBANK routine [$F73F].

200-201 $C8-$C9 RIBUF

Pointer to RS-232 input buffer
The value in these locations determines the starting address of the 256-byte RS-232 input buffer—the area where characters are stored as they are received via the RS-232 interface. The value here is initialized to 3072/S0C00 by the RAMTAS routine [$E093], part of the reset sequence. This places the input buffer at its default position, and no system routine changes this setting.

202-203 $CA-$CB ROBUF

Pointer to RS-232 output buffer
The value in these locations determines the starting address of the 256-byte RS-232 output buffer—the area where characters are stored while they await transmission via the RS-232 interface. The value here is initialized to 3328/S0D00 by the RAMTAS routine [$E093], part of the reset sequence. This places the output buffer at its default position, and no system routine changes this setting.

204-205 $CC-$CD KEYTAB

Pointer to current keyboard decode table
The value in these locations determines the starting address of the 89-byte area of memory which will be used to decode the current keyboard matrix code in location 212/$D4. The SCNKEY routine [$C55D], part of the normal IRQ sequence, checks on the shift-key status (in location 211/$D3) and selects the proper value from the list of keyboard table pointers at $83u-841/$033E-$0349.

206-207 $CE-$CF IMPARM

Pointer for Kernal PRIMM routine
These locations are used as a working pointer to the character to be printed during the Kernal PRIMM routine [$FA17].

208 SD0 NDX

Number of characters in the keyboard buffer
This location holds the number of characters awaiting processing in the keyboard buffer at 842/$034A. The value here is initialized to zero by the CINT routine, part of the RESET sequence. This location is also reset to zero by the STOP routine if the STOP key is pressed. It is incremented during the SCNKEY routine [$C55D] whenever a character is added to the buffer, and decremented whenever a key is removed (by the Kernal BASIN or GETIN routines). The value here is not allowed to exceed the maximum keyboard buffer length specified in location 2592/$0A20.

209 $D1 KYNDX

Number of characters pending from programmable key string
This location holds the number of characters remaining to be read from the string for the most recently pressed programmable key. The value here is initialized to zero by the CINT routine, part of the RESET sequence. When the press of a programmable key is detected during the SCNKEY routine [$C6CA], the length of the string for that key is stored here. The value is then decremented as each character is read from the string (by GETIN or BASIN).

210 SD2 KEYIDX

Pointer into the programmable key definition area
This location holds the offset to the next character to be read from the programmable key definition string area at 4106-4351/$100A-$10FF. When the press of a programmable key is detected during the SCNKEY routine [$C6CA], the offset to the definition string for that key is stored here. The value here is incremented as each character is read from the string.

211 $D3 SHFLAG

Shift key status flag

This location is set during the SCNKEY routine [$C55D] to indicate which of the shift keys—SHIFT, Commodore, CTRL, ALT, or CAPS LOCK—are currently being pressed. Each key has a corresponding bit which is set to %1 when the key is pressed:

Key	Bit	Bit value
SHIFT	0	l/$01
Commodore	**1**	2/$02
CONTROL	2	4/$04
ALT	3	8/$08
CAPS LOCK	4	16/$10

The values are cumulative; if both SHIFT and CONTROL are pressed simultaneously, the value here will be 5 (4 + 1). Based on the value here, the SCNKEY routine chooses a keyboard table pointer value to be stored in 204-205/$CC-$CD.

Bit 7 of this location is also used as a flag to indicate when the extra characters read using the VIC chip lines are being scanned.

212 $D4 SFDX

Current key pressed

This location is used during the SCNKEY routine [$C55D], part of the system jiffy IRQ sequence, to hold a value indicating which key was pressed. Each key has a unique keyscan matrix code here, but the code values are different from either character codes or screen codes. Refer to Appendix C for a list of keyscan codes. The key's keyscan code (0-87) serves as an offset into the keyboard decoding table pointed to by locations 204-205/$CC-$CD to select the character code to be added to the keyboard buffer at 872/$034A. A scan code of 88 indicates that no key was pressed.

It's possible to read this location as an alternative to using the BASIC GET or GETKEY statements or the machine language GETIN routine when you want to check for the press of a particular key. For example, the two following statements produce the same result, a delay until the X key is pressed:

```
100 IF PEEK(212)<>23 THEN 100
100 GET K$:IF K$o"X" THEN 100
```

Certain keyscan codes will not normally be recorded here. The codes for the left and right SHIFT keys, the CONTROL key, the Commodore key, and the ALT key—codes 15, 52, 58, 61, and 80, respectively—-are normally intercepted during the SCNKEY routine and used to generate the value at 211/SD3. The CAPS LOCK, 40/80 DISPLAY, and RESTORE keys are not part of the keyscan matrix, and the SHIFT LOCK key is just a switch that has the effect of holding down the left SHIFT key.

213 $D5 LSTX

Last key pressed
At the end of the SCNKEY routine [$C55D], the value in 212/$D4 is transferred here. This value is then used during the next pass through SCNKEY to determine if the same key is still being pressed. If so, no additional character code will be added to the keyboard buffer unless key repeating is enabled.

214 $D6 CRSW

Input source flag
This location is used during the screen editor BASIN routine [SC29B] to indicate whether the line of input is to come from the keyboard or from the screen. The default value of 0/S00 selects input from the keyboard, while a nonzero value selects input from the screen. The Kernal BASIN routine [$EF06] will set this location to 3/$03 before calling the screen editor routine when screen input is requested. Bit 7 of this location is used as an end-of-input flag; however, this is not handled properly for input from the screen. See the entry for the screen editor routine in Chapter 7 for details.

215 $D7 MODE

Active screen flag
Bit 7 of this flag determines which text screen is considered the active display. While the bit is %1, the 80-column display is selected. While the bit is %0, the 40-column display is active. Note that the inactive screen isn't actually turned off; it retains whatever display it had when the other screen was selected. However, only the active screen has a "live" cursor, and all printing is directed there. During the reset and

RUN/STOP-RESTORE sequences, the screen editor initialization routine [$C07B] sets this flag according to the position of the 40/80 DISPLAY key.

While it is often useful to check this flag to determine which display is active, it shouldn't be changed to switch active displays. Instead, use the escape sequence (ESC X) or the Kernal SWAPPER routine [$FF5F]. There's more to changing active displays than just toggling the flag bit—the active and inactive screen editor variable tables, line link bitmaps, and tab stop bitmaps must also be exchanged.

216 $D8 GRAPHM

Mode flag for 40-column screen

This location is used during the screen IRQ routine [$C194] to determine which display mode is selected for the 40-column (VIC) screen. The value here has no effect on the 80-column (VDC) screen. When this location contains 0/$00, text mode is selected. Bits *5-7* control the graphics mode configurations:

Bit	Bit value	Mode selected
5	32/$20	bitmapped
6	64/$40	split bitmapped/text
7	128/$80	multicolor

More than one of these can be selected at one time. The standard graphics modes place the following values here:

Mode	Value
GRAPHIC 0	0/$00
GRAPHIC 1	32/$20
GRAPHIC 2	96/$60
GRAPHIC 3	160/JAO
GRAPHIC 4	224/$E0

While the standard screen editor interrupt routine is in use, the value here determines how the screen mode will be set up. As a result, you cannot directly change the bitmapped or multicolor mode control bits of the VIC chip, since those bits will be set according to the value here. You can turn off the screen-setup portion of the screen editor IRQ routine by storing the value 255/$FF here. This gives you direct control over the VIC chip register settings, but disables BASIC'S ability to change display modes.

217 $D9 CHAREN

CHAREN bit shadow

Bit 2 of this location serves as a shadow for the CHAREN bit bit 2 of the processor I/O port at location l/$01. The value of the bit in this location is copied to the port bit during each pass through the text screen-setup portion of the screen editor IRQ routine [$C194]. Thus, the setting of the port bit cannot be changed directly while the standard interrupt routine is in use. Instead, you must set the bit here to the desired value and let the interrupt routine set the port bit accordingly.

The setting of the CHAREN bit determines whether the VIC chip sees the ROM character sets at offsets of 4096/$1000 and 6144/$1800 in the current video bank. When the bit is %0, the standard ROM character set is visible to the VIC. When the bit is set to %1, the VIC instead sees the true contents of memory in the video bank.

218-223 $DA-$DF SEDSAL

Screen editor zero-page work area

Assorted screen editor routines use these locations for various functions. Location 218/SDA is used as temporary storage by the routines that calculate bit positions in the line link map [$CB9F] or tab stop table [$C961, $C96C]. During a number of routines, location 222/$DE is used as temporary storage for the current cursor column, and 223/$DF is used as storage for the current cursor row.

For the PFKEY routine [$CCA2], location 218/$DA holds the length of the current key definition string. Location 219/$DB holds the total length of all programmable key definitions. Location 220/$DC holds the current key number (0-9). Location 221/$DD holds the index to the next key definition beyond the current one. Location 222/$DE holds the MMU setting for the bank where the definition string is found. 223/$DF is used as temporary storage for the index in the X register.

For the INIT80 routine [$CE0C], locations 218-219/$DA-$DB are used as a pointer to the character ROM at 53248/$D000. The screen-scrolling routine [$C40D] uses locations 218-219/SDA-DB as a pointer to the start of screen memory for the current line. Locations 220-221/$DC-$DD are used as pointers to the start of attribute memory for the current screen line.

Screen Editor Variable Table

Locations 224-249/$E0-$F9 comprise the screen editor variable table for the active display. All locations in the table are initialized during the CINT screen editor initialization routine [$C07B]. An equivalent table for whichever display is currently inactive is maintained at 2624-2649/$0A40-$0A59, Whenever the SWAPPER routine [$CD2Ej is called to switch active screen displays, the contents of this table are exchanged with the values from the inactive screen table. Thus, the table settings here are retained even when the screen is not active.

224-225 $EO-$E1 PNT

Pointer to first screen memory location for current line
Whenever the cursor is moved onto a new line, the screen memory address corresponding to the leftmost column of that line is calculated and stored in these locations. These locations can then be used as a pointer to screen memory locations for the current line. The value 236/$EC serves as an offset to the current cursor column. The low byte of the address comes from the value in the table at 49203/$C033 corresponding to the current row (multiplied by 2 if the 80-column display is active). The high byte comes from the value in the table at 49228/$C04C corresponding to the current row, adjusted for the starting screen memory page value in 2619/$0A3B in the case of the 40-column (VIC) display, or for the starting screen memory page value in 2606/$0A2E in the case of the 80-column (VDC) display. Since the tables have only 25 valid entries, the screen editor cannot support an output window with more than 25 rows.

226-227 $E2-$E3 USER

Pointer to first attribute memory location for current line
Whenever the cursor is moved onto a new line, the color memory address corresponding to the leftmost column of that line is calculated and stored in these locations. These locations can then be used as a pointer to attribute memory locations for the current line. The value 236/$EC serves as an offset to the current cursor column. The low byte of the address comes from the value in the table at 49203/$C033 corresponding to the current row (multiplied by 2 if the 80-column display is active). The high byte comes from the value in the table at 49228/$C04C corresponding to the current row, adjusted for a

starting page of 216/$D8 in the case of the 40-column (VIC) display, or for the starting color memory page value in 2607/$0A2F in the case of the 80-column (VDC) display. Since the tables have only 25 valid entries, the screen editor cannot support an output window with more than 25 rows.

228 $E4 SCBOT

Bottom margin of current window

The value in this location determines which screen row will be the bottom margin of the current output window. This value should be greater than or equal to the value in location 229/SE5. This location is reset to the maximum column number from location 237/$ED when the window is reset to full screen size, as when the CINT screen editor initialization routine is executed. This location can be assigned a specific row number using the screen editor WINDOW routine [$CA1B], which has a screen editor jump table entry at 49197/$C02D. From BASIC, the WINDOW statement can be used to change the value here. The ESC T sequence will cause the row number for the current cursor position to be stored here.

229 $E5 SCTOP

Top margin of current window

The value in this location determines which screen row will be the top row of the current output window. This value must be less than or equal to the value in location 228/$E4. The value here is reset to 0/$00, the top row of the screen, when the window is reset to full screen size, as when the CINT screen editor initialization routine is executed. This location can be assigned a specific row number using the screen editor WINDOW routine [$CA1B], which has a screen editor jump table entry at 49197/$CO2D. From BASIC, the WINDOW statement can be used to change the value here. The ESC T sequence will cause the row number for the current cursor position to be stored here.

230 $E6 SCLF

Left margin of current window

The value in this location determines which screen column will be the left margin of the current output window. This value must be less than or equal to the value in location 231/SE7. The value here is reset to 0/$00, the left edge of the

screen, when the window is reset to full screen size, as when the CINT screen editor initialization routine is executed. This location can be assigned a specific column number using the screen editor WINDOW routine [$CA1B], which has a screen editor jump table entry at 49197/$C02D. From BASIC, the WINDOW statement can be used to change the value here. The ESC T sequence will cause the column number for the current cursor position to be stored here.

231 $E7 SCRT

Right margin of current window

The vaiue in this location determines which screen column will be the right margin of the current output window. This value should be greater than or equal to the value in location 230/$E6. This location is reset to maximum column number from location 238/$EE when the window is reset to full screen size, as when the CINT screen editor initialization routine is executed. This location can be assigned a specific column number using the screen editor WINDOW routine [$CA1B], which has a screen editor jump table entry at 49197/$C02D. From BASIC, the WINDOW statement can be used to change the value here. The ESC B sequence will cause the column number for the current cursor position to be stored here.

232 $E8 LSXP

Cursor row for start of input

This location determines the starting row for the logical line of input characters to be read by the BASIN routine [$C29B]. Location 235/$EB will hold the row for the end of the input line.

233 $E9 LSTP

Cursor column for start of input

This location determines the starting column for the logical line of input characters to be read by the BASIN routine [$C29B]. Location 2608/$0A30 will hold the column for the end of the input line.

234 $EA INDX

Column of last nonspace character on logical line

The screen editor includes a routine [$CBC3] to find the position of the last nonspace character in the current logical line. That routine stores the column number of the character position here.

235 $EB TBLX

Cursor row

This location holds the cursor's horizontal position on the screen. When the cursor is moved onto a new line, the value here is used as an offset into the screen memory line base starting address tables during the calculation of the starting address for the current screen memory line (224-225/ $EO-$E1) and the starting address for the current color memory line (226-227/$E2-$E3). When the output window is cieared or the cursor is moved to the home position, the value here will be reset to the value in location 229/$E5, the top margin of the window. The value here is incremented whenever the cursor wraps around from the right margin of the window back to the left—or whenever a RETURN character (code 13/$0D), SHIFT-RETURN (code 141/$8D), or cursor-down (code 17/ $11) is printed—unless the increment would cause the value here to exceed the bottom margin value in 228/$E4. The action taken in that case depends on whether the scrolling flag (248/$F8) is set to allow new lines to be scrolled onto the screen. If so, the value here remains unchanged and a new line is opened at the bottom of the window. If scrolling is not allowed, the value here is reset to the value in 229/$E5 to wrap the cursor to the top of the window.

The PLOT routine [$CC6A] can be used to set or read the value here, but the vertical coordinate used by PLOT is relative to the current top margin. That is, the vertical offset placed here when PLOT is used will be the vertical coordinate specified in the PLOT call plus the current top margin value in 235/$E5, and the coordinate value returned by PLOT will be the value here less the current top margin value in 229/$E5.

236 $EC PNTR

Position of cursor within current physical line

This location holds the cursor's horizontal position on the screen. The value here is used as an offset from the starting address of the current screen memory line (224-225/$E0-$El) to determine the screen memory position of the current character, and as an offset from the starting address of the current color memory line (226-227/$E2-$E3) to determine the color memory position of the current character.

When the output window is cleared or when the cursor is moved to the home position, the value here will be reset to

the value in location 230/SE6, the left margin of the output window. Each time a character is printed to the window, the value here is incremented, unless the increment would cause the value here to exceed the value in location 231/$E7, In that case, the value here is reset to the left margin value in 230/ $E6. The value here is also reset to the left margin value whenever a RETURN character (code 13/SOD) or SHIFT-RETURN (code 141/$8D) is printed.

The PLOT routine [$CC6A] can be used to set or read the value here, but the coordinates supplied to PLOT are relative to the current left margin. That is, the horizontal offset placed here when PLOT is used will be the horizontal coordinate specified in the PLOT call plus the current left margin value in 230/$E6, and the coordinate value returned by PLOT will be the value here less the current left margin value in 230/SE6.

237 $ED LINES

Maximum number of rows allowed in output window
The value here determines the maximum bottom row for the output window. The current bottom row number is specified in location 228/$E4. When the window is reset to full screen size by printing two {HOME} characters (code 19/$13) in sequence (or by directly calling the screen editor window reset routine [$CA24]), location 228/$E4 will be reset to the value here. This location is set to 24/$18 during the CINT screen editor initialization routine, which establishes the default maximum of 25 horizontal rows of characters in the output window (remember that row numbering begins at zero). No system routine changes this setting, but you can reduce the value here to restrict the maximum height of the output window. However, you should not increase the value above the default setting, since the screen editor printing routines will not properly support a window more than 25 lines tall.

238 $EE COLUMNS

Maximum number of columns allowed per row
The value here determines the maximum right margin column for the output window. The current right margin column number is specified in location 231/$E7. When the window is reset to full screen size by printing two {HOME} characters (code 19/$13) in sequence (or by directly calling the screen editor window reset routine [$CA24]), location 231/$E7 will be reset

to the value here. During the CINT screen editor initialization routine, this location is set to 39/$27 if the 40-column (VIC) display is the default, or to 79/$4F if the 80-column (VDC) display is the default. This establishes the default widths of the respective displays (remember that column numbering begins at zero). No system routines change these settings, but you can reduce the value here to restrict the maximum width of the output window. However, you should not increase the value above the default settings, since the screen editor printing routines will not properly support windows wider than the respective defaults.

239 $EF DATAX

Character to print

This location is used during the screen editor printing routines to hold the character code (not the screen code) for the character to be printed.

240 $F0 LSTCHR

Last character printed

This location is used during the screen editor printing routines to hold the character code for the previous character printed. After each character is printed, the code for that character is transferred here from location 239/$EF. The value is used to detect when certain key sequences have been printed, such as the escape (ESC) sequences and the HOME HOME sequence to reset the output window margins. One shortcut to printing an escape sequence is to set this location to 27/$1B (the code for the ESC character), then call the screen BSOUT routine [$C00C] with the accumulator holding the second character of the escape sequence.

241 $F1 COLOR

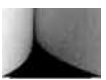
Attribute of current character

The value in this location determines the color (and attribute for the VDC display) that will be used for the next character printed to the output window. When the screen code for the character is placed in screen memory, the value here will be placed in the corresponding position in color memory. When the 40-column (VIC) screen is the active display, only the lower four bits of this location are meaningful. Those bits will hold the VIC color code (0-15) for the character position, See

the discussion of the VIC chip in Chapter 8 for details. For the 80-column (VDC) display, the lower four bits also hold the color value, but the relationship of values to colors is different from that for the VIC chip. Refer to the discussion of the VDC in Chapter 8 for more information.

When the VDC display is active, the upper four bits of this location hold the attribute value for the next character to be printed. Refer to the discussion of the VDC chip in Chapter 8 for more information on attributes. Bit 4 determines whether the character will flash. Printing character code 15/$0F will set bit 4 to %1, which specifies a flashing character. Printing character code 143/$8F resets bit 4 to %0, which turns off the flashing attribute. Bit 5 determines whether the character will be underlined. Printing character code 2/$02 will set bit 5 to %1, which specifies an underlined character. Printing character code 130/$82 resets bit 5 to %0, which turns off the underlining attribute. Bit 6 could be used to determine whether the character is reversed. Setting the bit to %1 specifies a reversed image of the character pattern, and resetting the bit to %0 specifies a normal character. However, the 128's screen editor does not make use of this feature. Instead, each standard character set contains both normal and reversed character patterns and reversed characters are obtained by selecting the reversed character pattern. Bit 7 determines which of the two character sets will be used. When the bit is %0, the first (uppercase/graphics) set is selected, while setting the bit to %1 selects the second (lowercase/graphics) set. Thus, the VDC allows both character sets to be used on the same display. When the VDC display is active, printing character code 14/$0E sets this bit to %1, and printing character code 142/$8E resets the bit to %0. If character set switching with the SHIFT-Commodore key combination is allowed, then that combination will toggle the value of this bit.

The CINT screen editor initialization routine will set this location to 13/$0D if the VIC screen is the default display, or to 7/$07 if the VDC screen is the default. This selects light green characters for the VIC display or light cyan characters with no special attributes for the VDC display. The color value in the lower four bits can be changed by printing any of the 16 color change characters. Refer to Appendix C for a list of character code values.



242 $F2 TCOLOR

Temporary storage for attribute byte
This location is used to temporarily preserve the value from 241/$F1 during screen editor routines that insert or delete characters or scroll screen lines.

243 $F3 RVS

Reverse mode flag
The value in this location determines whether reverse mode is active. Reverse mode is active whenever this location contains a nonzero value. In this case, bit 7 will be set to %1 in each screen code placed in screen memory by the BSOUT screen printing routine. This effectively converts screen codes 0-127/$00-$7F to codes 128-255/$80-$FF. In the default character sets, character patterns in the upper half of each set are the reverse image of corresponding patterns in the lower half. The value here is initialized to 0/$00 (reverse mode off) by the CINT screen editor initialization routine. This location is set to 128/S80 when the reverse-on character (code 18/$12) is printed, and reset to 0/$00 when the reverse-off character (code 146/$92) is printed. The value here is also reset to zero each time a carriage return character (code 13/$0D) or shifted return (code 141/$8D) is printed to end the line. This location can also be reset to zero to disable reverse mode with either the ESC O or ESC ESC sequences.

244 $F4 9TSW

Quote mode flag
This value in this location determines whether quote mode is active. Quote mode will be in effect whenever this location contains a nonzero value. In this case, cursor movement keys, CONTROL key combinations, Commodore-number key (color change) combinations, and the insert key (SHIFT-INST/DEL) are deferred—they appear as reverse characters within the current screen line instead of having any direct effect on the screen display. The value here is initialized to 0/$00 (quote mode off) by the CINT screen editor initialization routine. The value here is exclusive-ORed with l/$01 each time a quote character (code 34/$22) is printed. Thus, quote mode will normally be on after an odd number of quotes (1, 3, and so on) and off after an even number of quotes (2, 4, and so on). This

location is reset to zero each time a carriage return character (code 13/$0D) or shifted return (code 141/$8D) is printed to end the logical line. This location can also be reset to zero to disable insert mode with either the ESC O or ESC ESC sequences.

245 $F5 INSRT

Number of pending inserts

This location holds the number of character positions which have been inserted in the current logical line. This is significant because insert mode is normally active for inserted character positions. Insert mode is similar to quote mode—cursor movement and color change characters are deferred—except that the insert key is not deferred in insert mode and the delete key (INST/DEL) is deferred. Insert mode is active whenever this location contains a nonzero value. The value here is incremented each time a blank character position is inserted in the current line, and decremented each time a character is typed in one of the inserted positions. This location is initialized to 0/$00 (insert mode off) by the CINT screen editor initialization routine. It is also reset to zero each time a carriage return character (code 13/$0D) or shifted return (code 141/$8D) is printed to end the line. This location can also be reset to zero to disable insert mode with either the ESC O or ESC ESC sequences.

246 $F6 INSFLG

Autoinsert mode flag

Bit 7 of this location determines whether the autoinsert feature is active. When the bit is %1, autoinsert mode is active, and a space is inserted following each character printed to the screen. If the bit is %0, autoinsert mode is disabled. In this case, the cursor simply moves to the next character position after each character is printed. This location is initialized to 0/$00 (autoinsert mode off) during the CINT screen editor initialization routine. It is also reset to zero by the BASIC subroutine that sets flag values when a RUN statement is executed [$5A81]. This location is set to 255/$FF (which sets bit 7 to %1) when the ESC A sequence is printed. It can be reset to 0/$00 with the ESC C sequence.



247 $F7 LOCKS

Case switching / scroll pause control flag
Bit 7 of this location determines whether the SHIFT-Commodore key combination can be used to switch character sets. If this bit is %0, the SCNKEY routine [$C55D] will switch character sets whenever the SHIFT-Commodore combination is detected. Setting this bit to %1 disables character set switching with SHIFT-Commodore. However, you can still change character sets by printing character 14/$0E for the lowercase/ uppercase set, or character 142/$8E for the uppercase/ graphics set. There is no provision for preventing character set switching using the character codes. This location is initialized to 0/$00 (switching enabled) by the CINT screen editor initialization routine. The bit can be set to %1 by printing character code 11/$OB, and reset to %0 by printing character code 12/$0C. (Note that this is a change from earlier Commodore models, where character 8/$08 disabled switching and character 9/$09 reenabled switching.)

Bit 6 of this location controls whether the NO SCROLL key or CONTROL-S key combination can be used to pause output to the screen. If this bit is %0, NO SCROLL or CONTROL-S will pause printing to the screen until another key is pressed. Setting this bit to %1 prevents pausing, so that neither NO SCROLL or CONTROL-S will have any effect on screen output. (The Commodore key can still be used to slow down printing.) This location is initialized to 0/$00 (pause enabled), and no system routine changes the setting of this bit. Since the screen editor doesn't provide any character code or escape sequence for disabling the pause feature, you must change the value here directly if you wish to make use of the pause disable feature.

248 $P8 SCROLL

Scroll/link control flag
Bit 7 of this location is tested during the screen editor cursor movement routines to determine whether a new line will be scrolled onto the output window after printing on the current bottom line. If the bit is %0, a new blank line will be opened at the bottom of the screen (and the top line will be scrolled off the screen) after printing on the bottom line. Setting the bit to %1 prevents scrolling; after printing on the bottom line, the

cursor will wrap around to the top screen line. This bit can be set to %1 with the ESC M sequence, and reset to %0 with ESC L.

Bit 6 of this location controls whether physical screen lines can be linked together to form logical lines. For example, BASIC allows logical lines up to 160 characters long. If this bit is %0, linking is allowed. The line link bitmap at 862-865/ $035E-$0361 will indicate which physical lines are part of longer logical lines. Setting this bit to %1 disables line linking, in which case no logical line can be more than one physical line long. This location is initialized to 0/$00 (linking enabled) during the CINT screen editor initialization routine, and no system routine changes the setting of this bit. Since the screen editor doesn't provide a character code or escape sequence for changing this bit, you must change the value here directly if you wish to make use of the linking disable feature.

249 $F9 BEEPER

Bell enable flag

Bit 7 of this location controls whether or not a tone is produced when character code 7, the {BELL} character, is printed. If the bit is %0, then a tone is produced. Setting the bit to %1 prevents the tone. The location is tested during the screen BSOUT subroutine that handles character 7 [$C98E], The location is initialized to 0/$00 (bell enabled) during the CINT screen editor initialization routine. The flag bit can be set to %1 using the ESC H sequence, and reset to %0 with ESC G.

250 $FA Unused

This location is unused in the sense that it is not intentionally altered by any 128 Kernal or BASIC routine. However, a bug in the screen editor CINT [$C07Bj and SWAPPER [$CD2E] routines causes this location to be overwritten whenever those routines are executed. Because those routines are called during the RUN/STOP-RESTORE sequence, any value you place in this location will be overwritten any time you press RUN/ STOP-RESTORE, as well as whenever you switch screens. Thus, if you use this location in your programs it should be only as temporary working storage, not for important values you might want preserved in the cases mentioned above.



251-254 $FB-$FE Unused

These locations are unused by any 128 ROM routines, and are thus available for use in your BASIC and machine language programs. This area is not affected by RUN/STOP-RESTORE, but remember that all zero-page locations, including these, are cleared to zero during a reset (unless the RUN/STOP key is held down during the reset; see the reset routine [$E000] for details).

255 $FF

This location is used as part of the assembly area for character strings representing the digits of numeric values. Refer to the next section for details.

Page 1: System Stack

256-511/$0100-$01FF

This page is the system stack, the area where the 8502 microprocessor stores information such as the return addresses for interrupts and subroutine calls. Some microprocessors allow longer stacks or allow the stack to be located at various places in memory, but 6502-family microprocessors like the 128's 8502 have only one 256-byte stack, and it's always page 1. Unlike other Commodore computers, however, the 128 has the ability to make the 8502 see page 1 anywhere in memory. The MMU chip has a feature which allows the processor to exchange page 1 with another page, so that all references to page 1 (including the processor's stack manipulations) are directed to the alternate page, and references to addresses in the alternate page are directed to page 1. See the discussion of the MMU in Chapter 8 for details. The 128 does not normally make use of this feature; page 1 is normally seen at the true page 1 locations here.

The storage of data in the stack is controlled by a register in the microprocessor called the stack pointer, which serves as an index to the next available address in the stack. The stack is filled from top to bottom—from location 511/$01FF downward to 256/$100. When no data is in the stack, the stack pointer contains 255/$FF, indicating that 511/$01FF is the first available location. (The pointer is a one-byte index, to which the microprocessor automatically adds 256/$0100 to get

the actual address in page 1.) When a byte of data is pushed (added) onto the stack, the stack pointer register is automatically decremented to point to the next available address. When a byte is pulled (removed) from the stack, the register is automatically incremented. The value is not actually deleted, but incrementing the stack pointer will cause the next byte pushed onto the stack to overwrite the old value. The RESET routine [$EO00] begins by resetting the stack pointer to 255/$FF, effectively emptying the stack.

In addition to the stack's use for processor address information, BASIC uses it to hold intermediate values during expression evaluation. In earlier Commodore computers, the system stack was also used to hold information for BASIC statements such as GOSUB and FOR that loop back to another line. Since every FOR statement requires 18 bytes of stack space, and every GOSUB or DO requires 5 bytes, only a limited amount of nesting would be possible before all system stack space was exhausted. BASIC 7.0 maintains a separate stack at 2048-2559/$0800-$09FF for FOR, GOSUB, and DO. This allows BASIC 7.0 to use more deeply nested FOR-NEXT and DO-LOOP loops and more levels of subroutines—and hence more complex programs. See Chapter 3 for details of the BASIC stack.

The 8502 normally uses all of page 1 as stack space, but BASIC manipulates the stack pointer to allow the 128 to use portions of this area in other ways. The BASIC cold-start routine [$4023] resets the stack pointer to 251/$FB, so locations 508-511/$01FC-$01FF are not used by BASIC. The CLR routine [$51F8], also part of NEW and RUN, resets the stack pointer to 250/$FA. BASIC limits the stack to 201 bytes, halting with an error if all BASIC stack space is exhausted. The portions of this page used for purposes other than the stack behave like any other part of RAM.

255-266 $00FF-$010A

Assembly area for numeric value strings

The routine [S8E42] which generates a character string representing the floating-point value in FAC1 uses this area to assemble the characters for the digits of the value. The first character of the string will be either a space (for a positive value in FAC1) or a minus sign (for a negative value). When numeric values are being printed, the string is assembled start-

ing at 256/$0100. However, when the string of characters for a line number is assembled, it starts at 255/$00FF. Because the routine to add the characters here to the string pool assumes that the string starts at 256/$0100, this will cause the leading space to be omitted for line number values.

256-268 $0100-$010C

Assembly area for disk boot command

The Kernal BOOT_CALL routine [$F890] uses this area to assemble the block read command string to be sent to the drive. The default command string is U1:13001 00 (held in reverse order) to read the contents of sector 0 of track 1, the first boot sector. If more boot sectors follow, the track and sector parameters will be updated to form the commands to read the additional sectors.

256-317 $O1OO-$O13D

Tape error log

The Kernal routine which stores blocks of data on tape writes two identical copies of the data. That way, if errors are detected when the first copy is read back in, it may be possible to correct that error from the second block. Whenever the routine to load a block of data from tape [$EAEB] detects an error in a byte read from the first copy of the block, it stores the address of the erroneous byte in this area. Location 158/$9E serves as an index to the next available address slot. This area is sufficient to hold 31 error addresses, so a load error occurs on the first pass only if more than 31 errors are recorded. When the second copy of the block is read, any address for which an error was recorded on the first pass will be loaded with the corresponding byte from the second copy (unless an error was also detected for that same address in the second copy; in that case, a load error occurs).

272-290 $0110-$0122

DOS command work area

The routine [$A3C3] to assemble command strings for the various BASIC DOS support commands such as HEADER, COPY, and SCRATCH uses this area to hold information about the type of command string to assemble.

291-310 $0123-$136

PRINT USING work area

The PRINT USING routine [$9520] uses this area to hold information about the way the output string is to be formatted.

294 $0126

Command type indicator for PLAY processing

This location is used during the PLAY statement routine [$6DE1] to hold a value indicating which PLAY command (V, O, T, X, or U) is currently being processed.

311-507 $0137-$01FB

Stack space used by BASIC

This is the portion of the stack used while BASIC is active. The BASIC cold-start routine initializes the stack pointer to 251/$FB, but any subsequent NEW will reinitialize it to 250/$FA. Thus, locations 508-511/$01FC-$01FF (and, after the first NEW, also 5O7/$1FB) are unused and available for your own programming. BASIC requires that at least 44 bytes be available in the stack at the start of any expression evaluation. The stack pointer is tested during the main expression evaluation routine [$77EF]; if it is less than 99/$63, a FORMULA TOO COMPLEX error occurs. (This is a change from Commodore 64 BASIC, where the same situation would result in an OUT OF MEMORY error.)

Input Buffer

512-672 $0200-$O2AO BUF

BASIC and monitor input buffer

This 161-byte area is used to hold input for both BASIC and the monitor. The BASIC input routine [$4F93] allows logical program lines up to 160 characters long to be entered. A byte with the value 0/$00 is added following the last character of the input. If the line starts with a line number, the line here is tokenized and transferred to the BASIC program text area. An immediate mode line (one with no line number) is tokenized and then executed from the buffer. This buffer is also used to hold input characters for the GET, GET#, GETKEY, INPUT, and INPUT* statements, which is why those statements are

not allowed in immediate mode. The monitor main loop [$B08B] accepts command strings up to 159 characters long, and also adds a zero byte following the last character of input to mark the end of the command.

673 $O2A1 Unused

The BASIC and monitor input routines restrict the length of an input line to 160 characters plus a zero byte to mark the end of input in the buffer, for a maximum of 161 bytes. Thus, this location will never be used for input, and is available for other uses.

Common Indirect Routines

The routines at 674-763/$02A2-$02FB are copied here from Kernal ROM at 63488-63577/$F800-$F859 by the routine at 57549/$E0CD, part of the reset sequence. The routines are placed here in page 2 because this is part of the IK block of memory that is visible in all banks. These routines are the key to the operation of the 128—they make the memory banking system possible by allowing a routine in one bank configuration to access data or call routines in another configuration. For example, these routines allow BASIC ROM routines to use different blocks of RAM for program text and variables, and to see program text in areas of RAM that lie at the same addresses as BASIC ROM itself. These routines are so integral to the successful operation of the 128 that the system will probably crash almost immediately if the routines are accidentally changed or overwritten.

674-686 $02A2-$02AE FETCH

Retrieves a value from any bank

This routine loads the accumulator value with the contents of a specified location in any bank. To use this routine, you must set up a two-byte pointer in zero page to hold the address of the target location, then store the one-byte address of the zero-page pointer in location 682/$02AA. You can use the Y register to specify an offset from the pointer address for the target location. (If no offset is desired, be sure that the Y register contains 0/$00.) The X register should contain the MMU configuration register setting value which will establish a memory configuration in which the target location is visible.

The routine reads and stashes the current MMU configuration register setting, then uses the value in the X register upon entry as the new configuration register setting. Next, the contents of the location specified by the address in the pointer plus the offset in the Y register are loaded into the accumulator. The MMU configuration register is restored to its original value before exiting.

This routine is normally called via its related Kernal routine at 63440/$F7D0, which has a jump table entry at 65396/$FF74. When calling via the Kernal routine, the accumulator should contain the zero-page pointer address; the Kernal routine stores the accumulator value upon entry in 682/$02AA, performing that setup step for you. The X register should contain a bank number (0-15) rather than an MMU configuration register setting value, since the Kernal routine also performs the chore of converting the bank number into a configuration register value,

687-701 $02AF-$02BD STASH

Stores a value in any bank

This routine stores the contents of the accumulator at a specified location in any bank. Before calling this routine, you must set up a two-byte pointer in zero page to hold the address of the target location, then store the one-byte address of the 2ero-page pointer in location 697/$02B9. You can use the Y register to specify an offset from the pointer address for the target address. (If no offset is desired, be sure that the Y register contains 0/$00.) The X register should contain the MMU configuration register setting value which will establish a memory configuration in which the target location is visible.

The routine reads and stashes the current MMU configuration register setting, then uses the value in the X register upon entry as the new configuration register setting. Next, the contents of the accumulator upon entry are stored in the location specified by the address in the pointer plus the offset in the Y register. The MMU configuration register is restored to its original value before exiting.

This routine is normally called via its related Kernal routine at 63450/$F7DA, which has a jump table entry at 65399/$FF77. When calling via the Kernal routine, the X register should instead contain a bank number (0-15), since the

Kernal routine performs the chore of converting the bank number into an MMU configuration register setting value.

702-716 $02BE-$02CC CMPARE

Compares accumulator contents against a value from any bank
This routine compares the accumulator value against the contents of a specified location in any bank. Before calling this routine, you must set up a two-byte pointer in zero page to hold the address of the target location, then store the one-byte address of the zero-page pointer in location 712/$02C8. You can use the Y register to specify an offset from the pointer address for the target address, (If no offset is desired, be sure that the Y register contains 0/$00.) The X register should contain the MMU configuration register setting value which will establish a memory configuration in which the target location is visible.

This routine reads and stashes the current MMU configuration register setting, then uses the value in the X register upon entry as the new configuration register setting. Next, the value in the accumulator upon entry is compared against the contents of the location specified by the address in the pointer plus the offset in the Y register. The MMU configuration register is restored to its original value before exiting. The status register value will reflect the result of the comparison.

This routine is normally called via its related Kernal routine at 63459/$F7E3, which has a jump table entry at 65402/$FF7A. When calling via the Kernal routine, the X register should instead contain a bank number (0-15), since that routine performs the chore of converting the bank number into an MMU configuration register setting value.

717-738 $02CD-$02E2 JSRFAR

Calls a subroutine in any bank
(This routine has a Kernal jump table entry at 65390/$FF6E.)

The routine here will jump to a subroutine at any address in any standard bank configuration. Upon completion of the target routine, control is returned to the routine which called JSRFAR, just like a JSR. However, this routine leaves the system in the bank 15 configuration, so a routine that uses JSRFAR must be located in an area of memory visible in the bank 15 configuration for JSRFAR to properly return to the calling routine.

Before calling the routine you must load location 2/$02 with the bank number (0-15) of the target routine and locations 3-4/$03-$04 with the address of the target routine. In contrast to the usual low-byte/high-byte format, location 3/$03 should be loaded with the high byte of the address and location 4/$04 with the low byte. Location 5/$05 should be loaded with the value you want in the status register when the target routine is called (use 0/$00 if you don't want any status register bits set). Optionally, you can also load locations 6-8/$06-$08 with any values you wish the accumulator, X register, and Y register, respectively, to have when the target routine is called.

The routine calls JMPFAR to call the subroutine addressed in locations 3-4 in the bank specified in location 2 and with the status register value specified in location 5 and processor register values from locations 6-8. Upon return from the target routine, the exit values of the accumulator, X register, and Y register are stored in location 6-8/$06-$08, respectively. The value of the status register upon exit from the target routine is stored in location 5/$05, and the exit value of the processor stack pointer is recorded in location 9/$09. Finally, the routine switches the system to the bank 15 configuration before returning to the calling routine.

739-763 $02E3-$02FB JMPFAR

Jumps to a routine in any bank
(This routine has a Kernal jump table entry at 65393/SFF71.)

The routine here will jump to a routine at any address in any standard bank configuration. Before calling the routine you must load location 2/$02 with the bank number (0-15) of the target routine and locations 3-4/$03-$04 with the address of the target routine. In contrast to the usual low-byte/high-byte format, location 3/$03 should be loaded with the high byte of the address and location 4/$04 with the low byte. Location 5/$05 should be loaded with the value you want in the status register when the target routine is called (use 0/$00 if you don't want any status register bits set). Optionally, you can also load locations 6-8/$06-$08 with any values you wish the accumulator, X register, and Y register, respectively, to have when the target routine is called. The routine pushes the address and status register values onto the stack, converts the bank number value to a configuration register value, stores

that value in the MMU configuration register, loads the processor registers from locations 6-8, and executes an RTI instruction, which causes the processor to retrieve status register and address values from the stack and resume processing at the specified address.

Indirect Vectors

The next 66 locations are indirect vectors for a variety of BASIC, Kernal, and screen editor routines. An indirect vector is a pair of locations that hold an address for an indirect jump instruction, such as JMP ($0300). The target address of the JMP will be determined by the value in the specified indirect vector. Having ROM routines jump through indirect vectors greatly increases the flexibility of the computer. Even though it isn't possible to change a routine in ROM, it's possible to add to or modify a routine that has an indirect vector by redirecting the vector to a RAM-based routine.

764-765 $02FC-$02FD ESC_FN_VEC

Indirect vector in extended function execution routine

The indirect jump through this vector is taken in the extended function handling subroutine when a two-byte extended function token is found for which the second byte is greater than the largest standard extended function token (10/$0A). When the jump is taken, the accumulator will hold the out-of-range token value and the status register carry bit will be set. If carry is not clear upon return from the jump, a SYNTAX error message will be generated. The vector normally holds 19576/$4C78, the address of the instruction following the call to this vector. This doesn't change the carry setting, so out-of-range extended function tokens normally result in an error message. If you add new functions to BASIC, you'll need to change this vector to point to the routine which executes your new function. See Chapter 5 for an example.

766-767 $02FE-$02FF BNKVEC

Reserved indirect vector

These two locations are not used by system ROM routines. Commodore literature indicates that they are reserved for use as an indirect vector for function ROM routines.

BASIC Indirect Vectors

The next nine vectors, 768-785/$0300-$0311, are used in BASIC statement processing routines. The default values for these vectors are copied from a table at 16999-17016/ $4267-$4278 in BASIC ROM by the BASIC vector initialization routine [$4251], part of the cold-start sequence. Thus, unlike the Kernal indirect vectors, the BASIC vectors are not affected by the RUN/STOP-RESTORE sequence. Any changes you make to the vectors will remain in effect until the next cold start of BASIC, as during a reset.

768-769 $0300-$0301 IERROR

Indirect vector for BASIC error handling routine

In BASIC ROM, the jump through this vector is taken at the beginning of the error handling routine (ERROR [$4D3C]). At the point the jump is taken, the X register will contain the current BASIC error number (0-41, or 128 to print READY) and the accumulator will hold the last character read from program text. The default target address of the vector is 19775/$4D3F, which simply reenters the error handling routine at the point immediately following the jump. You can redirect this vector to change the way BASIC handles errors.

In addition to modifying error handling, you can also use this vector to provide an alternate method of adding commands to BASIC.

770-771 $0302-$0303 IMAIN

Indirect vector in main BASIC loop

The jump through this indirect vector is taken in the main BASIC direct mode routine [$4DB7] at the point immediately after the READY prompt has been printed and the mode flag (127/$7F) has been set for immediate mode. The vector normally holds 19910/$4DC6, the address of the instruction immediately following the indirect jump. You can redirect this vector to a routine of your own if you wish to change the behavior of BASIC'S immediate mode.

772-773 $0304-$0305 ICRNCH

Indirect vector in BASIC tokenization routine

The jump through this indirect vector is taken at the beginning of the CRUNCH routine [$430A], which is responsible for converting lines of input text into tokenized program lines.

The vector normally holds 17165/S430D, the address of the instruction immediately following the indirect jump. You can redirect this vector to a routine of your own if you wish to change the way program lines are tokenized.

774-775 $0306-$0307 IQPLOP

Indirect vector in BASIC detokenization routine
The jump through this indirect vector is taken in the QPLOP routine [$5123] at the point where the accumulator contains the next character to be listed from the program line. The vector normally holds 20817/S5151, the address of the instruction immediately following the indirect jump. You can redirect this vector to a routine of your own if you wish to change the way program lines are listed.

776-777 $0308-$0309 IGONE

Indirect vector in BASIC execution routine
The jump through this indirect vector is taken at the beginning of the GONE routine [S4F92], the routine to execute a program line. The vector normally holds 19106/$4AA2, the address of the instruction immediately following the indirect jump. You can redirect this vector to a routine of your own if you wish to change the way program lines are executed.

778-779 $030A-$030B IEVAL

Indirect vector in BASIC evaluation routine
The jump through this indirect vector is taken at the beginning of the EVAL routine, which determines the value of the next variable, string, or number in the program. The vector normally holds 30938/$78DA, the address of the instruction immediately following the indirect jump. You can redirect this vector to a routine of your own if you wish to change the way values are evaluated.

780-781 $030C-$030D ICRNCH2

Indirect vector for tokenizing additional keywords
The jump through this vector is taken in the tokenization routine at the point where the first character of the keyword has been read into the accumulator and the carry bit has been set. If carry is still set upon return from this jump, the tokenization process will proceed normally. The vector normally holds 17185/$4321, the address of the instruction immediately fol-

lowing the jump. Thus, the jump normally has no effect. If you want to add extended tokens to BASIC, you should redirect this vector to your routine to tokenize the new keywords. The routine should compare the text pointed to by 61-62/$3D-$3E with the target keyword. If a match is found, your routine should return with the second byte of the two-byte extended token in the accumulator. The X register should be set to indicate whether the keyword is a statement or a function. X should be set to 0/$00 for a function, in which case the first byte will be 206/$CE, or to 255/$FF for a statement, in which case the first byte will be 254/$FE. The Y register should contain the length of the filename. Finally, you should make sure that the carry bit is clear upon exit so that your new token will be properly processed.

782-783 $030E-$030F IQPLOP2

Indirect vector for detokenizing additional keywords
The jump through this indirect vector is taken in the routine that lists BASIC program lines at the point where two-byte extended statement or function tokens have been found which are greater than the largest standard tokens. When the jump is taken, the accumulator will hold the second byte of the offending token and the X register will hold 0/$00 if the first byte was 206/$CE, indicating an extended function token, or 255/$FF if the first byte was 254/$FE, indicating an extended statement token. The status register carry bit will also be set. If that bit is still set upon return from this indirect jump, the character will simply be printed. However, if carry is cleared, the extended keyword will be listed. The vector normally holds 20941/S51CD, the address of the instruction immediately following the indirect jump, so carry will normally remain set. If you add new extended keywords to BASIC, you should change this vector to point to the routine to support listing the keywords.

784-785 $0310-$0311 IGONE2

Indirect vector in extended statement execution subroutine
The jump through this indirect vector is taken in the statement execution routine at the point where a two-byte extended statement token has been found with a value greater than one of the standard extended statement tokens (second byte greater than 38/$26). When the jump is taken, the accumu-

lator will hold the second byte of the extended token and the carry bit will be set. If the carry bit is not clear upon return, a SYNTAX error will be generated. The vector normally holds 19369/$4BA9, the address of the instruction immediately following the indirect jump. Thus, the out-of-range token will normally cause an error. If you add new extended-token statements to BASIC, you should change this vector to point to the address of the routine which executes the new statement. See Chapter 5 for an example.

786-787 $0312-$0313 Unused

These two locations are not used for any system vector, and are thus available for your programming. For example, you could use these locations to set up an indirect vector in one of your own programs, or to store the original value when changing one of the other vectors.

Kernal Indirect Vectors

The next 16 vectors, 788-819/$0314-$0333, are initialized from a table at 57459-57490/$E073-$E092 in Kemal ROM by the RESTOR routine [$E056]. The RESTOR routine is called during both the reset and RUN/STOP-RESTORE sequences, so either of those will reinitialize the vectors. The values in this vector table can be read or modified using the Kernal VECTOR routine [$E05BJ.

788-789 $0314-$03l5 IIR9

Indirect vector to IRQ interrupt handling routine

When an IRQ interrupt occurs or a BRK instruction is executed, a jump is automatically taken through the processor IRQ vector at 65534/$FFFE to the handling routine at 65303/$FF17. That routine stores the accumulator, X and Y register, and bank configuration values on the stack, then checks whether the routine was called as the result of an IRQ or a BRK. If an IRQ was responsible, a jump is taken through this indirect vector. The vector normally holds 64101/$FA65, the address of the standard system IRQ routine. You can redirect this vector to a routine of your own to add custom steps to the IRQ process. However, your target routine must be visible in bank 15, since that is how memory will be configured when

the jump through this vector is taken. If your routine does not jump to the standard IRQ handler, it must exit by jumping to the common IRQ exit routine at 65331/$FF33.

790-791 $0316-$0317 IBRK

Indirect vector to BRK instruction handling

When an IRQ interrupt occurs or a BRK instruction is executed, a jump is automatically taken through the processor IRQ vector at 65534/$FFFE to the handling routine at 65303/$FF17. That routine stores the accumulator, X and Y register, and bank configuration values on the stack, then checks whether the routine was called as the result of an IRQ or a BRK. If the execution of a BRK was responsible, a jump is taken through this indirect vector. The vector normally holds 45059/$B003, the address of the BRK entry into the machine language monitor. You can redirect this vector to a routine of your own if you want some other handling of BRK instructions.

792-793 $0318-$0319 INMI

Indirect vector to NMI interrupt handling routine

When an NMI interrupt occurs, a jump is automatically taken through the processor NMI vector at 65530/$FFFA to the NMI handling routine at 65285/$FF05. That routine stores the accumulator, X and Y register, and bank configuration values on the stack, then configures the system for bank 15 and takes a jump through this indirect vector. The vector normally holds 64064/$FA40, the address of the standard system NMI service routine. You can redirect this vector to a routine of your own to add custom steps to the NMI process. However, your routine must be in an area of memory visible in bank 15, since that is how memory will be configured when the jump is taken. If your routine does not jump to the standard NMI handler, it must exit by jumping to the common IRQ exit routine at65331/$FF33.

794-795 $031A-$031B IOPEN

Indirect vector to Kernal OPEN routine

This vector is the normal link between the Kernal jump table entry at 65472/$FFC0 and the OPEN routine at 61373/$EFBD, You can redirect this vector to a routine of your own if you wish to modify the behavior of OPEN.



796-797 $031C-$031D ICLOSE

Indirect vector to Kernal CLOSE routine
This vector is the normal link between the Kernal jump table entry at 65475/$FFC3 and the CLOSE routine at 61832/ $F188. You can redirect this vector to a routine of your own if you wish to modify the behavior of CLOSE. When the jump is taken, the accumulator should hold the number of the logical file to be closed.

798-799 $031E-$O3IF ICHKIN

Indirect vector to Kernal CHKIN routine
This vector is the normal link between the Kernal jump table entry at 65478/$FFC6 and the CHKIN routine at 61702/ $F106. You can redirect this vector to a routine of your own if you wish to modify the behavior of CHKIN. When the jump is taken, the X register should hold the number of the logical file selected as the input source.

800-801 $0320-$0321 ICKOUT

Indirect vector to Kernal CKOUT routine
This vector is the normal link between the Kernal jump table entry at 65481/$FFC9 and the CKOUT routine at 61772/ $F14C. You can redirect this vector to a routine of your own if you wish to modify the behavior of CKOUT. When the jump is taken, the X register should hold the number of the logical file selected as the output source.

802-803 S0322-S0323 ICLRCH

Indirect vector to Kernal CLRCH routine
This vector is the normal link between the Kernal jump table entry at 65484/$FFCC and the CLRCH routine at 61990/ $F226. You can redirect this vector to a routine of your own if you wish to modify the behavior of CLRCH.

804-805 S0324-$0325 IBASIN

Indirect vector to Kernal BASIN routine
This vector is the normal link between the Kernal jump table entry at 65487/$FFCE and the BASIN routine at 61190/ $EF06. You can redirect this vector to a routine of your own if you wish to modify the behavior of BASIN. The routines which call BASIN expect it to return a character in the accumulator.

806-807 $0326-$0327 IBSOUT

Indirect vector to Kernal BSOUT routine

This vector is the normal link between the Kernal jump table entry at 65490/$FFD2 and the BSOUT routine at 61305/$EF79. You can redirect this vector to a routine of your own if you wish to modify the behavior of BSOUT. When this routine is called, the value to be output should be in the accumulator.

808-809 $0328-^0329 ISTOP

Indirect vector to Kernal STOP routine

This vector is the normal link between the Kernal jump table entry at 655O5/$FFE1 and the STOP routine at 63086/$F66E. You can redirect this vector to a routine of your own if you wish to modify the behavior of STOP. The routines which call STOP expect it to return with the status register Z bit set if the RUN/STOP key was pressed, or clear otherwise.

810-811 $032A-$032B IGETIN

Indirect vector to Kernal GETIN routine

This vector is the normal link between the Kernal jump table entry at 65508/$FFE4 and the GETIN routine at 61163/$EEEB. You can redirect this vector to a routine of your own if you wish to modify the behavior of GETIN. The routines which call GETIN expect it to return a character code in the accumulator.

812-813 $032C-$032D ICLALL

Indirect vector to Kernal CLALL routine

This vector is the normal link between the Kernal jump table entry at 65511/$FFE7 and the CLALL routine 61986/$F222. You can redirect this vector to a routine of your own if you wish to modify the behavior of CLALL,

814-815 $032E-$032F IEXMON

Indirect vector in monitor command execution routine

This indirect vector appears in the machine language monitor's main loop [$B08B] at the point where the first nonspace character has been read from the input buffer and is ready to be interpreted as a command. The vector normally holds the address 45062/$B006, which in turn is a vector back to 45234/$B0B2, the address immediately following the indirect jump.

However you can redirect this vector to a routine of your own if you wish to add commands to the machine language monitor. The following example adds two new monitor commands—P, which behaves like D (disassemble) but routes output to the printer, and Q, which closes the file to the printer:

```
0D00  LDA  #$0B   ;Redirect vector to new handling routine
0D02  STA  $032E
0D05  LDA  #$0D
OD07  STA  $032F
ODOA  RTS
ODOB  CMP  #$51   ;Is character code for Q?
ODOD  BNE  $OD1A
ODOF  LDA  #$04   ;If so, close logical file 4
0D11  JSR  $FFC3
0D14  JSR  $FFCC  ;Restore normal I/O channels (CLRCH)
0D17  JMP  $B08B  ;Retum to monitor main loop
0D1A  CMP  #$50   ;Is character code for P?
GD1C  BNE  $0D3S
0D1E  LDA  #$04   ;H so, OPEN 4,4,0
0D2O  TAX
0D21  LDY  #00
0D23  JSR  $FFBA
0D26  LDA  #$00
0D28  JSR  $FFBD
0D2B  JSR  $FFCO
0D2E  LDA  #$04   ;Set logical file 4 for output
0D30  JSR  $FFC9
D033  LDA  #$44   ;Change monitor command to D
                  (disassemble)
0D35  JMP  $B006  ;Return to monitor command processing
                  loop
```

816-817 $0330-$0331 ILOAD

Indirect vector in Kernal LOAD routine

This indirect vector appears in the Kernal LOAD routine [$F265] at the point after the starting address (in X and Y when the routine is entered) has been stored in 195-196/ $C3-$C4. The accumulator should still contain a value indicating whether the operation is a load or a verify (0/$00 for load, nonzero for verify). The vector normally holds 62060/$F26C, the address immediately following the indirect jump. You can redirect this vector to a routine of your own if you wish to modify the behavior of LOAD.

818-819 $0332-$0333 ISAVE
Indirect vector in Kernal SAVE routine
This indirect vector appears in the Kernal SAVE routine [$F53E] at the point after the ending address has been stored in 174-175/$AE-$AF and the starting address has been stored in 193-194/$C1-$C2. The vector normally holds 62798/ $F54E, the address immediately following the indirect jump. You can redirect this vector to a routine of your own if you wish to modify the behavior of SAVE.

Screen Editor Indirect Vectors

The next five vectors, 820-829/$0334-$033D, are copied from a table at $C065-$C06E in screen editor ROM by the CINT routine [$C07B] during the reset sequence. CINT is also part of RUN/STOP-RESTORE, but a flag in the routine is normally used to skip the vector initialization step in this case. As a result, vector addresses aren't usually changed by RUN/STOP-RESTORE.

820-821 $0334-$0335 CTLVEC
Indirect vector in screen BSOUT handling
The jump through this indirect vector is taken as the first step in the screen BSOUT subroutine [$C7B6] which processes character code values less than 32/$20. At the time the jump is taken, the accumulator holds the current character code. The vector normally holds 51129/$C7B9, the address immediately following the indirect jump. You can change this vector to point to a routine of your own if you wish to change the printing behavior of character codes in the range 0-31/ $00-$lF. All codes in this range perform cursor movements, color changes, or other control functions rather than printing characters. If you wish to add new control functions, codes 0, 1, 3, 4, 6, 16, 21-23, 25, and 26 are currently unused.

822-823 $0336-$0337 SHFVEC
Indirect vector in screen BSOUT handling
The jump through this indirect vector is taken as the first step in the screen BSOUT subroutine [$C802] which processes character code values greater than 127/$7F. At the time the jump is taken, the accumulator holds the current character code. The vector normally holds 51205/$C805, the address

immediately following the indirect jump. You can change this vector to point to a routine of your own if you wish to change the printing behavior of character codes in the range 128-255/$80-$FF. Codes 128-159/$81-$9F perform cursor movements, color changes, or other control functions rather than printing characters. If you wish to add new control functions, codes 128, 131, and 132 are currently unused.

824-825 $0338-$0339 ESCVEC

Indirect vector in ESC sequence handling routine

The jump through this indirect vector is taken as the first step in the screen BSOUT subroutine [$C9BE] which processes ESC (escape) key sequences. At the time the jump is taken, a test will have determined that the previous character was ESC (code 27/$lB). The accumulator holds the current character code, the one which followed ESC. The vector normally holds 51649/$C9C1, the address immediately following the indirect jump. You can redirect this vector to add your own ESC sequences. The following example adds ESC T, which moves the position of the bitmap/text division of a screen line up one row each time the sequence is used:

```
1400    LDA   #$0B    ;Redlrect vector to handling routine
1402    STA   $0338
1405  LDA   #$14
1407    STA   $0339
140A    RTS
140B    CMP   #$5E    ;Is character t?
140D  BEQ   $1412
140F    JMP   $C9C1   ;If not, jump to normal processing
                      routine
1412    LDA   $D8     ;Is a split screen in use?
1414    AND   #$40
1416    BEQ   $140F   ;If not, use normal processing routine
1418    LDA   $0A34   ;Is the split already at the top row of the
                      screen?
141B    CMP   #$3A
141D    BCC   $140F   ;If so, ignore this sequence
141F    SBC   #$08    ;Move the split position up one row (8
                      scan lines)
1421    STA   $0A34
1424  RTS
```

826-827 $033A-$033B KEYVEC

Indirect vector in keyboard scanning routine

The jump through this indirect vector is taken during the SCNKEY routine [$C55D] at the point following completion of the keyscan and the evaluation of the shift key status. At the point the jump is taken, the accumulator (along with location 212/$D4) will contain the current keyboard matrix code, and location 211/$D3 will reflect the current shift key status. The vector normally holds the address 5O657/$C5E1, the point in the keyscan routine immediately following the indirect jump, but you can redirect the vector if you wish to change the behavior of the keyscan routine.

828-829 $033C-$033D KEYCHK

Indirect vector in keyboard scanning routine

The jump through this indirect vector is taken during the SCNKEY routine [$C55D] after the character code for the keypress has been read from the decoding table and immediately before the test for a programmable key. At the point the jump is taken, the accumulator will contain the character code corresponding to the current keypress and the X register will contain the current shift key status (from 211/$D3). The vector normally holds 50861/$C6AD, the point in the keyscan routine immediately following the indirect jump, but you can redirect this pointer to modify the behavior of the keyscan.

One use of this vector is to disable programmable keys. While the definition strings are handy, sometimes—particularly when you are adapting programs from the Commodore 64—you might like for them to instead generate their standard character codes. One way to achieve this is to change this pointer so that the test for programmable keys is bypassed:

```
POKE 828,183
```

This changes the low byte of the pointer so that the target address becomes 50871/$C6B7, the point in the routine immediately beyond the test for programmable keys.

Screen Editor Tables

Locations 830-865/$033E-$0361 are the domain of the screen editor.

830-841 $033E-$0349 DECODE

Keyboard table pointers

These six 2-byte pointers hold the starting addresses of the 89-byte tables used to translate the matrix code for the current keypress into a character code:

Pointer	Shift pattern	**Default table address**
830-831/$033E-$033F	Unshifted	64128/$FA80
832-833/$0340-$0341	SHIFT	64217/$FAD9
834-835/$0342-$0343	Commodore	64306/SFB32
836-837/$0344-$0345	CONTROL	64395/$FB8B
838-839/$0346-$0347	ALT	64128/$FA80 (same as unshifted)
840-841/$0348-$0349	CAPS LOCK	64484/$FBE4

The status of the five shift keys, recorded in 211/$D3, is used to select one of the table addresses from this area. If no shift key is pressed, the unshifted table is used. If one shift key is pressed, the appropriate decoding table is selected. If more than one shift key is pressed simultaneously, the table is selected as follows: CONTROL has the highest priority; when it is pressed in combination with any other shift keys, the CONTROL table is used. The SHIFT and Commodore keys are next in priority; however, when they are pressed simultaneously, no decoding table is selected (although the combination may cause character set switching). ALT and CAPS LOCK have the lowest priority. They are effective in selecting the decoding table only if no other shift keys are being pressed. If pressed simultaneously, both are ignored and the unshifted table is used. Once a table is selected, its address is loaded into 204-205/$CC-$CD, and the current matrix code in 212/$D4 is used as an offset to the specific character code to be retrieved from the table.

The default decoding table addresses are copied here from a table at 49263/$C06F in screen editor ROM by the CINT screen editor initialization routine [$C07B] during the reset sequence. CINT is also part of the RUN/STOP-RESTORE sequence, but it includes a flag that normally prevents the vectors from being reinitialized in that case. To redefine the 128 keyboard, you need only set up a new decoding table in RAM and change one of the address values here to point to the new table. For example, if you've used the CAPS LOCK key, you've probably discovered that it doesn't appear to affect the Q key. Actually, the problem is that whoever prepared

the CAPS LOCK decoding table used the wrong value for the Q key entry. The following shows how to fix the CAPS-Q bug by setting up a new copy of the decoding table for that shift pattern:

```
100 REM ** COPY CAPS LOCK TABLE TO RAM
110 FOR 1=0 TO 88:POKE 6912+I,PEEK(64484+I):NEXT
120 REM ** CHANGE INCORRECT CHARACTER CODE FOR Q
130 POKE 6912+62,209
140 REM ** REDIRECT POINTER TO NEW TABLE
150 POKE 840,0:POKE 841,27
```

A custom table should consist of 89 values in matrix code order. Refer to Tables 9-1-9-5 for a listing of the default tables. The final value in the table should be 255/$FF, and you should be sure to include the shift key codes in the proper locations. The following program sets up a Dvorak-style keyboard:

```
100 FOR 1=0 TO 88:READ K:POKE 691.2 + 1 ,K:NEXT
110  POKE  830,0:POKE  S31,27:END
120  DATA  20,  13,  29,  136,  133,  134,  135,  17,  51,  44
130  DATA  65,  52,  59,  79,  46,  I,  53,  80,  69,  54
140  DATA  74,  85,  89,  81,  55,  70,  73,  56,  88,  68
150  DATA  71,  75,  57,  67,  72,  48,  77,  84,  82,  66
160  DATA  43,  76,  78,  45,  86,  83,  47,  87,  92,  42
170  DATA  59,  19,  1,  61,  94,  90,  49,  95,  4,  50
180  DATA  32,  2,  39,  3,  132,  56,  53,  9,  50,  52
190  DATA  55,  49,  27,  43,  45,  10,  13,  54,  57,  51
200  DATA  8,  48,  46,  145,  17,  157,  29,  255,  255
```

842-851 $034A-$0353 KEYBUF

Keyboard buffer

This ten-byte area is the keyboard buffer. When the SCNKEY routine [$C55D] detects a valid keypress, it generates a corresponding character code. The character code is then stored in this buffer to await processing. (The Kernal GETIN and BASIN routines are normally used to retrieve characters from this buffer.) Location 208/$D0 holds the number of characters currently waiting in the buffer. The maximum number of characters that can be held in the buffer is determined by the value in location 2592/$0A20. If the value there is greater than 10, the keyboard buffer will overwrite the following memory areas such as the tab stop bitmap. When the value in 208/$DO equals the value in 2592/$0A20, the buffer is full; any further

keypresses will be ignored until one or more characters are removed from the buffer.

This key buffering system allows for a powerful programming technique known as the dynamic keyboard. By storing character code values in the buffer and storing the number of characters in 208/$D0, a program can appear to type on the keyboard. For example, the following lines add a default answer to the INPUT prompt:

```
200 POKE 842,89: POKE 208,1: REM PLACE Y IN BUFFER
210 INPUT'ARE YOU SURE";A$
```

When the INPUT statement begins to look for characters, it will find the Y already in the buffer.

An even more powerful use of the dynamic keyboard technique is to allow a program to execute a series of commands after it ends. When a program is finished executing, BASIC looks to the keyboard buffer for the characters of the next command. Thus, any characters placed in the buffer while a program is running will effectively be typed if the program ends. Since the buffer can hold only ten characters, the common practice is to print commands at carefully planned places on the screen, then fill the buffer with cursor movement and RETURN characters to execute the commands. The following program illustrates this technique. It creates DATA statements for one of the sprite patterns. You can adapt the program for your own needs by changing the values in line 10. AD is the starting address of the data, NI is the number of DATA items to be generated, and LN is the line number of the first DATA statement to be generated. The program prints a DATA line and a GOTO statement on the screen, then places {HOME} {RETURN} {RETURN} in the buffer and ends. The buffered characters are executed, entering the DATA line and restarting the program.

```
10  AD=3584:NI=64:LN=100!l=0
20  IF  I=>NI  THEN  END
30  PRINT"[CLR}";LN;"DATA";:LN=LN+I0:J=0
40  PRINT  PEEK(AD+I);:I=I+1:IP  I=>NI  THEN  60
50  J=J+1;IF  J<8  THEN  PRINT"{LEFT},";:GOTO  40
60  PRINT:PRINT"GOTO  20"
70  POKE  842,I9:POKE  843,13:POKE  844,.13:POKE  203,3:E
    ND
```

852-861 $0354-$035D TABMAP

Tab stop bitmap

These ten bytes provide an 80-bit map of the display's horizontal character positions. Each horizontal position in the currently active display has a corresponding bit in the map. For the VIC chip's 40-column display, only the first five bytes (40 bits) are used. When a bit is set to %1, a tab stop is established at the corresponding screen column. Printing the TAB character, code 9/$09, or pressing the TAB key will move the cursor rightward to the next tab stop (or the right window margin if no tab stops are set between the current cursor position and the right margin).

During the CINT screen editor initialization sequence, all locations in the map are set to 128/$80, which establishes a tab stop every eighth column. Printing character code 24/$18 (or pressing SHIFT-TAB) toggles the map bit corresponding to the current cursor column, setting a tab stop if the bit was previously %0 or clearing the tab stop if the bit was previously %1. The ESC Z sequence can be used to clear all tab stops (all locations in the map will be filled with 0/$00), and the ESC Y sequence can be used to restore default tab stops (all locations in the map will be filled with 128/S80).

When the active display is switched, screen editor SWAPPER routine [$CD2E] exchanges the contents of this area with the contents of locations 2656-2665/$0A60-$0A69, the storage area for the inactive display tab stop bitmap. Thus, tab stop settings are preserved while the screen is inactive,

862-865 $035E-$0361 LNKMAP

Line link bitmap

These four bytes are used to provide a 25-bit map of the 25 rows of the active screen display (bits 0-6 of location 865/$361 are unused). Each row of the currently active display has a corresponding bit in the map. When a bit is set to %1, the corresponding row is linked to the row above as part of a logical line. Bits set to %0 indicate unlinked lines (or rows that are the first physical line of a logical line). These locations are cleared to 0/$00, unlinking all lines, whenever the output window is cleared or reset to full screen size, or whenever the screen editor WINDOW routine is used to change the size of the output window. A screen line is normally linked to the one above when the cursor moves onto the line by wrapping

from the right margin of the line above. Line linking can be disabled by setting the flag bit in location 248/$F8.

When the active display is switched, the screen editor SWAPPER routine [$CD2E] exchanges the contents of this area with the contents of locations 2666-2669/$0A6A-$0A6D, the storage area for the inactive display line link bitmap. Thus, the line link status is preserved when the screen is inactive.

Kernal File Tables

The following three ten-byte tables hold information on any currently open logical files. The three tables are parallel; the entry for a particular file will appear at the same position in all three tables. Location 152/$98 serves as an index to the next available entry in the tables. The fact that there are only ten bytes per table means that no more than ten logical files may be opened simultaneously.

866-875 $0362-$036B LATBL

Logical file number table for currently open files

When a logical file is opened, the OPEN routine [$EFBD] examines the contents of this table. A FILE OPEN error occurs if an existing file already uses the specified logical file number, and a TOO MANY FILES error occurs if ten files are already open. Otherwise, the logical file number for the file is stored in the next available entry in this table. When the Kernal CHKIN [$F106] or CKOUT [$F14C] routines are used to select a logical file for input or output, this table is searched for the specified logical file number. A FILE NOT OPEN error occurs if the file number is not found in the table. Otherwise, the corresponding device number and secondary address will be read from the respective tables. When a file is closed, the Kernal CLOSE routine [$F188] removes the file's entry from this table.

The Kernal includes a routine [LKUPLA, $F79D] to search for a particular file number in this table.

876-885 $036C-$0375 DNTBL

Device number table for currently open files

When a logical file is opened, the OPEN routine [$EFBD] stores the device number for the file in the next available entry in this table. When the Kernal CHKIN [$F106] or CKOUT [$F14C] routines are used to select a logical file for input or

output, the device number for the selected file will be read from this table. When a file is closed, the Kernal CLOSE routine [$F188] removes the file's entry from this table.

886-895 $0376-$037F SATBL

Secondary address table for currently open files
When a logical file is opened, the OPEN routine [$EFBD] will OR the specified secondary address for the file with the value 96/$60, then store the result in the next available entry in this table. When the Kernal CHKIN [$F106] or CKOUT [$F14C] routines are used to select a logical file for input or output, the secondary address for the selected file will be read from this table. When a file is closed, the Kernal CLOSE routine [$F188] removes the file's entry from this table.

The Kernal includes a routine [LKUPSA, $F786] to search for a particular secondary address in this table.

BASIC Working Storage

The remainder of page 3, locations 896-1023/$0380-$03FF, is used to hold BASIC character retrieval subroutines and to store values for a variety of BASIC routines. The subroutines in locations 896-980/$0380-$03D4 are copied here from locations 17017-17101/$4279-$42CD in BASIC ROM during the BASIC cold-start sequence. The routines are not reinitialized by RUN/STOP-RESTORE.

896-926 $0380-$039E CHRGET

Main BASIC character retrieval routine
This is BASIC'S primary routine for reading program text for interpretation and execution. The routine is designed to retrieve the next nonspace character from a BASIC line (in bank 0), and to return information about the type of character retrieved. The routine begins by incrementing the current address in the text pointer at locations 61-62/$3D-$3E. The system is set for the bank 0 configuration, the value at the location specified in the pointer is loaded into the accumulator, and the system is reset for the bank 14 configuration. The routine then performs a series of tests that will set the processor status register to reflect the type of character that was read. If a space character (code 32/$20) is read, the routine loops back to read another character (which is why spacing is not usually

significant in BASIC program lines). If the character is one of the numbers 0-9, the carry bit will be clear (carry will be set if the character is not a digit). If the character was a colon (:), BASIC'S end-of-statement marker, or a zero byte, BASIC'S end-of-line marker, the status register Z bit will be set; otherwise, the Z bit will be clear.

This routine has an alternate entry point at 902/$386, called CHRGOT, which retrieves and tests the current character, the one at the address currently in 61-62/$3D-$3E, without updating the pointer.

Since this routine is in RAM, it can be modified to change the way BASIC program text is read. Refer to Chapter 5 for details on how you can use this technique to add new commands to BASIC.

927-938 $039F-$03AA INDSUB_RAM0

Alternate routine for reading characters from program text

This routine retrieves a character from program text (bank 0). The value in the accumulator upon entry specifies the address of the zero-page pointer containing the base address, and the value in the Y register specifies the offset from this base address to the character to be read. The character will be in the accumulator upon return from the routine and the system will be left in the bank 14 configuration. BASIC ROM includes a collection of character retrieval subroutines (17102-17159/ $42CE-$4307) that make use of this routine.

939-950 $03AB-$03B6 INDSUB_RAM1

Alternate routine for reading characters from variable storage

This routine retrieves a character from the variable storage area (bank 1). The value in the accumulator upon entry specifies the address of the zero-page pointer containing the base address, and the value in the Y register specifies the offset from this base address to the character to be read. The character will be in the accumulator upon return from the routine and the system will be left in the alternate bank 14 configuration that includes block 1 RAM. BASIC ROM includes a collection of character retrieval subroutines (17102-17159/ $42CE-$4307) that make use of this routine.

951-959 $03B7-$03BF INDIN1_RAM1

Alternate routine to retrieve a character from variable storage
This routine retrieves a character from the variable storage area (bank 1) using locations 36-37/$24-$25 as a pointer and the contents of the Y register as an offset from the address in the pointer. The character will be in the accumulator upon return from the routine and the system will be left in the alternate bank 14 configuration that includes block 1 RAM.

960-968 $03C0-$03C8 INDIN2

Alternate routine to retrieve a character from program text
This routine retrieves a character from program text (bank 0) using locations 38-39/$26-$27 as a pointer and the contents of the Y register as an offset from the address in the pointer. The character will be in the accumulator upon return from the routine and the system will be left in the bank 14 configuration.

969-977 $03C9-$03Dl INDTXT

Alternate routine to retrieve current program text character
This routine retrieves the current program text character using 61-62/$3D-$3E as a pointer. The character will be in the accumulator upon return from the routine and the system will be left in the bank 14 configuration. The routine is similar to the CHRGOT entry into GHRGET, but without the tests for character type.

978-980 $03D2-$03D4 ZERO

Null descriptor
If the routine [$7AAF] which searches for a variable name in the variable table fails to find the name when called by EVAL [$7978] or POINTER [$82FA], the address of this area is returned as the variable descriptor address. This prevents variable table entries from being created if a variable name is first used in an expression argument or in the POINTER function. For example, if you use B$ = A$ or AD = POINTER(A$) when no variable A$ exists, no entry for A$ will be created. These three locations are all filled with the value 0/$00, copied here from ROM along with the preceding subroutines.

981 $03D5 CURRENT-BANK

Bank number for BASIC operations

The value in this location specifies the bank number used during BASIC routines which directly access memory. The value here doesn't affect the current system configuration—only the configuration that will be established for certain operations. The value here specifies the bank to which data will stored when the POKE statement is used, or from which data will be read when the PEEK statement is used. The value here determines the bank configuration in which the target address for a SYS statement will be seen. It also determines the bank for the address used in the WAIT statement. The value here determines the default bank for the BOOT, BLOAD, and BSAVE statements. It also determines the system bank affected by the STASH, FETCH, and SWAP statements.

The value here is initialized to 15/$0F during the BASIC cold-start sequence, so bank 15 is the default. The BANK statement can be used to change the value here.

982-985 $03D6-$03D9 TMPDES

Pointers for INSTR evaluation

These locations are used as working pointers for the INSTR statement routine [$99C1]. Locations 982-983/$03D6-$03D7 hold the address of the first string parameter and locations 984-985/$03D8-$03D9 hold the address of the second string.

986 $03DA FIN_BANK

String block flag

This location is used during the routine [$8D22] to convert character strings into floating-point values to indicate whether the string being converted resides in BASIC program text (block 0 RAM) or in the string pool {block 1 RAM).

987-990 $03DB-$03DE SAVSIZ

Temporary storage for SHAPE data

These locations are used during the SSHAPE routine [$642B] to hold coordinates of the area being saved, and during the SPRSAV routine [$76EC] to hold the descriptor of the first parameter value.

991 $03DF BITS

Floating-point overflow byte

This location is used for working storage while aligning floating-point values for mathematical operations, or for converting floating-point values into integers. The value here is initialized to O/$0O during the BASIC cold-start sequence, and will be reset to that value during CLR or warm start.

992-993 $03E0-$03El SPRTMP

Temporary pointer storage

These locations are used during the SPRSAV routine [$76EC] to temporarily preserve the current value in the CHRGET pointer (61-62/$3D-$3E).

994 $O3E2 FG_BG

Standard bitmap color fill value

This location holds the color memory fill pattern for standard bitmapped mode. When the SCNCLR routine [$6A79] is used to clear the standard bitmapped (GRAPHIC 1 or GRAPHIC 2) display, all locations in the video matrix area will be filled with the value here. (The SCNCLR routine is also used when the clear parameter is specified in a GRAPHIC statement.) In standard bitmapped mode, the video matrix area holds foreground and background color information, so the value here determines the default foreground and background colors for all screen positions after the screen is cleared.

Whenever the BASIC graphics routines are used to draw anything on the standard bitmapped display, the value in this location will determine the color of the line drawn. If color source 0 was specified for the line, the value in the lower four bits here will be stored in the lower four bits of the video matrix locations corresponding to the line's location in the bitmap. If color source 1 is specified, the value in the upper four bits here will be stored in the upper four bits of the video matrix locations corresponding to the line's location in the bitmap.

The value in this location is updated whenever the COLOR statement [$69E2] is executed. The high four bits here are set to the value in the lower four bits of the foreground color in location 134/$86. The lower four bits here are set to the value in the lower four bits of the VIC background color register at 53281/$D021. These locations are initialized during

the BASIC cold-start sequence to the default foreground and background colors, so this location will initially hold 219/$DB, for a light green foreground and dark gray background. The value here is not affected by RUN/STOP-RESTORE.

995 $03E3 FG-JMC1

Multicolor bitmap color fill value

This location holds the color memory fill pattern for multicolor bitmapped mode. When the SCNCLR routine [$6A79] is used to clear the multicolor bitmapped (GRAPHIC 3 or GRAPHIC 4) display, all locations in the video matrix area will be filled with the value here. (The SCNCLR routine is also used when the clear parameter is specified in a GRAPHIC statement.) In multicolor bitmapped mode, the video matrix area holds color information for pixels with %01 and %10 bit patterns, so the value here determines the default colors for those pixel patterns in all screen positions after the screen is cleared.

For the BASIC graphics routines that draw on the multicolor bitmapped display, the value in this location will determine the color of any lines drawn using color sources 1 or 2. If color source 1 was specified for the line, the value in the upper four bits here will be stored in the upper four bits of the video matrix locations corresponding to the line's location in the bitmap. If color source 2 was specified, the value in the lower four bits here will be stored in the lower four bits of the video matrix locations corresponding to the line's location in the bitmap.

The value in these locations can be changed with the COLOR statement [$69E2]. The high four bits here are set to the value in the lower four bits of the foreground color in location 134/$86. The lower four bits here are set to the multicolor 1 value in the lower four bits of location 132/$84. The location is initialized during the BASIC cold-start sequence to the default multicolor pixel colors for %01 and %10 patterns, so this location will initially hold 2O9/$D1, for light green %01 pixels and white %1O pixels. The value here is not affected by RUN/STOP-RESTORE.

996-1007 $03E4-$03EF Unused

The locations in this range are not used by any system ROM routine, and are thus available for your own programming.

1008-1020 $03F0-$03FC DMA

DMA_CALL execution routine

This area holds the RAM-resident portion of the Kernal DMA_CALL routine [$F7A5]. The routine is copied here from Kernal ROM during the reset sequence. It is designed to initiate a DMA (direct memory access) command to the REC (RAM expansion controller) chip in an installed memory expansion module. The routine loads the current memory configuration register contents, then stores the contents of the Y register in the REC command register address (57089/$DF01) and stores the contents of the accumulator in the MMU memory configuration register (65280/$FF00). If the REC is configured in its default state, storing the value in the MMU register should trigger the specified REC operation. See Chapter 8 for more information about the REC chip. Upon completion of the operation, the original memory configuration register setting will be restored.

1021-1023 $03FD-$03FF Unused

The locations in this range are not used by any system ROM routine, and are thus available for your own programming.

Bank 0 Working Storage

VIC Default Screen Memory

1024-2047/$0400-$07FF

This IK area is the default location for the VIC chip's screen memory in character (GRAPHIC 0) mode. It's not used by the system for any other purpose. Screen memory can be relocated to any other IK block in RAM by changing the appropriate bits in the registers at 53272/$D018 and 56576/$DD00. For example, screen memory for the system bitmapped modes (GRAPHIC 1 or GRAPHIC 3) is located at 7168-7423/ $1COO-$1CFR However, the screen editor CINT routine [$C07B] and Kernal IOINIT routine [$E109], both included in the reset and RUN/STOP-RESTORE sequences, will reset the registers to have screen memory at this default area. Conveniently, this is the same area used for default screen memory in the Commodore 64.

When used for screen memory, the first 1000 locations of this area (1024-2023/$0400-$07E7) correspond to the 1000 character positions of the VIC chip's 40-column X 25-row screen display. The value in each screen memory location determines what will be displayed in the corresponding position on the screen. The screen memory values, called screen codes, are used as indexes into character pattern memory. Any character pattern can be displayed at any screen location by storing the appropriate screen code in the appropriate screen memory location. Appendix C lists the standard character pattern for each screen code. Clearing the screen fills these 1000 memory locations with 32/$20, the screen code for the space character.

While this area is used as screen memory, the highest eight locations (addresses 2040-2047/$07F8-$07FF) are used to hold definition pointers for the eight sprites supported by the VIC chip. The pattern definition for a sprite requires 64 bytes, so there is room within a 16K VIC video bank for 16384 / 64, or 256 sprite definitions. These pointer locations each

hold a value between 0-255/$00-$FF indicating which 64-byte area within the current video bank will be used to hold the pattern definition for the corresponding sprites. Changing the shape of a sprite is as simple as changing the corresponding sprite pointer to select a new pattern definition. To find the address specified by the pointer value, multiply the value by 64/$40 and add the base address for the video bank. The default pointer values, established by the BASIC cold start initialization subroutine [$4045], are as follows:

Location	Sprite	Default pointer value
2040/S07F8	0	56/$38
2041/$07F9	1	57/$39
2042/$07FA	2	58/$3A
2043/$07FB	3	59/$3B
2044/S07FC	4	60/$3C
2045/S07FD	5	61/$3D
2046/$07FE	6	62/$3E
2047/$07FF	7	63/$3F

The default values point to the eight sprite definition areas at 3584-4095/$0E00-$OFFF. While BASIC initializes each pointer to a different definition area, this is not mandatory. For example, if you want all eight sprites to have the same shape you can just design one sprite pattern and store the pointer to that pattern in all eight locations. BASIC doesn't have any statement specifically for changing pointers, so you'll have to use POKE to change the values here. Because the default definition area only has room for eight sprites, you'll have to use some other area of free memory if you want to use more than eight sprite shapes. All sprite pattern definitions must lie within the current 16K video bank. For the default video bank (0-16383/$0000-$3FFF in block 0 RAM), the free space at 4864-7167/$1300-$lBFF can be used.

Remember that the sprite pointers are dependent on the current screen memory block, and aren't an absolute feature of these locations. The sprite pointers always appear at an offset of 1016-1023/$03F8-$03FF bytes beyond the specified starting address of screen memory. For example, when one of the bitmapped modes is selected, screen memory (in that case used for color information and usually referred to as the video matrix) moves to 7168-8191/$1COO-$1FFF, so the sprite pointers for the default bitmapped screen are instead located at 8184-8191/$1FF8-$1FFR Thus, in split-screen displays (GRAPHIC 2 or

GRAPHIC 4) it is possible for the same sprite to have different shapes in the text and bitmapped portions of the display. The SCNCLR routine (also called when you add an extra ,1 to the GRAPHIC statement) copies the text screen sprite pointer values from locations 2040-2047/$07F8-$07FF to the bitmapped screen sprite pointers at 8184-8191/$1FF8-$1FFF, but there's no ROM routine to perform a copy in the opposite direction—from bitmapped screen sprite pointers to text screen sprite pointers—so your text screen pointers should be preserved even if you change the pointers for sprites displayed on the bitmapped screen.

The remaining 16 locations in this area, addresses 2024-2039/$07E8-$07F7, are unused by any system routine. They are not affected by clearing the screen or changing sprite pointers, or by reset or RUN/STOP-RE STORE. Thus, they are available for your own programming uses.

BASIC Runtime Stack

2048-2559/$0800-$09FF

This 512-byte area is used by BASIC for its runtime stack. When you use the machine language JSR instruction to call a subroutine, there must be some way to record the address to return to upon completion of the subroutine. As explained in the entry for locations 256-511/$D1OO-$O1FF, the return address is placed in a special area of memory called the processor stack. BASIC'S GOSUB statement and looping statements like FOR and DO also need some place to store the address to return to upon completion of the subroutine or loop. In earlier versions of Commodore BASIC, this information was also kept in the processor stack. However, only 256 bytes of storage space are available in the processor stack, and BASIC allows only a portion of that to be used while it is in control. This limits the level to which loops and subroutines can be nested. For example, each FOR-NEXT loop requires 18 bytes of stack space, so the Commodore 64's BASIC 2.0 allows loops to be nested only nine levels deep. Because the more complex memory banking routines in the 128 require more machine language subroutine calls, the 128's BASIC 7.0 would allow even fewer levels of nesting if it used this same system. However, BASIC 7.0 instead stores the information for GOSUB, FOR, and DO in this totally separate runtime stack area.

Because this storage area does not have to be shared with processor return addresses, as is the case with the processor stack, the entire 512-byte space is available. Thus, you can have up to 28 nested FOR-NEXT loops (each requires an 18-byte entry on the stack), or up to 102 nested DO loops or GOSUB subroutines (each of which requires a 5-byte entry), or any combination thereof.

Using this runtime stack requires slightly more software overhead than using the processor stack. The 8502 processor has an internal stack pointer register that indicates the position of the next available position in the processor stack, and it also has PHA and PLA instructions specifically for adding and removing instructions from this stack. None of this is handled automatically for the BASIC runtime stack; instead, the routines which use the stack must explicitly update locations 125-126/ $7D-$7E, the runtime stack pointer. The GOSUB [$59CF], FOR [$5DF9], and DO [$5FE0] statement routines add entries to the stack, and the RETURN [$5262], NEXT [$57F4], and LOOP [$608A] statement routines can remove entries from the stack. The COLLISION statement [$7164] also causes the equivalent of a GOSUB entry to be placed on the stack when a collision of the specified type occurs.

This area is not used by the system for any purpose other than the BASIC stack, so this entire area is available for use by machine language programs that don't require BASIC.

Kernal and Screen Editor Working Storage

2560-2687/$0A00-$0A7F

2560-2561 $OA00-$0AOl SYSTEM-VECTOR

BASIC restart vector

This pair of locations contains the address of the routine that will be used to restart BASIC. The RAMTAS routine [$E093], part of the reset sequence, puts the value 163 84/$4 000 here—the address of the BASIC cold start routine. Unless the Commodore or RUN/STOP keys are held down, the RESET routine [$EO0O] ends with a JMP ($0A00) to cold start BASIC. One of the final steps in the BASIC cold start routine is to

change the value here to 16387/$4003—the address of the BASIC warm start routine.

The RUN/STOP-RESTORE sequence in the NMI handling routine [$FA40] ends with a JMP ($0A00). Because of the cold start's routine initialization, this will normally cause a warm start of BASIC. However, you can make RUN/STOP-RESTORE cause a jump to another routine by changing the value in these locations to point to the address of the new routine. The only restriction is that the target routine must be visible in the bank 15 configuration, since that is how memory is arranged when the JMP is executed.

The monitor X command routine [$B0E3] also performs a JMP ($0A00), so the value in these locations determines the address of the routine which will be executed when you use that command to exit the built-in machine language monitor.

2562 $0A02 DEJAVU

Memory initialization status flag

This location is used to indicate whether the RAMTAS routine has been performed. If the RESET routine [$E000] detects that the RUN/STOP key is being held down, indicating that the reset sequence should end by entering the monitor rather than BASIC, then the value here will be tested. If this location contains the value 165/$A5, the RAMTAS routine will be omitted from this reset sequence. The routine will hold a random value when the computer is first turned on, but the first call of the RAMTAS routine [$E093] will initialize this location to 165/$A5. Thus, once RAMTAS has been performed at least once, the test of this flag location can be used to prevent its being performed again when entering the monitor after a reset. This allows you to preserve the contents of zero page, normally cleared by RAMTAS during the reset.

2563 $0A03 PALNTS

PAL/NTSC flag

The IOINIT routine [SE109], part of the RESET sequence, checks the number of scan lines produced by the VIC chip to determine whether the 128 is using a NTSC (North American) or PAL (European/British) video system. This location is set to reflect the result of that test: to 0/$00 for NTSC systems or 255/$FF for PAL systems. Later routines that initialize the

video chips and timers can then adjust the default settings accordingly. This eliminates the need for different versions of the Kernal ROM for different countries.

2564 $0A04 INIT_STATUS

System initialization status flag
This location is initialized to 0/$00 near the beginning of the Kernal RESET routine [$EOOOj. Bits are then set to %1 by later routines to indicate that certain initialization steps have been performed.

Bit 0: This bit is set to %1 during the BASIC cold start routine [$4023] to indicate that the cold start has been performed. The IRQ handling routine [$FA65] checks this bit and calls the BASIC IRQ routine [$A84D] only if the bit is %1. The BASIC IRQ routine, which handles sprite movement sprite collision detection, and sound generation, copies the contents of a number of shadow locations into VIC and SID hardware registers. One way to turn off this interrupt routine and gain direct access to the hardware registers is to set this bit to %0.

Bits 1-5: Unused.

Bit 6: This bit is tested during the screen editor initialization (CINT) routine [$C07B] to determine whether the keyboard table pointers and function key definitions need to be initialized. If the bit is %0, the default pointer values and key definitions will be copied from ROM into the proper areas of RAM; then this bit will be set to %1. While this bit is %1, the pointer and function key initialization portion of the routine will be skipped. CINT is part of both the reset and RUN/STOP-RESTORE sequences, but the pointers and key definitions are normally initialized only during the reset sequence, which resets this bit to %0 before calling CINT. Custom keyboard table pointers and function key definitions are usually preserved during RUN/STOP-RESTORE, which does not affect this bit.

Bit 7: This bit is tested during the IOINIT routine [$E109] to determine whether the 80-column (VDC) character set needs to be initialized. If the bit is %0, the INIT80 routine [$CE0C] will be called to copy the standard character patterns from ROM into the VDC chip's private RAM; then this bit will be set to %1. While this bit is %1, the character initialization portion of the routine will be skipped. IOINT is part of both the reset and RUN/STOP-RESTORE sequences, but the character

patterns are normally initialized only during the reset sequence, which resets this bit to %0 before calling IOINT. Custom 80-column characters are usually preserved during RUN/STOP-RESTORE, which does not affect this bit.

2565-2566 SO A05-S0A06 MEMSTR

Kernal MEMBOT pointer

This pair of locations holds the default value of the lowest memory address available in block 0 RAM. The value here can be read or changed using the Kernal MEMBOT routine [$F772], which has a Kernal jump table entry at 65436/$FF9C. The RAMTAS routine, part of the RESET sequence, calls MEMBOT to initialize these locations to 7168/$1COO. However, the value here is not used by any other system routine, so changing this value will not affect system operation in any way. This is a change from the Commodore 64, where the value in the MEMSTR pointer is used to establish the lowest address available of BASIC. In the 128, the start-of-BASIC pointer is always initialized to 7169/$1CO1, regardless of the value here.

2567-2568 $0AO7-$0A08 MEMSIZ

Kernal MEMTOP pointer

This pair of locations holds the default value of the highest memory address available in block 0 RAM. The value here can be read or changed using the Kernal MEMTOP routine [$F763], which has a Kernal jump table entry at 65433/$FF99. The RAMTAS routine, part of the RESET sequence, calls MEMTOP to initialize these locations to 65280/$FF00. However, the value here is not used by any other system routine, so changing this value will not affect the system operation in any way. This is a change from the Commodore 64, where the value in the MEMSIZ pointer is used to establish the highest address available for BASIC variable storage. In the 128, the top-of-BASlC pointers are always initialized to 65280/$FF0O/ regardless of the value here.

2569-2570 $OA09-$OAOA IRQTMP

Temporary storage for IIRQ vector during tape operations

These locations are used for temporary storage of the address value in the IIRQ vector at 788-789/$0314-$0315 during tape operations. The tape routines stash the current IIRQ address

here, then substitute the address of the IRQ service routine to handle the tape operation. Upon completion of the operation, the original address stored here will be restored to the IIRQ vector.

Location 2570/$0A0A is also used as a flag to indicate whether or not a tape IRQ routine is active. That location is initialized to 0/$00 by the IOINIT routine, part of the RESET sequence, and will also be reset to that value upon completion of the tape operation. Thus, a nonzero value in the flag location indicates that a tape interrupt routine is active.

2571 $0AOB CASTON

CIA #1 control register A log

This location is used to record the status of CIA #1 control register A (56334/$DC0E) during tape operations.

2572 $0A0C

CIA #1 interrupt control register log

This location is used to record the status of the CIA #1 interrupt control register (56333/$DC0D) during tape operations.

2573 $0A0D

CIA #1 timer A status log

The CIA #1 control register A log value from 25 71 /$0AOB is stored here during certain tape operations to preserve the timer A status.

2574 $0A0E TIMOUT

IEEE timeout flag

When the VIC-20 was introduced, its Kemal included a jump table entry (SETTMO, at 65442/$FFA2) to support a proposed IEEE bus interface. The IEEE bus is the parallel data bus used by Commodore's original PET/CBM models for communications with peripheral devices. The interface was never introduced, but the Kernals of all subsequent Commodore models have slavishly included the SETTMO jump table entry. In the 128, the SETTMO routine [$F75F] does nothing more than store the accumulator contents in this location. This location is

not used by any other 128 routine, and is provided strictly to maintain Kernal jump table compatibility with previous Commodore models.

2575 $0A0F ENABL

RS-232 activity flag

This location is used during the RS-232 routines to record the value in the CIA #2 interrupt control register (56589/$DD0D). CIA #2 interrupt requests generate the NMI interrupts that drive RS-232 transmission and reception. While the CIA #2 interrupts for RS-232 are disabled, this location will be set to 0/$00. When bits are set in the CIA #2 interrupt control register to enable RS-232 operations, the corresponding bits are also set in this location. If any of the following bits is set to %1, the corresponding interrupt is enabled:

Bit	Interrupt source	RS-232 activity
0	Timer A	bits being transmitted
1	Timer B	bits being received
4	FLAG line	waiting for start bit to be received

This location is initialized to 0/$00 during the IOINIT routine [$E109], part of the reset and RUN/STOP-RE STORE sequences.

2576 $0A10 M51CTR

RS-232 control register

This location controls some of the operating characteristics of the RS-232 interface. When a file is opened to device 2, the first character of the filename is copied here. Although RS-232 communications in the 128 are managed by software, the bits of this location are defined to simulate the control register of a 6551 UART chip, a hardware device for controlling serial communications. The bits are used as follows:

Bits 0-3: These bits determine the baud rate for both transmission and reception—the rate (in bits per second) at which bits will be sent or received. Valid settings are as follows:

Bits				Bit	Baud rate
3	2	1	0	Value	
0	0	0	0	0/$0	user defined
0	0	0	1	1/*1	50 baud
0	0	1	0	2/$2	75 baud
0	0	1	1	3/$3	110 baud
0	1	0	0	4/$4	134.5 baud
0	1	0	1	5/$5	150 baud
0	1	1	0	6/$6	300 baud
0	1	1	1	7/%7	600 baud
1	0	0	0	8/$8	1200 baud
1	0	0	1	9/$9	1800 baud
1	0	1	0	10/$A	2400 baud

When the user-defined rate is selected, the baud rate is determined by the value in locations 2578-2579/$0A12-$0A13. The remaining possible bit patterns, %1011-%llll, result in invalid baud rates.

Bit 4: Unused.

Bits 5-6: These bits determine the number of data bits in each character sent or received (sometimes referred to as the word size). The total character length will also include a start bit, possibly a parity bit, and one or more stop bits.

Bits		Bit	Number of
6	5	value	data bits
0	0	0/$00	8 data bits
0	1	32/$20	7 data bits
1	0	64/$40	6 data bits
1	1	96/S60	5 data bits

Bit 7: This bit determines the number of stop bits in each character. Stop bits are %1 bits added to the end of the character. They represent the minimum amount of time the communications line will remain at the low (%1 bit) level before the next start bit can be sent or received.

Bit 7	Bit value	Number of stop bits
0	0/$00	1 stop bit
1	128/S80	2 stop bits

2577 $OA11 M51CDR

RS-232 command register

This location controls some of the operating characteristics of the RS-232 interface. When a file is opened to device 2, the

second character of the filename, if any, is copied here. Although RS-232 communications in the 128 are managed by software, the bits of this location are defined to simulate the command register of a 6551 UART chip, a hardware device for controlling serial communications. The bits are used as follows:

Bit 0: This bit controls the handshaking mode for RS-232 transmission and reception. The RS-232 interface consists of three primary signal lines—transmitted data, received data, and ground—plus a number of supplementary control lines—data set ready (DSR), data terminal ready (DTR), ready to send (RTS), and clear to send (CTS). The control lines are called handshaking lines because they allow the sending and receiving units to exchange signals (handshakes) indicating whether data is being successfully transmitted and received. The 128's RS-232 software interface can operate in two different modes: 3-line, where none of the handshaking lines are used, and x-line, where all of the handshaking lines are used. These bits control the interface mode as follows:

Bit 0	Interface mode
0	3-line interface {no handshaking)
1	x-line interface {full handshaking)

For 3-line mode, the output handshaking lines (DTR and RTS) will be held at a constant high (+ 5 volts) level. The input handshaking lines {DSR and CTS) will be ignored.

Bits 1-3: Unused.

Bit 4: For unknown reasons, Commodore literature continues to indicate that this bit controls the duplex mode of the RS-232 interface. The bit is supposed to select full duplex when set to %0 or half duplex when set to %1. However, this bit is not checked by any RS-232 routine, and its setting has no effect on the operation of the interface.

Duplex is often confused with local echo. A full-duplex interface can simultaneously send and receive data, while a half-duplex interface can send data and receive data, but not both at the same time. The 128's RS-232 interface always operates in full-duplex mode. In casual usage, however, duplex is often used to describe whether or not the system echoes back the characters it receives. In remote echo mode {incorrectly referred to as full duplex), the system displays only characters received from the remote unit (the one being called). The assumption is that the remote unit will send back an "echo" of

each character it receives from the system. In local echo mode (incorrectly called half duplex), the system displays the characters it sends as well as the ones it receives. The assumption in this case is that the remote unit will not echo the characters it receives.

Bits 5-7: This bit controls the parity generated for transmitted characters and the parity tested for in received characters. Parity is a simple method of detecting some errors in data transmission. A parity bit can be added between the data and stop bits in the character. The value of the parity bit is selected to make the total number of %1 bits in the character (not counting stop bits) either even or odd. The receiving unit can then count the number of %1 bits in the received character to determine if bits have been garbled in transmission. Parity checking did not work properly in the original versions of the Commodore 64 Kernal ROM, but that problem has been corrected in the 128's Kernal (and in the version of 64 Kernal ROM for the 128's 64 mode). Possible parity selections are as follows:

Bits			Parity selection
7	6	5	
x	x	0	parity not used
0	0	1	odd parity
Oil			even parity
10		1	mark parity
1	1	1	space parity

If bit 5 is %0, no parity bit will be generated in transmitted characters and the system will expect incoming characters to have no parity bit. This selection is common when a word size of eight data bits per character is used. Odd parity means that a parity bit will be generated for each transmitted character such that the character will have an odd total number of %1 bits (not counting the stop bits). When even parity is selected, the parity bit will be set to make the total number of %1 bits in the character even. For either even or odd parity, the number of %1 bits in each character received will be counted and compared against the parity selection. If the number does not match the specified parity type, the error will be indicated by setting bit 0 of the status register location (2580/ $0A14) to %1. Mark and space parity are alternate forms of no parity. When mark parity is selected, the parity bit for each transmitted character will always be set to %1, and the parity

bit for each received character will be ignored. When space parity is selected, the transmitted parity bit will always be %0 and the received parity bit will be ignored.

2578-2579 $0A12-$0A13 M51AJB

RS-232 baud-rate factor

The value in these locations determines the baud-rate timing factor. When a file is opened to device 2, the third and fourth characters of the filename are copied here (if the filename has that many characters). However, the filename characters are meaningful only if a user-defined baud rate has been specified—if bits 0-3 of the first character of the filename (copied into 2576/$0A10) are %0000. In that case, the value in these locations specifies the baud rate according to the following formula:

baud rate = clock frequency/(2 * (rate factor + 100))

For the standard baud-rate settings, the Kernal RS-232 OPEN routine copies the proper rate factor into these locations from tables in Kernal ROM (59472-59491/$E850-$E863 for NTSC systems or 59492-59511/$E864-$E877 for PAL systems). The two separate tables are required because the different video systems use different clock frequencies.

There's rarely a need to specify a custom baud rate, since the standard settings encompass all standard rates that the 128 can support. (The 128 cannot handle RS-232 communications faster than 2400 baud, so don't try to specify a faster rate.) However, should you ever want to do so, the formula for the rate factor is as follows:

rate factor = {(clock frequency / desired baud rate)/ 2) — 100

The clock-frequency value is 1022730 for NTSC (North American) systems or 985250 for PAL (European) systems. The low byte of the resulting factor should be stored in 2578/$0A12 and the high byte in 2579/$0A13.

2580 $0A14 RSSTAT

RS-232 status register

Although RS-232 communications in the 128 are managed by software, the bits of this location are defined to simulate the status register of a 6551 UART chip, a hardware device for controlling serial communications.

It is possible to read the value here directly, but this location can also be read using the Kernal READSS routine [$F744] if the current device number in location 186/$BA is 2 (for RS-232). The READSS routine also has a Kernal jump table entry at 65463/$FFB7. From BASIC, the reserved variable ST will reflect the value in this location as long as the current device number is 2. This location is initialized to 0/$00 each time the Kernal OPEN routine is called to open a file to device 2, the RS-232 interface. The value here is also reset to zero after each call to the READSS routine when device 2 is active, including each reference to the ST variable in a BASIC program.

Bit 0: This bit is the parity-error indicator. It is used only when either even or odd parity is selected, and is relevant only to received characters. The bit is set to %1 whenever a character is received for which the calculated total of %1 bits received for the character does not match the specified parity selection.

Bit 1: This bit is the framing-error indicator. The bit is set to %1 when a framing error occurs—when no stop bits are found following the specified number of data and parity bits.

Bit 2: This bit is the receiver buffer-overflow indicator. It will be set to %1 when a character is received after the RS-232 input buffer at 3072/$0C00 is already full.

Bit 3: This bit is the receiver buffer-empty indicator. It will be set to %1 whenever there are no characters waiting to be read from the input buffer. This bit should be tested before each attempt to read characters from the RS-232 interface.

Bit 4: This bit is the CTS-missing error indicator. It is used only when x-line handshaking is selected. The bit will be set to %1 if the CTS (clear to send) input line drops to a low (0 volts) state while data is being transmitted. When x-line handshaking is used, the external device connected to the interface is expected to hold the CTS line at a high (+ 5 volts) state. If the line goes low, it is taken as an indication that the external device is not ready to receive data, so transmission is suspended until CTS goes high.

Bit 5: Unused. This bit should always be %0 when read.

Bit 6: This bit is the DSR-missing error indicator. It is used only when x-line handshaking is selected. The bit will be set to %1 if the DSR (data set ready) signal input line drops to a

low (0 volts) state during either transmission or reception of characters. When x-line handshaking is used, the external device connected to the interface is expected to hold the DSR line high (+ 5 volts). If the line goes low, it is taken as an indication that nothing is connected to the interface.

Bit 7: This bit is the break indicator. A break occurs when, during reception of characters, a byte is received consisting of all %0 bits not followed by stop bits (which are always %1). In other words, a break occurs if the received data signal line is held at the %0 bit (+ 5 volt) level for longer than the time required to receive a character.

2581 S0A15 BITNUM

RS-232 bit count

This location will hold the number of bits prior to the parity and stop bits for each character received or transmitted. The location is initialized during the RS-232 OPEN routine [$F040] to the number of data bits (specified in bits 5-6 of 2576/ $0A10) plus 1. For transmission, the value here is copied into location 180/$B4, the countdown of bits to send. For reception, the value here is copied into location 168/$A8, the countdown of bits remaining to be received.

2582-2583 $0A16-$0A17 BAUDOF

RS-232 baud-rate timing constant

Commodore's insistence on providing an exact software emulation of the 6551 UART chip leads to some odd software gyrations. The baud-rate timing factor specified in locations 2578-2579/$0A12-$0A13 must be converted back into an absolute timing value. The Kernal OPEN routine for RS-232 performs the following calculation:

timing constant = (rate factor * 2) — 200

Given the formula for rate factor, this is equivalent to:

timing constant = clock frequency / baud rate

This yields the number of system clock cycles required to send or receive each bit at the specified baud rate. The resulting value is stored in these locations. When transmission or reception is initiated, the value here is copied into one of the CIA #2 timers. This determines the time between the NMI interrupts that drive the transmission or reception of bits.

2584 S0A18 RIDBE

Index to first character in RS-232 input buffer

This location holds the offset from the start of the RS-232 input buffer to the position where the next character received will be stored. The input buffer normally begins at location 3072/$0C00. The value here is incremented before each received character is added to the buffer, unless incrementing would make the value here equal to the value in location 2585/$0A19. In that case, a buffer overflow has occurred {more characters have been received than the buffer can hold), so bit 2 of the status location (2580/S0A14) is set to %1.

2585 $0A19 **RIDBS**

Index to last character in RS-232 input buffer

This location holds the offset from the start of the RS-232 input buffer to the position of the next character waiting to be removed from the buffer. The buffer normally begins at 3072/$0C00. The value here is incremented after each character is retrieved from the buffer. When the value here equals the value in location 2584/$0A18, all characters have been removed and the buffer is empty. In this case, bit 3 of the status location (2580/$0A14) will be set to %1.

2586 S0A1A **RODBS**

Index to first character in RS-232 output buffer

This location holds the offset from the start of the RS-232 output buffer to the position of the next character awaiting transmission. The output buffer normally begins at 3328/$0D00. The value here is incremented after each character is removed from the buffer for transmission. When the value here equals the value in location 2587/$0AlB, all characters awaiting transmission have been sent and the buffer is empty.

2587 $OA1B **RODBE**

Index to last character in RS-232 output buffer

This location holds the offset from the start of the RS-232 output buffer to the position where the next character will be added to await transmission. The output buffer normally begins at 3328/S0D00. The value here is incremented before each character is added to the buffer, unless the incrementing would make the value here equal the value in 2586/$0AlA.

In that case, the buffer is already full, and some characters must be transmitted before any more can be added to the buffer.

2588 $OA1C SERIAL

Fast serial mode flag

This location is used to indicate whether the currently specified serial bus device, such as the 1571 disk drive, is capable of fast serial communications. The location is initialized to 0/$00, the value for standard (slow) communications, during the IOINIT routine [$E109]. The routines that handle the serial bus TALK and LISTEN commands attempt a fast serial handshake. If the external device responds properly, bits 6 and 7 of this location will be set to %1. Bit 7 indicates that the external device is capable of fast transmission and reception of individual bytes. The bit is reset to %0 during the Kernal routines that handle the serial bus UNTALK and UNLISTEN routines. Bit 6 indicates that the system is capable of high-speed burst mode loading.

2589-2591 $OA1D-$OA1F TIMER

Software jiffy timer

The three-byte value in these locations is decremented 60 times per second by the Kernal UDTIM routine [$F5F8], part of the IRQ sequence. Thus, these locations function in a manner opposite that of the jiffy clock at 160-l62/$A0-$A2, which is incremented 60 times per second by UDTIM. The order of bytes here is the opposite of the order of bytes in the jiffy clock: $OA1D is the low byte and $OA1F is the high byte.

Since the countdown for this timer is handled automatically during the IRQ, it is useful for many timing applications. The way to use this timer is to load the locations with the value in jiffies (1/60-second intervals) for the desired delay period, then test for a value of $FF in location 2591/$OA1F. That location will contain $FF after the three-byte value rolls over from $000000 to $FFFFFF at the end of the countdown. The highest allowable initial value when using this scheme is $FF0000, which corresponds to 16,711,680 jiffies—a little over three days.

There is one caution in using this location from BASIC. The SLEEP statement routine [$6BD7] uses this timer for its delay countdown, so any use of the SLEEP statement will overwrite any values you may have stored in these locations.

2592 $0A20 XMAX

Maximum number of keys in the keyboard buffer

The value in these locations determines the maximum number of characters that can be held in the keyboard buffer pending processing. The value is initialized to 10/$OA—the full length of the standard keyboard buffer at 842-851/$034A-$0353—by the CINT screen editor initialization routine [$C07B], part of the reset sequence. During the SCNKEY routine [$C55D], the value here is compared against the value in location 208/$D0, the count of characters currently in the buffer, to determine if there is room in the buffer to record another keypress.

You can reduce the value here to decrease the number of unprocessed keypresses that can accumulate in the buffer. However, you should not increase the value above 10, as this will cause overflow from the buffer to overwrite the tab stop table at 852-861/$0354-$035D.

2593 $OA21 PAUSE

Scroll pause flag

This location is used to pause printing. During the screen BSOUT routine [$C72D], the value here is tested. If it is non-zero, the routine will wait indefinitely for the location to be reset to zero. The value is initialized to 0/$00 by the CINT routine [$C07B]. To implement the pause feature, the SCNKEY routine [$C55D], part of the system IRQ sequence, sets this location to 13/$0D when either the NO SCROLL or CONTROL-S keys are pressed, and resets the location to 0/$00 when the next key is pressed.

2594 $OA22 RPTFLG

Key repeat flag

The value here determines which keys, if any, will repeat if held down. If bit 7 of this location is %1 (value 128/S80), all keys repeat. If bit 6 is %1 (value 64/$40), no keys repeat. Otherwise, only the cursor, space, and INST/DEL keys repeat. This location is initialized to 128/S80—all keys repeat—by the screen editor CINT routine [$C07B], This is different from the Commodore 64, where the default value is 0/$00—only cursor, space, and INST/DEL repeating.

2595 $0A23 KOUNT

Countdown between key repeats

This location is used as a counter to establish the delay between repeats when a key is held down. Once repeating has begun, indicated by a value of 0/S00 in location 2596/$0A24, the value here will be decremented on each pass through the SCNKEY routine [$C55D] as long as the same key is held down. Each time the count reaches zero, the key is repeated and this location is reinitialized to 4/$04. This results in a key-repeat rate of 15 times per second. The starting value of 4 for this countdown is loaded from ROM in the SCNKEY routine, and thus cannot be changed, so the delay period between repeats is not programmable.

2596 $0A24 DELAY

Countdown until key repeating begins

This location is used as a counter to establish the delay before repeating begins when a key is held down. (Location 2594/$0A22 controls which keys, if any, will repeat if held down.) If the scan code of the current key is the same as the scan code detected on the last pass through the SCNKEY routine [$C55D], the value here will be decremented. When the count reaches zero, repeating can begin at the rate determined by location 2595/$0A23. When the key is released, this location is reinitialized to 16/$10. This results in a delay before repeating of about 1/4 second. The starting value of 16 is loaded from ROM in the SCNKEY routine, and thus cannot be changed, so the delay before repeating begins is not programmable.

2597 $0A25

Delay between case-switching repeats

This location is used to provide a delay between character case switches when the SHIFT-Commodore key combination is held down. This location isn't a countdown. Rather, it is initialized to 128/$80; then the value is shifted one bit to the right on each pass through the SCNKEY routine until it becomes zero. This provides a delay of about 1/8 second.

2598 $0A26 BLNON

Cursor blink flag

Bit 6 of this location controls whether the cursor on the 40-column (VIC) screen will blink. When the bit is %0, the cursor blinks at a rate determined by location 26O0/S0A28. When the bit is %1, the cursor will be a solid, unblinking block. This location is initialized to 0/$00 during the CINT screen editor initialization routine [$C07B], so the default cursor will be blinking. The bit can be set to %1 using the ESC E key sequence, and cleared to %0 using ESC F.

Bit 7 of this location indicates the blink phase of the cursor on the 40-column (VIC) screen. When the bit is %0, the character at the cursor position is in its original state. When the bit is %1, the character is reversed to provide the cursor blink effect.

2599 $0A27 BLNSW

Cursor enable flag

This location controls whether a cursor will be present on the 40-column (VIC) screen. The cursor will be enabled when the value here is zero and disabled when this location contains any nonzero value. This location can be used to enable the cursor when it is normally turned off. For example, the following statement provides a cursor at the prompt (GET and GETKEY don't normally provide a cursor):

```
300 POKE 2599,0: PRINT"PRESS A KEY: ";: GETKEY K$
```

2600 $0A28 BLNCT

Cursor blink countdown

This location determines the delay between cursor blinks for the 40-column (VIC) screen. Bit 6 of location 2598/$0A26 must be %0 to enable cursor blinking. The value here is decremented on each pass through the screen editor IRQ routine. Whenever the value reaches zero, the blink phase of the cursor changes and bit 7 of the screen code at the cursor position is toggled. This reverses the character at thai position. The value here is reinitialized to 20/$ 14 whenever it counts down to zero. It takes two countdowns to complete a cursor blink (one while the character is in its normal state and one while it is reversed), so 40 passes of the screen editor IRQ routine are required for each cursor blink. As a result, the cursor-

blink rate for the VIC screen is about every 2/3 second. The initialization value is read from ROM, so the 40-column blink rate is not programmable.

2601 $0A29 GDBLN

Character under cursor

This location is used to hold the original (unblinked) screen code for the character at the current 40-column screen cursor position. If the cursor is moved from the current position without printing a new character, this value will be restored to the position when the cursor is moved.

2602 $0A2A GDCOL

Color under cursor

This location is used to hold the original color of the character at the current 40-column screen cursor position. If the cursor is moved from the current position without printing a new character, this value will be restored to corresponding color memory location for the position when the cursor is moved.

2603 $0A2B CURMOD

VDC cursor mode

This location is a shadow for VDC internal register 10/SOA. See the entry for that register in Chapter 8 for details. The value here is copied into the register every time the screen editor routines are used to print a character to the 80-column screen.

2604 $0A2C VM1

VIC text screen and character base

This location is a shadow for the VIC chip screen and character base address register (53272/$D018) for the text (GRAPHIC 0) screen, or for the text portion of a split display. The value here is copied into the VIC register during the text screen-setup portion of the screen editor IRQ routine [$C194]. Refer to the discussion of the register in Chapter 8 for details. During the screen editor initialization [$C07B], this location is set to 20/$14. That value places screen memory at 1024/$0400 and character memory at 4096/$1000. If the SHIFT-Commodore combination is detected during the SCNKEY routine [$C55D], bit 1 of this location is toggled. This switches the character set base address between 4096/S1000 and 6144/$1800.

2605 $0A2D VM2

VIC bitmap and video matrix base

This location is a shadow for the VIC chip bitmap and video-matrix base address register (53272/$D018) for the bitmapped (GRAPHIC 1 or GRAPHIC 3) screen, or for the bitmapped portion of a split display. The value here is copied into the VIC register during the bitmapped screen-setup portion of the screen editor IRQ routine [$C194]. Refer to the discussion of the register in Chapter 8 for details. During the screen editor initialization routine [$C07B], this location is set to 120/$78. This value places the default video-matrix area at 7168/$1COO and the bitmap at 8192/$2000.

2606 $0A2E VM3

Starting page for VDC screen memory

The value in this location is used during the screen editor routines to determine the starting page within VDC RAM for 80-column screen memory. During the screen editor initialization routine [$C07B], this location is set to 0/$00, which places screen memory at address 0/$0000 in VDC RAM.

The value here determines where the screen editor thinks VDC screen memory begins, but not the actual starring address of the screen. (This location is not a shadow for a VDC register.) The actual screen starting address is determined by the value in VDC internal registers 12-13/$0C-$0D. If you change the register value, you should also change the value in this location, and vice versa.

2607 $0A2F VM4

Starting page for VDC attribute memory

The value in this location is used during the screen editor routines to determine the starting page within VDC RAM for 80-column attribute memory. During the screen editor initialization routine [$C07B], this location is set to 8/$08, which places attribute memory at address 2048/S0800 in VDC RAM.

The value here determines where the screen editor thinks VDC attribute memory begins, but not the actual starting address of attributes. (This location is not a shadow for a VDC register.) The actual attribute starting address is determined by the value in VDC internal registers 20-21/$14-$15. If you change the register value, you should also change the value in this location, and vice versa.

2608 $0A30 LINTMP

Ending row for screen input

This location is used by the routines which accept lines of input from the screen or keyboard to hold the number of the screen row on which the displayed line of characters ends. This value is tested to determine when the end of the input has been reached. For input from the screen, the BASIN routine [$C29B] fails to set this location, so the row number for the end of the input line must be set explicitly by storing the proper value (0-24) in this location.

2609-2610 $0A31-$0A32 SAV80

Temporary storage for 80-column memory manipulation

These locations are used to store the current row and column number values during the routines that clear or scroll lines on the 80-column screen. Location 2609/$0A31 holds the row number and 2610/$0A32 holds the column number.

2611 $0A33 CURCOL

Attribute of current cursor position

This location is used to hold the original attribute of the character at the current 80-column screen cursor position. If the cursor is moved from the current position without printing a new character, this value will be restored to corresponding attribute memory location for the position when the cursor is moved.

2612 $0A34 SPLIT

Scan line for screen split

This location holds the scan line for the raster interrupt which will set up the lower (text) portion of a split bitmapped/text screen. When a split screen is selected (when bit 6 of location 216/SD8 is set to %1), the value here will be copied into the VIC raster compare register (53266/$D012) during the bitmapped screen-setup portion of the screen IRQ routine [$C194]. This will cause a raster interrupt at the specified scan line, which will execute the text screen-setup portion of the interrupt routine to establish the text portion of the split screen. To find the scan-line value corresponding to a character row number, use the following formula:

scan line = (row number * 8) + 48

The default value for this location, set during the screen editor initialization routine [SC07B], is 208/$D0, which places the default split at screen row 20. The value here can be changed in BASIC by specifying a split parameter with the GRAPHIC statement.

2613 $OA35 FNADRX

Temporary storage for X register
This location is used to preserve the contents of the X register during the Kernal routine that reads a character from a file-name [$F7AE].

2614 $0A36 PALCNT

Jiffy clock compensation flag
In systems using the PAL video format, this location is used as a counter during the Kernal UDTIM routine [$F5F8], It is incremented each time UDTIM is called to update the jiffy clock locations (160-162/$A0-$A2). When the count has reached 5, the UDTIM routine is repeated and the counter is reset to zero. This triggers an extra update of the jiffy clock every fifth IRQ in PAL systems, so that ten extra jiffy clock "ticks" occur for each 50 IRQs. This means that the jiffy clock still increments 60 times per second on PAL-video 128s where the IRQ rate is only 50 per second. For systems using the NTSC format, this portion of the UDTIM routine is skipped, so the location will always hold its initial value of zero.

2615 $0A37 SPEED

Temporary storage for clock rate register
This location is used to hold the value in the VIC system clock rate register (53296/SD030) during tape and serial bus operations. The current register value is stored here at the beginning of the operation and the register is reset for slow (1-MHz) mode for the duration of the operation; then the value here is restored to the register once the operation is completed.

2616 $0 A38 SPRITES

Temporary storage for sprite enable register
This location is used to hold the value in the VIC sprite enable register (53269/$D015) during tape and serial bus operations. The current register value is stored here at the beginning of

the operation and the register is reset to 0/$00 to disable all sprites for the duration of the operation. The VIC chip requires extra timing cycles while sprites are active, so they are disabled to avoid disrupting the precise timing required for tape and serial operations. Once the operations are complete, the value here is restored to the VIC register.

2617 $0A39 BLANKING

Temporary storage for VIC control register
This location is used to hold the value in the VIC control register at 53265/$D011 during tape operations. The register value is stored here at the beginning of the tape operation, before bit 4 of the register is set to %0 to blank the screen during the operation. Upon completion of the tape operation, the value here is restored to the register.

2618 $0A3A HOLD_OFF

Custom mode flag
Normally, the system clock is set for the slow (1-MHz) rate and sprites are disabled during tape and disk operations to insure proper timing. However, this location can be used to allow the VIC clock and sprite registers to retain their current settings during such operations. When bit 7 of this location is set to %1, the registers are left unchanged. This location is initialized to 0/$00 by the IOINIT routine [$E109], part of the reset sequence. That setting is not changed by any ROM routine, so this feature is not used by the system.

2619 $0 A3B LDTB 1_SA

Starting page for 40-column screen memory
The value in this location is used during the screen editor routines to determine the starting page for 40-column (VIC) screen memory. During the screen editor initialization routine [$C07B], this location is set to the value in location 49228/$C04C, which is currently 4/$04. This specifies screen memory at address 1024/$0400.

The value here determines where the screen editor thinks VIC screen memory begins, but not the actual starting address of the screen. (This location is not a shadow for a VIC register.) The actual screen starting address is determined by the

value in bits 4-7 of the VIC register at 53272/SD018. If you change the register value, you should also change the value in this location, and vice versa.

2620-2621 $0A3C-$0A3D CLR_EA

Working pointer into 80-column memory

These locations are used to hold an address in VDC memory during the routine that clears a line of text on the 80-column screen [$C4C0] and the one that copies a line up or down for scrolling [$C53C],

2622-2623 $0A3E-$0A3F Unused

These locations are unused by any system ROM routine, and are thus available for your own programming.

2624-2650 $0A40-$0A5A

Screen editor variable storage for the inactive screen

The screen editor variables for whichever screen (40- or 80-column) is currently inactive are stored here. When the screens are switched, the screen editor SWAPPER routine [$CD2E] exchanges the contents of this area with the values for the active screen at 224-250/$E0-$FA. Thus, the active and inactive screens are totally independent, and all window size settings for the inactive screen will be preserved until the screen becomes active again.

Location 2650/$0A5A should not be included in this range. Only locations 224-249/$E0-$F9 actually hold screen editor variables. However, the SWAPPER routine incorrectly copies 27 values instead of the proper 26, so the contents of location 2650/$0A5A and location 250/$FA will be exchanged whenever the active screen is switched. Both locations are normally unused.

2651-2655 $0A5B-$0A5F Unused

None of the locations in this range is used by any system routine, so all are available for your own programming,

2656-2665 $0A60-$0A69

Storage for inactive tab-stop bitmap

The tab-stop bitmap for whichever screen (40- or 80-column) is currently inactive is stored here. When the screens are

switched, the screen editor SWAPPER routine [$CD2E] exchanges the contents of this area with the tab-stop bitmap for the active screen at 852-861/$0354-$035D.

2666-2669 $0A6A-$0A6D

Storage for inactive line-link bitmap

The line-link bitmap for whichever screen (40- or 80-column) is currently inactive is stored here. When the screens are switched, the screen editor SWAPPER routine [$CD2E] exchanges the contents of this area with the line-link bitmap for the active screen at 862-865/$035E-$0361.

2670-2687 $0A6E-$0A7F Unused

None of the locations in this 18-byte area is used by any system ROM routine, so they are available for your own programming.

Monitor Working Storage Area
2688-2751/$0A80-$0ABF

2688-2703 $0A80-$0A8F FNBUFF

Filename buffer for load, save, or verify

The load/save/verify setup routine stores the filename associated with a monitor L, S, or V command here (up to 16 characters).

2688-2719 $0A80-$0A9F HBUFF

Search pattern buffer

The monitor H (hunt for byte pattern) routine [$B2CE] fills this buffer with the byte pattern being searched for (up to 32 characters). Characters in the specified memory range are then compared against the buffer contents to search for a matching pattern in memory. If the address range for the search includes this buffer, an artificial match will be found—the buffer contents will always match themselves.

2720-2727 $0AA0-$0AA7 XFORM

Working storage for base conversion

The hex-to-decimal conversion routine [$BA07] uses these locations for working storage during the conversion, leaving the results in 2720-2723/$0AA0-$0AA3 in BCD (binary coded

decimal) format. The base conversion routine [$BA47] puts the value to be converted into 2720-2723/$0AA0-$0AA3 for manipulation. The routine to print octal, binary, or decimal values uses 2720-2723/$0AA0-$0AA3 to hold the value to be displayed.

2720-2729 $0AA0-$0AA9 ASMBUF

Instruction assembly buffer

The assemble routine [$B406] packs the three-character mnemonic in the instruction being assembled into two bytes and stores them in 2720-2721/$0AA0-$0AAl. If characters follow the mnemonic, they are copied into 2722-2729/$0AA2-$0AA9. If a numeric parameter is found, a dummy value consisting of a $ character followed by either two or four zeros (depending on whether the value is greater than 255) is substituted. This character pattern is then evaluated to determine the addressing mode for the instruction.

2730 $0AAA FORMAT

Instruction format flag

The routine to calculate the mnemonic and addressing mode for an opcode [$B659] uses this location to hold a flag value indicating the addressing mode in use, which determines the format in which the instruction must be displayed or entered.

2731 $0AAB LENGTH

Instruction length

The routine to calculate the mnemonic and addressing mode for an opcode [$B659] uses this location to hold the number of bytes which should follow the opcode in the instruction (0-2).

2732-2734 $0AAC-$0AAE MSAL

Three-character mnemonic pattern

The assemble routine [$B406] stores the first three-character group from the input line in these locations for evaluation as an ML mnemonic.

2735 $0AAF SXREG

Temporary storage tor X register

The subroutine to determine the proper opcode for a mnemonic [$B57C], the routine to print hexadecimal byte values,

and the routine to test the next buffer character [$B8E7] all stash the X register contents here upon entry, and restore the value to the X register upon exit. The routine to decrement the pointer address or line count stored in 96-98/$60-$62 uses this location to hold the amount by which the stored value is to be decremented.

2736 $0AB0 Unused

This location is not used by any 128 ROM routine, and is thus available for your own programming.

2737 $0AB1 OPCODE

Calculated opcode

The assemble routine [$B406] uses this location to hold the opcode calculated for the instruction being assembled.

2738 S0AB2 XSAVE

Temporary storage for X register

The monitor indirect fetch [$B11A], indirect store [$B12A], and indirect compare [$B13D] routines stash the value in the X register here upon entry, then restore the value to the X register upon exit.

2739 $0AB3 DIRECTION

Transfer direction flag

The monitor compare/transfer routine [$B231] uses this location in execution of the T (transfer) command to indicate the direction in which bytes are to be transferred. For downward moves (source address greater than destination address), the flag will be set to zero; for upward moves (destination address greater than source address), the flag will be set to 128/S80.

2740 $0AB4 COUNT

Digit counter

The routine to convert input parameters into numeric values [$B7CE] uses this location to hold a count of the hexadecimal digits in the converted value. The routine to print values in octal, binary, or decimal [$BA47] uses this location as a counter of digits printed in the value being displayed.

2741 $0AB5 NUMBER

Temporary storage for parameter conversion
The routine to convert input parameters into numeric values [$B7CE] uses this location to hold the numeric value of the input digit currently being evaluated.

2742 $0AB6 SHIFT

Number of bits per digit for base
The routine to convert input parameters into numeric values [$B7CE] and the routine to print values in octal, binary, or decimal [$BA47] use this location to hold the number of bits to be interpreted per digit for the value being converted or displayed.

2743-2745 $0AB7-$0AB9 TEMPS

Monitor temporary storage
The routine to convert input parameters into numeric values [$B7CE] uses these locations as working storage when evaluating decimal digits. Monitor routines which accept two or more address parameters store the second (ending) address here. For upward transfers, the compare/transfer routine [$B231] moves the value here to the working pointer at 102-104/$66-$68.

2746-2751 $OABA-$OABF Unused

These locations are not used by any 128 ROM routines, and are thus available for your own programming.

Kernal Working Storage

2752-2815/$OAC0-$OAFF

2752 $OACO CURBNK

Counter for function ROM testing
Both the Kernal routine which checks for the presence of ROM in the internal and external (cartridge) ROM address slots [$E26B] and the Kernal PHOENIX routine [$F867] which initializes function ROMs use this location as a countdown for the number of slots remaining to be tested. The location is set to 3, then decremented each time a slot is checked or initialized. The routines end when the value here rolls over from 0/$00 to 255/$FF after the fourth slot is tested or initialized.

Thus, this location will normally contain 255/$FF upon completion of either routine.

2753-2756 $0ACI-$0AC4 PAT

Table of identifiers for function ROMs

The Kernal routine which tests for the presence of function ROMs [$E26B] initializes these locations to 0/$00 before checking any of the four possible ROM address slots. A ROM is considered present in the slot if the character codes for the letters CBM are found at an offset of seven bytes from the starting address of the slot. If this test pattern is found, the function ROM ID byte is copied from an offset of six bytes beyond the starting address of the slot into the corresponding location in this table:

Location	ROM slot address
2753/$0ACl	32768/$8000 internal
2754/$0AC2	49152/$C0O0 internal
2755/$0AC3	32768/S8000 external
2756/$0AC4	49152/$C0O0 external

If the identifier byte is l/$01, the cartridge is autostarting and the test routine immediately calls the cold-start vector for that ROM. Otherwise, the Kernal PHOENIX routine [$F867], part of the BASIC cold-start sequence, will call the cold-start vector for any ROM slots with nonzero entries in this table.

2757 $0AC5 DK_FLAG

This location is mentioned in Commodore literature as "reserved for foreign screen editors/' but its exact use is unclear. It is unused by any routine in the U.S. version of the system ROMs.

2758-2815 $0AC6-$0AFF Unused

This 58-byte area is described in Commodore literature as "reserved for system use." However, no routines in the current version of the system ROMs make use of any of these locations. Still, unless you are desperate for free locations outside the BASIC area, it's probably best to avoid using these locations to insure compatibility with future versions of the ROM.

Cassette Buffer and Disk Boot Buffer

2816-3071/$0B00-$0BFF

The first 192 bytes of this area (2818-3007/$0B00-$0BBF) are used for the cassette buffer. When you're reading from tape, file headers are loaded into this area until one is found with the specified filename. When you're writing to tape, this area is used to assemble the tape header for the file. When the buffer contains a tape header, the locations are used as follows:

Location	Function
2816/$BO0	Header type identifier
2817-2818/$0B01-$0BO2	Starting address for file
2819-2820/$OB03-$OB04	Ending address for file
2821-3007/$OB05-$OBBF	Filename

A header type identifier of 1 indicates a relocatable program file; 3 indicates a nonrelocatable program file; 4, a data file; and 5, an end-of-tape marker. A type identifier value of 2 means that the block contains data rather than a header. Although the filename can be up to 187 bytes long, such names are unusual. When a header is read into the buffer, only the first 16 characters of the filename will be displayed following the FOUND message. When a header is being assembled, all unused filename bytes will be set to 32/$20, the value for the space character.

The cassette buffer is used to hold blocks of data when data files are read from or written to tape. When a data file is being written, after the header is written to tape, the first byte here is set to 2, the identifier for a data block; then the remaining 191 bytes are filled with the data to be written to the file. Only when the buffer is completely full is the block of data actually added to the file. This is why it is important to properly close any file opened for writing. If the file is not closed, the last block of data will not be written to tape. When a file is opened for reading, after the proper header is identified, the first block of data is read into the buffer. Subsequent bytes will be read from the buffer—not directly from tape—until all 191 data bytes have been read from the buffer; then the next block will be read into the buffer. See Chapter 9 for more information on tape data storage.

In the 128, this area has a second function: the boot sector buffer. If the BOOT_CALL routine [$F890] finds a disk in the specified drive when it is called, the contents of sector 0 of

track 1 of the disk are read into this area. The first three bytes of the buffer are then examined. If those locations (the first three bytes from the sector) contain the character codes for the letters CBM, then the routine assumes that a boot disk has been found and proceeds with the boot process. Refer to the entry in Chapter 9 for details. It would be possible to simulate a boot by filling the buffer with the proper values, then jumping into the BOOT_CALL routine at address 63737/$F8F9.

The B0OT_CALL routine is normally executed during each reset as part of the BASIC cold-start sequence. It can also be called from the Kernal jump table entry at 65363/$FF53, and can be initiated from BASIC via the BOOT statement.

Actually, the cassette buffer is not located absolutely at this area. The base address of the cassette buffer is determined by the value in locations 178-179/$B2-$B3. Those locations are initialized to 2816/$0BO0 during the RAMTAS routine [$E093], part of the reset sequence. No system ROM routines change that setting, but another free 192-byte area could be used, with two restrictions: the buffer must start at an address greater than 511/S1FF, and the buffer must be visible in the bank 15 memory configuration. The disk-booting routines, on the other hand, do use the absolute address of this area. This area will always be used as the boot sector buffer, regardless of the value in the cassette buffer pointer.

If tape data storage is not being used, this 256-byte area is available for other uses, such as to hold short machine language routines. The cassette buffer has been a popular area for ML since the days of the first Commodore PET/CBM computers. However, the contents of this area will be overwritten whenever the system attempts to boot a disk, including the time during any reset when the drive is turned on and contains a disk. You should choose another area if you want your machine language to survive intact following a reset.

RS-232 Input Buffer

3072-3327/$0C00-$0CFF

The routines that receive characters via the RS-232 interface are executed during NMI interrupts. Any characters received are held in this area until they can be read, usually by using the Kernal GETIN routine. This is a circular buffer with no fixed beginning or end. Location 2585/S0A19 holds the offset

from the starting address of the buffer to the next character waiting to be read (called the head of the buffer). Location 2584/$0A18 holds the offset to the next free position in the buffer (called the tail of the buffer). The buffer is considered empty when the two offset addresses are equal, and full when the buffer tail offset is one less than the head offset. Bits 2 and 3 of the RS-232 status byte at 2580/$0A14 indicate, respectively, when the buffer is full or empty.

Actually, the buffer is not located absolutely at this area. The starting address of the RS-232 input buffer is determined by the value in locations 200-201/$C8-$C9. The pointer locations are initialized to 3072/S0C00 by the RAMTAS routine [$E093], part of the reset sequence. No other system routine changes the address in that pointer, so the buffer will be located here unless you explicitly move it. However, the buffer can be moved to another free area of memory simply by changing the address in the pointer. (The area selected must be visible in the bank 15 configuration.)

If RS-232 communications are not used, this buffer area, along with the one at 3328-3583/$0D00-$0DFF, is available for other purposes such as machine language routines or additional sprite definitions.

RS-232 Output Buffer

3328-3583/$0D00-$0DFF

The routines that transmit characters via the RS-232 interface are executed during NMI interrupts. Characters are stored in this area, usually by the BSOUT routine, while awaiting transmission. This is a circular buffer with no fixed beginning or end. Location 2586/S0A1A holds the offset from the starting address of the buffer to the next character waiting to be sent (called the head of the buffer). Location 2587/S0A1B holds the offset to the next free position in the buffer (called the tail of the buffer). The buffer is considered empty when the two offset addresses are equal, and full when the buffer tail offset is one less than the head offset.

Actually, the buffer is not located absolutely at this area. The starting address of the RS-232 output buffer is determined by the value in locations 202-203/$CA-$CB. The pointer locations are initialized to 3328/$0D00 by the RAMTAS routine [$E093], part of the reset sequence. No other system routine

changes the address in the pointer, so the buffer will be located here unless you explicitly move it. However, the buffer can be moved to another free area of memory simply by changing the address in the pointer. (The area selected must be visible in the bank 15 configuration.)

If RS-232 communications are not used, this buffer area, along with the one at 3072-3327/$0CO0-$0CFF, is available for other purposes such as machine language routines or additional sprite definitions.

Sprite Pattern Storage Area

3584-4095/$0E00-$0FFF

This area is reserved by the system to hold sprite pattern definitions. Each sprite pattern requires 64 bytes and must start at an address which is an exact multiple of 64/$40. Other free locations within the current video bank which meet these criteria can also be used for sprite pattern storage, but BASIC sprite routines such as SPRSAV and SPRDEF assume that sprite patterns reside in this area. The sprite pointers for the default screen memory position (2040-2047/$07T8-$07FF) are initialized during the BASIC cold-start sequence as follows:

Pattern area	Pointer value	Default sprite
3584-3647/$0E00-$0E3F	56/$38	0
3648-371l/$0E40-$0E7F	57/$39	1
3712-3775/$OE80-$0EBF	58/S3A	2
3776~3839/$0ECO-$0EFF	59/$3B	3
3840-3903/$OFOO-$OF3F	60/$3C	4
3904-3967/$0F40-$0F7F	61/$3D	5
3968-4O31/$0F80-$0FBF	62/$3E	6
4032-4095/$0FCO-$0FFF	63/$3F	7

No system ROM routines change these settings. If your program doesn't require sprites, this area can be used for other purposes such as to hold machine language routines.

Programmable Key Definition String Area

4096-4351/$1000-$10FF

This 256-byte area is used to hold the strings for the ten programmable keys supported by the 128's screen editor routines: F1-F8, SHIFT-RUN/STOP, and HELP. Each of the first 10

bytes of the area (4096-4105/$1000-$1009) holds the length of one of the definition strings:

Location	Key
4096/$1000	Fl
4097/$1001	F2
4098/$1002	F3
4099/$1003	F4
4100/$1004	F5
4101/$1005	F6
4102/$1006	F7
4103/$1007	F8
4104/$1008	SHIFT-RUN/STOP
4105/$1009	HELP

The remaining 246 bytes (4106-435l/$100A-$lOFF) are available for definition strings. There is no particular limit on the length of an individual definition string, except that the combined length of all the definition strings cannot exceed 246 bytes. The definition strings correspond to keys in the order shown above. The offset to the first character in a particular string is found by adding the lengths of all preceding definitions. No special characters are used to separate the strings. It is possible for a key to have no associated definition string, in which case the length location for the key should be set to 0/$00. The default definitions for the keys are as follows:

Key	Default definition
Fl	GRAPHIC
F2	DLOAD"
F3	DIRECTORY {RETURN}
F4	SCNCLR {RETURN}
F5	DSAVE"
F6	RUN {RETURN}
F7	LIST {RETURN}
F8	MONITOR{RETURN}
SHIFT-RUN/STOP	DL"*{RETURN}RUN{RETURN}
HELP	HELP{RETURN}

These definitions, along with the corresponding length values, are copied from locations 52904-52980/$CEA8-$CEF4 in screen editor ROM during the Kernal CINT routine [$C07B]. In BASIC, the KEY statement can be used to change definitions. From machine language, the Kernal PFKEY routine [$FF65] (or screen editor KEYSET routine [$C021]) can be used. Programmable keys are handled by a subroutine [$C6CA] within the screen editor keyscan routine.

BASIC Working Storage
4352-4607/$1100-$1IFF

4352-4400 $1100-$1130 DOSSTR

DOS command assembly area

The BASIC statements that issue DOS commands—for example, HEADER, COPY, CATALOG, and SCRATCH—use this area to assemble the command string to be sent to the disk drive.

4401-4402 $1131-$1132 XPOS

Bitmapped-screen pixel-cursor horizontal position

These locations hold the horizontal *(x)* coordinate of the current position of the pixel cursor on the bitmapped screen. The range of values here depends on the scale factor currently in use. If scaling is not used, the value can be found in the range 0-319. In any case, a value of zero specifies the left edge of the screen. The value here is set to 0/$00 whenever the bitmapped screen is cleared, either by the SCNCLR routine or by adding the clear parameter to a GRAPHIC statement. After execution of any BASIC graphic statement, this location will hold the value of the final horizontal pixel position affected by the operation. The value here can be set explicitly using the LOCATE statement, which stores the specified horizontal position in this location. If the DRAWTO form of the DRAW statement is used, the line will begin at the horizontal position specified here.

4403-4404 $1133-$1134 YPOS

Bitmapped-screen pixel-cursor vertical position

These locations hold the vertical (y) coordinate of the current position of the pixel cursor on the bitmapped screen. The range of values here depends on the scale factor currently in use. If scaling is not used, the value can be found in the range 0-199. In any case, a value of zero specifies the top edge of the screen. The value here is set to 0/$00 whenever the bitmapped screen is cleared, either by the SCNCLR routine or by adding the clear parameter to a GRAPHIC statement. After execution of any BASIC graphic statement, this location will hold the value of the final vertical pixel position affected by the operation. The value here can be set explicitly using the

LOCATE statement, which stores the specified vertical position in this location. If the DRAWTO form of the DRAW statement is used, the line will begin at the vertical position specified here.

4405-4406 $1135-$1136 XDEST

Final horizontal pixel position for graphics operations
These locations hold the calculated ending horizontal pixel-cursor position for BASIC graphics operations. The operation is complete when the value in locations 4401-4402/$1131-$1132 equals the value here.

4407-4408 $1137-$1138 YDEST

Final vertical pixel position for graphics operations
These locations hold the calculated ending vertical pixel-cursor position for BASIC graphics operations. The operation is complete when the value in locations 4403-4404/$1133-$1134 equals the value here.

4409-4455 $1139-$1167

Working storage for assorted graphics routines
BASIC graphics routines such as BOX, CIRCLE, DRAW, and PAINT use various locations in this range to perform the calculations necessary to plot the points for the figure being drawn. The MOVSPR routine also uses some of these locations for sprite position calculations in those cases where the sprite is moved relative to the pixel cursor.

4456 $ 1168 CHRPAG

Starting page for character pattern definitions
This location is used during the CHAR routine [$67D7] in the calculations to determine where character shapes are to be placed on the bitmapped screen. The value here is the starting page of character memory. This location will hold the value from either 4588/$llEC or 4587/$llEB.

4457 $1169 BITCNT

Bit counter for shape retrieval
This location is used during the GSHAPE routine [$658D] as a counter for the bits to be read from each byte of the storage string.

4458 $116A SCALEM

Scaling flag
This location indicates whether the scaling feature is to be used when graphics are drawn on the bitmapped screen. While this location contains 0/$00, scaling will not be used. When the location contains any nonzero value, the horizontal and vertical coordinates for all graphics routines will be scaled according to the values in locations 135-136/$87-$88 and 137-138/$89-$8A. This location is initialized to 0/SOO (scaling off) during the BASIC cold-start sequence, and also whenever the clear-screen parameter is included in a GRAPHIC statement. The routine to execute the SCALE statement [$6960] will store the first parameter following SCALE (0 or 1) here.

4459 $116B WIDTH

Line width for bitmapped graphics routines
The value here determines whether the lines drawn by BASIC bitmapped graphics routines are to be standard width (indicated when this location contains 0/$00) or double width (indicated when this location contains any nonzero value). The value here is initialized to 0/$00 (normal width) during the BASIC cold-start sequence. The routine to execute the WIDTH statement [$71B6] will store the width parameter minus 1 in this location.

4460 $116C FILFLG

BOX fill flag
This location is used during the BOX routine [$62B7] to specify whether the shape is to be open or filled. If the value here is 0/$00 the shape will be open; otherwise, it will be filled. This location is initialized to 0/$00 (open shapes) during the BASIC cold-start sequence. When a BOX statement is executed, the seventh parameter (paint) following the statement will be copied here. If that parameter is omitted, it will default to 0/$00.

4461 S116D BITMSK

Bit mask value
This location is used as a mask value to select individual bits during the DRAW and SPRDEF routines.

4462 $116E NUMCNT

Temporary storage for assorted routines

This location is used for temporary working storage during the CHAR, MOVSPR, and SPRDEF routines.

4463 $116F TRCFLG

Trace mode flag

This location is used to indicate whether BASIC is in trace mode. When a program is executed in trace mode, the line number of each program line is printed as it is executed. Trace mode is off when bit 7 of this location is %0, and on when bit 7 is %1. This location is initialized to 0/$00 (trace off) during the BASIC cold-start sequence. It is also reset to 0/$00 during the NEW routine. The value here can be changed using the TRON and TROFF statements. TRON sets this location to 255/$FF, and TROFF resets it to 0/$00.

4464-4467 $1170-$1173 RENUM_TMPS

Working storage for RENUMBER

These locations are used for working storage during BASIC'S RENUMBER routine [$5AF8], Locations 4464-4465/$1170-$1171 hold the line number at which renumbering is to begin, and 4466-4467/$1172-$1173 hold the increment by which subsequent lines are to be renumbered.

4468 $1174 T3

Loop counter for reading directory entries

This location is used during the CATALOG/DIRECTORY routine [$A07E] as a counter in the loop to discard extraneous characters before the block count.

4469-4470 $1175-$1176 T4

Block count for directory entry

These locations are used during the CATALOG/DIRECTORY routine [$A07E] to hold the block count for each file entry read from the drive.

4471 $1177 VTEMP3

Working storage for graphics parameter scaling

This location is used for working storage during the routine that scales graphics parameters [$9DAE] when the SCALE option is in effect.

4472 $1178 VTEMP4

Working storage for graphics parameter evaluation

This location is used during the routine that evaluates parameters for graphics routines [$9E6D] to hold the offset to the parameter being evaluated.

4473 $1179 VTEMP5

Working storage for graphics parameter evaluation

This location is used during the routine that evaluates parameters for graphics routines [$9E6D] to hold a value indicating the parameter type.

4474-4475 $117A-$117B ADRAY1

Pointer to floating point-to-integer conversion routine

These locations point to the routine that converts the floating-point value in FAC1 (99-103/$63-$67) into a two-byte integer value in the accumulator (low byte) and Y register (high byte). The BASIC cold-start sequence initializes the value here to 33951/$849F, the address of that routine in the current version of BASIC ROM.

This pointer is especially useful in conjunction with the USR function. See the entry in Chapter 5 for details. To perform this conversion, it's best to use the indirect JMP ($117A) instead of the absolute JMP $849F. That way, your program will still work if the ROM is revised in future versions.

4476-4477 $117C-$117D ADRAY2

Pointer to integer-to-floating point conversion routine

These locations point to the routine that converts a two-byte integer value in the accumulator (low byte) and Y register (high byte) to a floating-point value in FAC1 (99-103/$63-$67). The BASIC cold-start sequence initializes the value here to 31036/$793C, the address of that routine in the current version of BASIC ROM.

This pointer is especially useful in conjunction with the USR function. See the entry in Chapter 5 for details. To perform this conversion, it's best to use the indirect JMP ($117F) rather than the absolute JMP $793C. That way, your program will still work if the ROM is revised in future versions.

4478-4565 $117E-$11DS SPRITE-DATA

Sprite movement control data

The MOVSPR statement has an option to set sprites in motion at a given angle and speed. These locations hold data concerning sprite motion. For moving sprites, the values here will be used to generate position values that will be copied to the shadow registers at 4566-4582/$llD6-$llE6. The locations are used as follows:

		Sprite 0	Sprite 1	Sprite 2	Sprite 3
Speed		4478/$117E	4489/J1189	4500/$1194	4511/$119F
Speed countdown		4479/$117F	4490/I118A	4501/$1195	4512/$11AO
Direction		4480/S1180	4491/$118B	4502/$1196	4513/$11A1
Horizontal increment	(low)	4481/$1181	4492/$118C	4503/$1197	4514/$11AZ
	(high)	4482/$1182	4493/$118D	4504/$1198	4515/$11A3
Vertical increment	(low)	4483/$1183	4494/$118E	4505/$1199	4516/S11A4
	(high)	4484/S1184	4495/$118F	4506/$119A	4517/S11A5
Horizontal position	(low)	4485/$1185	4496/$1190	4507/S119B	4518/$11A6
	(high)	4486/$1186	4497/S1191	4508/$119C	4519/$11A7
Vertical position	(low)	4487/$1187	4498/$1192	4509/$119D	4520/$11 A3
	(high)	4488/$1188	4499/S1193	4510/$119E	4521/$11A9

		Sprite 4	Sprite 5	Sprite 6	Sprite 7
Speed		4522/$! 1AA	4533/$llB5	4544/$llC0	4555/$llCB
Speed countdown		4523/$ 11AB	4534/S11B6	4545/$llO	4556/$llCC
Direction		4524/$ 11 AC	4535/$]1B7	4546/$11C2	4557/S11CD
	(low)	4525/$ 11 AD	4536/$11B8	4547/S11C3	4558/$llCE
Horizontal increment	(high)	4526/$llAE	4537/$11B9	4548/$llC4	4559/$llCF
	(low)	4527/$llAF	4538/$ 11BA	4549/$llC5	4560/$11 DO
Vertical increment	(high)	4528/S11B0	4539/S11BB	455O/$11C6	4561/$I1D1
	(low)	4529/$llBl	4540/$11BC	4551/$11C7	4562/$llD2
Horizontal position	(high)	4530/$11B2	4541/$11BD	4552/$HC8	4563/$llD3
	(low)	4531/S11B3	4542/$llBE	4553/$llC9	4564/$llD4
Vertical position	(high)	4532/S11B4	4543/$llBF	4554/$llCA	4565/$ 11D5

In addition to using the MOVSPR routine, the values here can be set directly to set a sprite in motion. (The corresponding sprite must be enabled before the motion values have any effect.) The speed value (0-15) determines how many times per IRQ interrupt the horizontal and vertical increment values will be applied to the horizontal and vertical position values. If the speed value is 0/$00, the corresponding sprite will not be moved. The speed value is copied to the countdown value during each interrupt. The direction value can have one of the following values:

Direction value	Sprite motion
0	*x* increasing, *y* decreasing
1	*x* increasing, *y* increasing
2	*x* decreasing, *y* increasing
3	*x* decreasing, *y* decreasing

All locations in this range are initialized to 0/$00 during the BASIC cold-start sequence. The warm-start sequence performs the less dramatic initialization step of resetting all the speed locations to 0/$00, which stops all sprite motion.

4566-4582 $11D6-$11E6 VIC-SAVE

Shadows for VIC sprite position registers

The contents of these locations are copied into the VIC chip sprite position registers (53248-53264/$D000-$D010) during each pass through the BASIC IRQ routine [$A84D]. As long as the BASIC IRQ routine is active, the VIC registers cannot be changed directly. Instead, you should store the desired register value in the corresponding shadow location. All locations in this range are initialized to 0/$00 during the BASIC cold-start sequence. The values here can be set using the MOVSPR statement. The MOVSPR routine [$6CC6] sets the value here directly when a static sprite position is specified. When a moving sprite is specified, the movement information is stored in the table at 4478-4565/$117E-$llD5 and the values here are updated during each pass through the BASIC IRQ routine [$A84D]. The locations are used as follows:

Location	Register	Function
4566/$llD6	53248/$D0O0	Sprite 0 horizontal position
4567/$llD7	53249/$D001	Sprite 0 vertical position
4568/$llD8	53250/$D002	Sprite 1 horizontal position
4569/$llD9	53251/$D003	Sprite 1 vertical position
4570/$1 IDA	53252/$D004	Sprite 2 horizontal position
4571/S11DB	53253/$D005	Sprite 2 vertical position
4572/$llDC	53254/$D006	Sprite 3 horizontal position
4573/$llDD	53255/$D007	Sprite 3 vertical position
4574/$llDE	53256/$D008	Sprite 4 horizontal position
4575/$llDF	53257/$D009	Sprite 4 vertical position
4576/$llE0	53258/$D00A	Sprite 5 horizontal position
4577/$llEl	53259/$D00B	Sprite 5 vertical position
4578/$llE2	53260/$D00C	Sprite 6 horizontal position
4579/$llE3	53261/$D0OD	Sprite 6 vertical position
458O/$11E4	53262/$D00E	Sprite 7 horizontal position
4581/$11E5	53263/$D00F	Sprite 7 vertical position
4582/S11E6	53264/$D010	Most significant bits of horizontal position

4583-4584 $11E7-$11E8

Shadows for VIC sprite-collision registers
During any pass through the BASIC IRQ routine [$A84D] where either or both of the sprite-collision latch flags {bits 1-2 of 53273/$D019) are found to be set to %1, the contents of the corresponding VIC chip sprite-collision register will be recorded in these locations. For sprite-foreground collisions (indicated when bit 1 of the flag is set), any bits in the register at 53279/$D01F which are set to %1 will also be set to %1 in location 4584/$llE8. Likewise, for sprite-sprite collisions (indicated when bit 2 of the flag is set), the %1 bits in the register at 53278/$D01E will be recorded in location 4583/$llE7. Thus, these locations will accumulate collision readings until they are cleared, rather than simply holding the most recent collision values. The routine for the BASIC function BUMP [$837C] returns values based on the contents of these locations, rather than on the actual register contents. BUMP(l) returns the sprite-sprite collision value in 4583/$llE7, and BUMP(2) returns the sprite-foreground collision value in 4584/$llE8. Either location will be reset to 0/$00 after being read by BUMP. Both locations are also initialized to 0/$00 during the BASIC cold-start sequence.

4585-4586 $1lE9-$11EA

Shadow for VIC light pen registers
During any pass through the BASIC IRQ routine [$A84D] where the light pen latch flag (bit 3 of 53273/$D019) is found to be set to %1, the contents of the VIC chip light pen registers at 53267-53268/$D013-$D014 will be copied into these locations. The routine for the BASIC function PEN [$82AE] returns values based on the contents of these locations, rather than on the actual register contents. PEN(O) returns the value in 4585/S11E9, multiplied by 2. PEN(l) returns the value in 4586/S11EA. Either location will be reset to 0/$00 after being read by PEN. Both of these locations are also initialized to 0/$00 during the BASIC cold-start sequence.

4587 $11EB UPPER-LOWER

Starting page for alternate character set during CHAR
The value here determines the starting page in system memory for the alternate set of characters used during the CHAR

statement for bitmapped screens (see location 4588/$llEC for more information). CHAR will always begin using the character set pointed to in location 4588/$llEC. To switch to the alternate character set, the CHAR string must include character code 14/$0E. Character code 142/$8E switches back to the default set. This location is initialized to 216/$D8 during BASIC cold start. That value selects the ROM lowercase/ uppercase set at 55296/$D800 as the alternate character pattern source. If you use a custom character set, you can change the value here to have CHAR use your new characters. However, the new character set must be visible in the bank 14 memory configuration, since that is how the system will be configured when character pattern data is read.

4588 $11EC UPPER-GRAPHIC

Starting page for default character set during CHAR

The value here determines the starting page in system memory for the default set of characters used during the CHAR statement. The value here doesn't affect any statement other than CHAR, and is used only when character shapes are being placed on a bitmapped screen. (When CHAR is used to place characters on the text screen, the screen editor printing routines are used instead.) For bitmapped screens, CHAR will always begin using the character set pointed to here, regardless of the character set in use on the text screen. To switch to the alternate character set (starting page in 4587/$llEB), the CHAR string must include character code 14/$0E. Character code 142/$8E switches back to the character set pointed to here. Thus, it is possible to mix the two character sets in a single CHAR statement. This location is initialized to 208/$D0 during BASIC cold start. That value makes the ROM uppercase/ graphics set at 53248/$D000 the default character pattern source. If you use a custom character set, you can change the value here to have CHAR use your new characters. However, the new character set must be visible in the bank 14 memory configuration, since that is how the system will be configured when character pattern data is read.

4589 SUED DOSSA

Channel number for BASIC relative file operations
The channel number (secondary address) for BASIC relative file operations is stored in this location.

4590-4607 $11EE-$11FF Unused

None of these locations is used by any system ROM routine.

BASIC General-Purpose Working Storage

4608-4863/$1200-$12FF

4608-4609 $1200-$1201 OLDLIN

Line number where program stopped
Whenever a BASIC program stops because of an END or STOP statement, or because the end of the program has been reached, or because the RUN/STOP key has been pressed, then the STOP/END routine [S4BCB] will be executed. That routine stores the line number where the program stopped in this pair of locations (in low-byte/high-byte format). The CONT routine [$5A60] uses the value here to determine where to restart the program. These locations are also used for temporary storage during the RENUMBER routine [$5AF8].

4610-4611 $1202-61203 OLDTXT

Pointer to the start of current line

Each time a BASIC program line is executed, the address of the first character of program text in the line is stored in these locations. The high byte (4611/$1203) is also used as a flag to indicate whether the program can continue after being halted. The CONT routine [$5A60] will give a CAN'T CONTINUE error message if the flag byte is 0/$00. The flag location is initialized to 0/$00 during the CLR routine [$51F8]—you can't CONTinue a program before it is run. If the program halts without errors, the flag location will hold the high byte of the address of the line where the program stopped, which will always be nonzero, so the program can be CONTinued. However, if the program stopped because of an error, or if any lines are changed after the program has stopped, then the flag location will be reset to zero and the program cannot be CONTinued.

4612-4615 $1204-$1207 PUCHRS

Character definitions for PRINT USING

The values in these locations determine which characters will be used for the redefinable characters in the PRINT USING format. The default values are copied from locations $5250-$5253 in BASIC ROM during the CLR routine, so the default definitions will be restored each time a program is run. In BASIC, the definitions here can be changed using the PUDEF statement.

4612/$1204: This location holds the filler character for the pattern, the one which will be used to fill unused positions in the format. The default value is 32/$20, the space character.

4613/$1205: This location holds the comma character for the pattern. The character with the code specified here will be substituted wherever a comma appears in the PRINT USING format. The default value is 44/$2C, the comma character.

4614/$1206: This location holds the decimal point character for the pattern. The character with the code specified here will be substituted wherever a decimal point (period) appears in the PRINT USING format. The default value is 46/$2E, the period (.) character.

4615/$1207: This location holds the monetary symbol character for the pattern. The character with the code specified here will be substituted wherever a dollar sign ($) appears in the PRINT USING format. The default value is 36/$24, the dollar sign character.

4616 $1208 ERRNUM

Number of most recent error

Whenever a BASIC error occurs, the ERROR routine [$4D3C] stores the error number here. The reserved variable ER always reflects the value in this location. Refer to the entry for the error message table in Chapter 5 [$484B] for a complete list of error numbers. Once an error number is stored here, the value is retained until another error occurs or until the location is reinitialized. This location is initialized to 255/$FF during CLR [$51F8] (also executed as part of NEW and RUN). This setting results in a value of —1 in the reserved variable ER, so when ER contains that value no error has yet occurred.

4617-4618 $1209-$120A ERRLIN

Line number where most recent error occurred

Whenever a BASIC error occurs, the ERROR routine [$4D3C] checks the run mode flag (127/$7F) to see if the error occurred in a program line or an immediate mode line. If the error was in a program line, the current line number is copied here from locations 59-60/$3B-$3C. The reserved variable EL always reflects the value in these locations. Once a line number is stored here, it will be retained until another error occurs or until the locations are reinitialized. These locations are initialized to 65535/$FFFF during CLR [$51F8] (also executed as part of NEW and RUN). Thus, when the reserved variable EL contains 65535 no error has yet occurred.

4619-4620 $120B-$120C TRAPNO

Target line number for TRAP statement

When error trapping is enabled with the TRAP statement, the target line number to which the program will be directed when an error occurs is stored here (in standard low-byte/high-byte order). Location 4620/$120C is also used as a flag to determine whether trapping is enabled. The flag location is initialized to 255/$FF during CLR (which is also part of NEW and RUN), Since the high bytes of all valid line numbers are less than 255/$FF, trapping is considered disabled as long as the flag location contains that value. When trapping is enabled, the ERROR routine [$4D3C] will transfer control to the line number indicated here whenever a BASIC error occurs.

4621 $120D TMPTRP

Temporary storage for high byte of TRAP line number

When an error is trapped to a specified line, the ERROR routine [$4D3C] copies the high byte of the target line number from 4620/$120C into this location, then stores the value 255/$FF in 4620/$120C. This disables the trapping of errors during the error-handling routine, which would otherwise put the program into an infinite loop. The value here is copied back into 4620/S120C during execution of the RESUME statement [$5F62].

4622-4623 $120E-$120F ERRTXT

Pointer to start of statement where last error occurred
Whenever an error occurs, the ERROR routine [$4D3C] copies the value in 4610-4611/$1202-$1203 into these locations. The HELP subroutine that highlights the portion of the line where the error occurred uses the value here to determine where to begin the highlighting. The RESUME routine [$5F62] uses the value here to determine where to resume execution.

4624-4625 $1210-$1211 TEXT_TOP

End-of-program pointer
These locations contain the address of the location immediately following the end of BASIC program text. The NEW statement [$51D6] initializes the value here to two bytes beyond the address in the start-of-program pointer (45-46/$2D-$2E). The value here is updated to reflect the new ending address whenever a line is added or deleted from the program. An OUT OF MEMORY error occurs if the value here ever exceeds the value in 4626-4627/$1212-$1213. Following a LOAD or DLOAD, these locations are set to one byte beyond the last location to which data was loaded. For a SAVE or DSAVE, the value here determines the last address from which data will be saved.

4626-4627 $1212-$1213 MAX_MEM_0

Top-of-BASIC pointer
The value in this pair of locations determines the top of free memory in block 0 RAM. The address will be one location beyond the highest one available for BASIC program text. An OUT OF MEMORY error will occur when the value in locations 4624-4625/$1210-$1211 exceeds the value here. The BASIC cold-start initialization subroutine [$4045] writes the value 65280/$FF00 here, so that all block 0 RAM below the MMU registers is available for program text. You can reserve an area at the top of program memory by reducing the value in these locations. Unlike some of the other pointers, you need only store the new value here; no subsequent NEW or CLR is required.

4628-4629 $1214-$1215 TMPTXT

Temporary text pointer storage for DO
This pair of locations is used to temporarily hold the CHRGET text pointer value (from 61-62/$3D-$3E) during execution of the DO statement [$5FE0].

4630-4631 $1216-$1217 TMPLIN

Temporary line number storage for DO
This pair of locations is used to temporarily hold the current line number value (from 59-60/$3B-$3C) during execution of the DO statement [$5FE0].

4632-4634 $1218-$121A USRPOK

USR function jump vector
The BASIC function USR calls a machine language routine, like SYS, but it also allows a numeric value to be passed to and from the ML routine. The routine in BASIC ROM that executes USR ends with a JMP $1218. Location 4632/$1218 contains 76/$4C, the machine language JMP instruction. Locations 4633-4634/$1219-$121A should contain the address of the target machine language routine (in the usual Iow-byte/high-byte format). You must explicitly load these locations with the address of the target routine before you use USR. This location is initialized to 32040/$7D28 during BASIC cold start. This is the address of the routine that issues the ILLEGAL QUANTITY ERROR message, which is what you'll get if you use USR without changing locations 4633-4634/$1219-$121A. Refer to Chapter 5 for more information on passing values to and from the called routine.

4635-4639 $121B-$121F RNDX

Random number seed value
This five-byte area holds the seed value for BASIC'S random-number-generator routine [$8434]. When a positive argument is supplied, the RND routine generates the next random number by performing calculations and manipulations with the value here. The generated values aren't really random—any given seed value here will always produce the same result. However, the process is sufficiently complicated that the results aren't easily predictable. Whenever any random number is generated, the resulting value is stored here for possible use

as the seed for the next random number. Location 4635/$121B is initialized to 0/$00 during the BASIC cold-start routine. That is a change from previous Commodore models, where all five bytes of the seed value were initialized. The zero byte has the effect of making the initial seed value 0, so the first random number value generated after the computer is turned on or after a reset will always be 1.07870447E-03.

4640 $1220 CIRCLE_SEGMENT

Degrees between segments for CIRCLE routine

This location is used during the BASIC CIRCLE statement routine [$668E] to hold the number of degrees to turn for each segment of the circle. The value here is set from the ninth parameter in the CIRCLE statement, and defaults to 2 if that parameter is omitted.

4641 $1221 DEJAVU1

Although Commodore literature states that this location holds a value relating to the reset status, no reference to this location occurs in any ROM routine.

4642 $1222 TEMPO-RATE

Tempo setting for PLAY statement

The value here determines the tempo for notes played by the BASIC PLAY statement. The value here is subtracted from the sound duration value for each voice (in locations 4643-4648/$1223-$1228) during each pass through the BASIC IRQ routine. The larger the value here, the faster the duration decreases and the faster each note plays. The value here is initialized to 16/$10 during the SID initialization routine [$4112], part of both the BASIC cold start and warm start sequences. In BASIC, the TEMPO statement can be used to change the value here.

4643-4648 $1223-$1228 VOICES

Durations for currently active notes

These locations hold the durations of the current PLAY statement notes for each of the SID chip voices:

Voice 0: 4643-4644/$1223-$1224
Voice 1: 4645-4646/$1225-$1226
Voice 2: 4647-4648/$1227-$1228

Bit 7 of each of the high bytes (4644/$1224, 4646/$1226, and 4648/$1228) is used to indicate whether a note is currently being played by that voice. The voice is active when that bit is %0. The duration value for each active voice is decremented by the tempo value specified in location 4642/$1222 during each pass through the BASIC IRQ routine [$A84D]. When a duration is decremented below $0000, the high byte will roil over to $FF, which will set bit 7 to %1, marking the end of the note. At this point, the gate bit for the voice will be turned off, stopping sound output for that voice. Large tempo values cause the value here to decrease more rapidly, increasing the speed at which notes are played, while small tempo values increase the note time.

The high bytes for all three voices are set to 255/$FF by the SID initialization routine [$4112], part of both the BASIC cold start and warm start sequences. This makes all voices initially inactive. When the PLAY statement plays a note, the duration for that note will be copied from 4649-4650/$ 1229-$122A into the slot for the voice specified for that note.

4649-4650 $1229-$122A NTIME

Duration of current note

When the PLAY statement prepares a note, the duration for the note is first calculated in this location, then transferred to the proper slot in 4643-4648/$1223-$1228. The value here is set to 288/$0120, the value for a quarter note, by the SID initialization routine [$4112], part of both the BASIC cold start and warm start sequences.

4651 $122B OCTAVE

Octave for current note

The value in this location determines the octave for the current notes played by the PLAY statement. This value will affect the calculation of the frequency for the notes. The value here is set to 4/$04, the octave containing middle C, by the SID initialization routine [$4112], part of both the BASIC cold start and warm start sequences. The octave value here remains in effect until changed by an O parameter in the PLAY string.

4652 $122C SHARP

Sharp/flat flag

The value in this location holds a value that indicates whether the current note will be either sharp or flat. The location normally holds 0/$00 for natural notes. When a # character is found in the PLAY string, this location will be set to l/$01 to indicate that the next note should be sharp. When a $ character is found in the string, this location will be set to 255/$FF to indicate that the next note should be flat.

4653-4654 $122D-$122E PITCH

Frequency for current note

When the PLAY statement prepares a note, the frequency for the note is calculated in these locations before being transferred into the SID chip registers for the specified voice. The frequency is calculated by loading a base frequency for the specified note from the table at 28665-28688/$6FF9-$7010, adjusted for the octave specified in 4651/$122B. If the flag at 4652/$122C indicates that the note is to be sharp or flat, the frequency is adjusted accordingly.

4655 $122F VOICE

Voice number for current note

The value in this location specifies which voice will be used to play the next note. The value here is set to 0/$00 by the SID initialization routine [$4112], part of both the BASIC cold start and warm start sequences. This selects voice 0 as the default voice. The value here will remain in effect until changed by a V parameter in the PLAY string. The parameter value will be reduced by 1 to convert the BASIC voice number (1-3) to a VIC voice number (0-2).

4656-4658 S1230-S1232 WAVE

Waveforms for current notes

The values in these locations determine which waveforms will be used for each of the three voices:

Voice 0: 4656/$1230
Voice 1: 4657/$1231
Voice 2: 4658/$1232

These locations hold the waveform value for the instrument specified for the voice. All three voices are initialized to the default value for instrument 0. This selects a default pulse waveform for all three voices. The value here remains in effect until changed by a T parameter in the PLAY string. The T parameter causes the value for the current voice to be changed to the waveform value for the specified instrument from the table at 4671-4720/$123F-$1270.

4659 $1233 DNOTE

Dotted-note flag
The value in this location determines whether the next note will be normal or "dotted." Dotted notes play IV2 times as long as a standard note. This location normally holds 0/S00, but will be set to 35/$23 if a period (.) is found in the play string. In this case, the duration of the next note will be increased by 50 percent.

4660-4663 $1234-$1237 FILTSAV

Temporary storage for filter parameters
The filter parameters are copied here from 4721-4722/ $1271-$1272 at the beginning of the FILTER statement routine [$7046], The filter parameter manipulations are then performed on these locations, and the results are copied back to the working storage area.

4664 $1238 FLTFLG

Filter type index
This location is used as a mask value to select individual filter control bits when evaluating the FILTER statement parameters.

4665 $1239 NIBBLE

Temporary storage
This location is used as working storage by the FILTER and ENVELOPE routines.

4666 $123A TONNUM

Current instrument number
This location will hold the instrument number specified in the most recent ENVELOPE statement.

4667-4669 $123B-$123D TONVAL

Envelope parameters for current instrument

The current parameters for the specified instrument are read from the instrument table into these locations at the beginning of the ENVELOPE routine [$7OC1]:

Attack/decay: 4667/$123B
Sustain/release: 4668/$123C
Waveform: 4669/$123D

If the ENVELOPE statement specifies new values for any of these parameters, the new values replace the original values here. The values here are then copied back into the table entries for the specified instrument number.

4670 $123E PARCNT

Index into instrument table for current instrument

This value is used during the ENVELOPE routine [$7OC1] to hold the index to the current set of instrument parameters.

4671-4720 $123F-$1270

Instrument parameter tables

This area is used to hold the envelope parameters for the ten defined instruments supported by the PLAY statement [$6DE1]:

Instrument	Attack/ Decay	Sustain/ Release	Waveform	Pulse width low byte	high byte
0	4671/S123F	4681/S1249	4691/S1253	4701/$125D	4711/S1267
1	4672/$1240	4682/$124A	4692/$1254	4702/$125E	4712/S1268
2	4673/$1241	4683/$124B	4693/S1255	4703/$125F	4713/$1269
	4674/$1242	46S4/S124C	4694/$1256	4704/S1260	4714/$126A
4	4675/$1243	4685/$124D	4695/$1257	4705/J1261	4715/S126B
5	4676/$1244	4686/$124E	4696/$1258	4706/S1262	4716/$126C
6	4677/S1245	4b87/$124F	4697/S1259	4707/S1263	4717/$126D
7	4678/S1246	4688/51250	469S/$1Z5A	4708/$1264	4718/S126E
8	4679/$1247	4689/$125I	4699/S125B	4709/$1265	4719/S126F
9	4680/S1248	4690/$1252	4700/S125C	4710/$1266	4720/S1270

All three voices are initially assigned the envelope parameters for instrument 0. These settings remain in effect until changed with a T parameter in the PLAY string.

The values for any table entry can be changed using the ENVELOPE statement. Default instrument table values are copied into this area from a table in ROM at 28689-28728/ $7011-$7038 during the SID initialization routine [$4112], part of both the BASIC cold start and warm start sequences. The default values are as follows:

Envelope	Attack/ Decay	Sustain/ Release	Waveform	Pulsewidth
0 (piano)	9/$09	0/$00	65/$41 (pulse)	1791/$06FF
1 (accordion)	192/$C0	192/$C0	33/$21 (sawtooth)	0/$0000
2 (calliope)	0/$00	240/$F0	17/$ 11 (triangle)	255/SOOFF
3 (drum)	5/$05	80/$50	129/S81 (noise)	0/$0000
4 (flute)	148/$94	64/$40	17/$11 (triangle)	255/$00FF
5 (guitar)	9/$09	33/S21	33/$21 (sawtooth)	0/$D000
6 (harpsichord)	9/$09	0/$00	65/$41 (pulse)	767/$02FF
7 (organ)	9/$09	144/$90	65/$41 (pulse)	2048/$0800
8 (trumpet)	137/$89	65/$41	65/$41 (pulse)	767/$02FF
9 (xylophone)	9/$09	0/$00	17/$11 (triangle)	0/$0000

Note that the pulsewidth values here are different from those specified in Commodore literature. The official values assume that all pulsewidth low bytes will be 0/$00. However, these bytes are not explicitly initialized, so they will hold their previous values after a reset. On power on, alternating pulsewidth low-byte locations will hold 255/$FF instead of 0/$00.

4721-4722 $1271-$1272 FILTERS

Current filter cutoff frequency

These locations hold the current cutoff frequency register setting, an 11-bit value divided among the two locations with bits 0-2 of the value in location 4721/$1271 and bits 3-10 of the value in location 4722/$1272. The value is copied to the SID cutoff frequency registers (54293-54294/$D415-$D416) when the XI parameter is included in the PLAY string.

4723 $1273

Current filter control and resonance setting

Bits 0-3 of this location determine which voices will be filtered and bits 4-7 control the filter resonance setting. The resonance setting can be changed using the FILTER statement. When an XI parameter is included in the PLAY string, the filter control bit in this location corresponding to the current voice will be set to %1 and the value here will be copied into the SID register at 54295/SD417 to enable filtering of that voice. When an X0 parameter is included, the corresponding voice bit will be set to %0 and the value will again be copied to the SID register to turn off filtering for that voice.

4724 $1274

Current filter type selection
Bits 4-6 of this location determine the type of filtering currently enabled. The setting of those bits can be changed using the FILTER statement. Because the SID register that controls filter type also controls volume, bits 0-3 of this location reflect the current volume setting as well. When an XI parameter is included in a PLAY string, the value here is copied into the SID register at 54296/$D418 to enable the specified filter type. This location is set to 15/$0F, the value for all filters off and maximum volume, during the SID initialization routine [$4112], part of both the BASIC cold start and warm start sequences.

4725 $1275

Current SID chip volume setting
Bits 0-3 of this location reflect the current volume setting for the SID chip. The value here can be changed either with the VOL statement or with the U parameter in a PLAY string. Because the SID register which controls volume also controls filter type selection, bits 4-6 of this location will reflect filter type as well. The value here is set to 15/$0F, the value for maximum volume, during the SID initialization routine [$4112], part of both the BASIC cold start and warm start sequences.

4726-4728 $1276-$1278 INT_TRIP_FLAG

Collision flags
The VIC internal interrupt register (53273/$D019) is read during each pass through the BASIC IRQ routine [$A84D] to determine if a sprite collision has occurred or if a new light pen value has been latched. Because that register is automatically cleared after a read, the system uses these locations to record which conditions were detected:

Location	Collision type
4726/$1276	Sprite-sprite collision
4727/S1277	Sprite-foreground collision
4728/S1278	Light pen latch

If a collision has occurred, the corresponding flag location will be set to 255/$FF, but only if the collision type has been enabled by setting to %1 the appropriate bit in location 4735/ $127F. All three of these locations are set to 0/$00 during the

SID initialization routine [$4112], part of both the BASIC cold start and warm start sequences. A location set to 255/SFF will be reset to 0/$00 after COLLISION processing in the GONE routine [$4A9F].

4729-4734 $1279-$127E INT_ADR

Target line numbers for COLLISION

If the trapping of sprite collisions or light pen latches is enabled, a BASIC subroutine will be called (effectively a GOSUB) whenever one of the selected events occurs. These locations are used to hold the number (in low-byte/high-byte integer format) of the starting BASIC program line of the subroutine to be called. (The subroutine should end with a RETURN statement.) The values here can be set with the COLLISION statement.

Low byte	High byte	Collision type
4729/$1279	4732/$127C	sprite-sprite
473O/$127A	4733/$127D	sprite-foreground
4731/$127B	4734/S127E	light pen

4735 $127F INTVAL

Collision enable flag

Bits 0-2 of this location indicate the collision types for which trapping is currently enabled:

Bit	Collision type
0	sprite-sprite
1	sprite-foreground
2	light pen

Trapping is enabled if the bit is set to %1. Enabling trapping will allow the corresponding collision type to be recorded in the flags at 4726-4728/$1276-$1278. Bits 3-7 of this location are unused. The value here is reset to 0/$00 during the BASIC cold-start sequence. This disables all COLLISION branching.

4736 $1280 COLTYP

Collision type index

The value here is used during the COLLISION routine [$7164] to hold an index into the line number table at 4729-4734/ $1279-$127E.

4737 $1281 SOUND-VOICE

Voice for current SOUND statement

The value in this location specifies which group of entries in the following table should be loaded with the current SOUND parameters. The value here is set to the value of the first parameter in the SOUND statement, minus 1 to convert the BASIC voice number (1-3) into a SID voice number (0-2).

4738-4770 $1282-$12A2

Table of SOUND statement settings

These locations hold the current SOUND parameters for the three SID chip voices:

Parameter	Voice 0	Voice 1	Voice 2
Duration (low)	4738/$1282	4739/$1283	4740/$1284
(high)	4741/$1285	4742/$2286	4743/$1287
Frequency (low)	4744/S1288	4745/$1289	4746/$128A
(high)	4747/5128B	4748/$128C	4749/$128D
Minimum frequency (low)	4750/$128E	4751/$128F	4752/$1290
(high)	4753/$1291	4754/$I292	4755/$1293
Step direction	4756/$1294	4757/$1295	4758/$1296
Step size (low)	4759/51297	4760/51298	4761/$1299
(high)	4762/$129A	4763/$129B	4764/$129C
Current frequency (low)	4765/$129D	4766/$129E	4767/$129F
(high)	4768/$12A0	4769/$12Al	4770/$12A2

Bit 7 of each of the duration high-byte values (locations 4741-4743/$1285-$1287) is used to indicate whether any SOUND statement is active for the corresponding voice. If the bit is %0, the voice is assumed to have an active sound, and the current frequency value for the voice will be copied into the SID chip frequency registers during each pass through the BASIC IRQ routine [$A84D]. See Chapter 5 for a discussion of the other SOUND effects, such as frequency sweeps. Also during each pass through the interrupt routine, the duration value for each active voice will be decremented. When the value is decremented below $0000, the high byte will roll over to 255/$FF, setting bit 7 to %1, which marks the end of the sound. At this point the gate bit for the voice will be set to %0 to turn off the sound.

Each of the duration high-byte locations will be set to 255/$FF during the SID initialization routine, part of both the BASIC cold start and warm start sequences. This will turn off

SOUND output for all three voices. The values in these locations are updated when the contents of 4771-4776/$ 12A3-$12A8 are copied to the entries for the specified voice during execution of the SOUND statement [$71EC].

4771-4776 $12A3-$12A8

Parameters for most recent SOUND statement

These locations are used to assemble the parameters for the current SOUND statement. The SOUND statement must include voice number, frequency, and duration parameters. The remaining parameters are optional; if they are omitted, default values are supplied. The base frequency value is initialized to the specified starting frequency. The locations and default values are as follows:

Parameter	Locations	Default value
Duration	4771-4772/$12A3-$12A4	
Frequency	4773-4774/$12A5-$12A6	
Minimum frequency	4775-4776/$ 12A7-S12A8	0/$0000
Step direction	4777/$12A9	0/$00 (sweep up)
Step size	4778-4779/$ 12AA-S12AB	0/$0000 (no sweep)
Base frequency	4780-4781/$12AC-$12AD	
Pulsewidth	4782-4783/$12AE-$12AF	2048/S0800
Waveform	4784/$12B0	2/$02 (pulse)

After all the parameters for a SOUND statement have been evaluated and assembled here, the values are transferred to the entries in the table at 4738-4770/$1282-$12A2 for the specified voice. The base frequency value is used as the beginning current frequency value.

4785 $12B1 POT_TEMP_1

Temporary storage for POT and PEN routines

This location is used as temporary storage during the routines to perform the BASIC functions POT [$824D1 and PEN [$82AE].

4786 $ 12B2 POT_TEMP_2

Temporary storage for POT routine

During execution of the POT function routine [$824D], the potentiometer reading from the SID chip register is stored here temporarily while the paddle buttons are being read.

4787-4790 $12B3-$12B6 WINDOW_TEMPS

Temporary parameter storage for WINDOW statement
When the WINDOW statement [$72CC] is executed, the parameter values associated with the statement are stored in these locations before the screen editor WINDOW routine [$C02D] is called to actually set the new window margins.

Location	Parameter
4787/$12B3	left column
4788/$12B4	top row
4789/$12B5	right column
4790/$12B6	bottom row

4791-4806 $12B7-$12C6

Filename buffer for DOS support commands
The routine which handles the BASIC 7.0 DOS support commands such as SCRATCH and RENAME copies the filename portion of the command here temporarily while the remainder of the command is being processed. Once the command is set up, the filename here is copied into the DOS command buffer at 4352-4399/$1100-$112F.

4791-4853 $12B7-$12F5 SAVRAM

Sprite pattern storage
These locations are used during the SPRDEF statement routine [$7372] to hold the original sprite pattern while a sprite is being defined. If the STOP key is pressed to cancel the current modifications, the pattern definition here will be restored to the definition area for the sprite. The first 63 of these locations are also used during the SPRSAV routine [$76EC] to hold the sprite pattern to be transferred to a string variable.

4854-4857 $12F6-$12F9

Sprite pattern suffix
During the SPRSAV routine [$76EC], these locations are initialized with the pattern $17 $00 $14 $00. When a sprite pattern is saved in a string variable, these bytes are appended to the sprite pattern in 4791-4853/$12B7-$12F5 before the data is transferred to the string pool. Two bytes are needed as the tag for the variable, but exactly what this four-byte pattern is intended to achieve is unclear.

4858 $12FA DEFMOD

Sprite mode indicator for SPRDEF
This location is used during the SPRDEF statement routine [$7372] to hold a value indicating the mode of the sprite currently being defined. A value here of 0/$00 indicates a standard sprite, while a value of 128/$80 indicates a multicolor sprite.

4859 $ 12FB LINCNT

Sprite pattern line count for SPRDEF
This location is used during the SPRDEF statement routine [$7372] to hold the number of the vertical line (0-20) within the sprite pattern which is currently being defined.

4860 $12FC SPRITE_NUMBER

Sprite number for SPRDEF
This location is used during the SPRDEF statement routine [$7372] to hold the number (0-7) of the sprite currently being defined.

4861 $ 12FD IR9_WRAP_FLAG

BASIC IRQ activity flag
This location is tested at the beginning of the BASIC IRQ service routine [$A84D]. If it contains any nonzero value, the routine exits immediately. The location is initialized to 0/$00 during the SID initialization routine [$4112]. The BASIC IRQ routine increments this location (to l/$01) when it begins, so the test prevents the routine from being restarted if another interrupt occurs before the current pass is completed. The IRQ routine resets the value here to 0/$00 before exiting.

The BASIC portion of the IRQ sequence is responsible for moving sprites, detecting sprite collisions, and handling the BASIC sound statements. The routine maintains a number of shadow locations which are copied into VIC and SID chip hardware registers during each interrupt. Sometimes you may want to turn off these shadow locations to have direct access to the hardware registers. One way to do that is to store some nonzero value in this location. While turning off the BASIC IRQ routine will give you direct access to the hardware registers, you should keep in mind that it will also effectively disable the BASIC statements MOVSPR, COLLISION, SOUND and PLAY.

4862-4863 $12FE-$12FF Unused

These locations are not used by any 128 ROM routine.

Application Program Area

4864-7167/$1300-$lBFF

None of the 2304 (2V4-K) locations in this area are used by any system ROM routines. Thus, this area is free for your own programming uses—machine language routines, alternate screens, and so on. Because this is the largest block of free RAM protected from BASIC, the area is becoming extremely popular with machine language programmers, much like the area at 49152/$C000 in the Commodore 64. As a result, you'll probably encounter instances where two programs you want to use simultaneously will be incompatible because they reside at overlapping addresses within this range.

One thing this area cannot normally be used for is to hold additional sprite patterns or custom character patterns. While the standard ROM-based character sets are enabled, the VIC chip will see character ROM at addresses 4096-8191/$1000-$lFFF. As a result, this RAM is not visible to the VIC chip and cannot be used for sprite or character information. Sprite or character patterns can be stored here if the ROM-based characters are disabled; refer to the entry for location l/$01 in Chapter 2 for details.

RAM Usage

The Commodore 128, as its name implies, has 128K of primary RAM in two 64K blocks. Memory configurations are discussed in detail in Chapter 1, but in general the 128 sees RAM from block 0 in even-numbered banks (0, 4, 8, 14) and RAM from block 1 in odd-numbered banks (1, 5, 9). A notable exception is bank 15, where RAM from block 0 is seen. Another significant exception is that in every bank the system normally sees RAM from block 0 in locations 2-1023/$0002-$03FF. (Remember that locations 0-l/$00-$01 are used for the processor's on-chip I/O port and are never seen as RAM.) This means that the lowest IK of RAM in block 1 normally remains invisible and unused. As explained in Chapters 2 and 3, the common IK block and locations 1024-7167/$0400-$lBFF in block 0 have special uses. Also, remember that MMU registers, rather than RAM or ROM, are seen at addresses 65280-65284/$FF00-$FF04 in every bank configuration.

Two pointers in page 10/$0A indicate the range of locations in block 0 considered free RAM. Locations 2565-2566/$0A05-$0A06 point to the lowest free address, and locations 2567-2568/$0A06-$0A07 point to one byte beyond the highest free address. These pointers are initialized during the RAMTAS subroutine [$E093], part of the reset sequence, to 7168/S1C00 and 65280/$FF00, respectively. The pointer values can also be changed with the Kemal MEMTOP [SFF99] and MEMBOT [$FF9C] routines. However—unlike earlier Commodore computers—these pointers have no effect on the range of addresses used by BASIC and are not read by any other Kernal or BASIC routine.

BASIC RAM Usage

For BASIC programming, the areas of RAM normally available for storage of programs and variables are locations 7168-65279/$1COO-$FEFF in block 0 and 1024-65279/$0400-$FEFF in block 1. This is a total of 122,368 bytes of available RAM space (illustrated in Figure 4-1). This explains why part of the message you see when you turn on or reset the computer says 122365 BYTES FREE. (The missing three bytes are to account

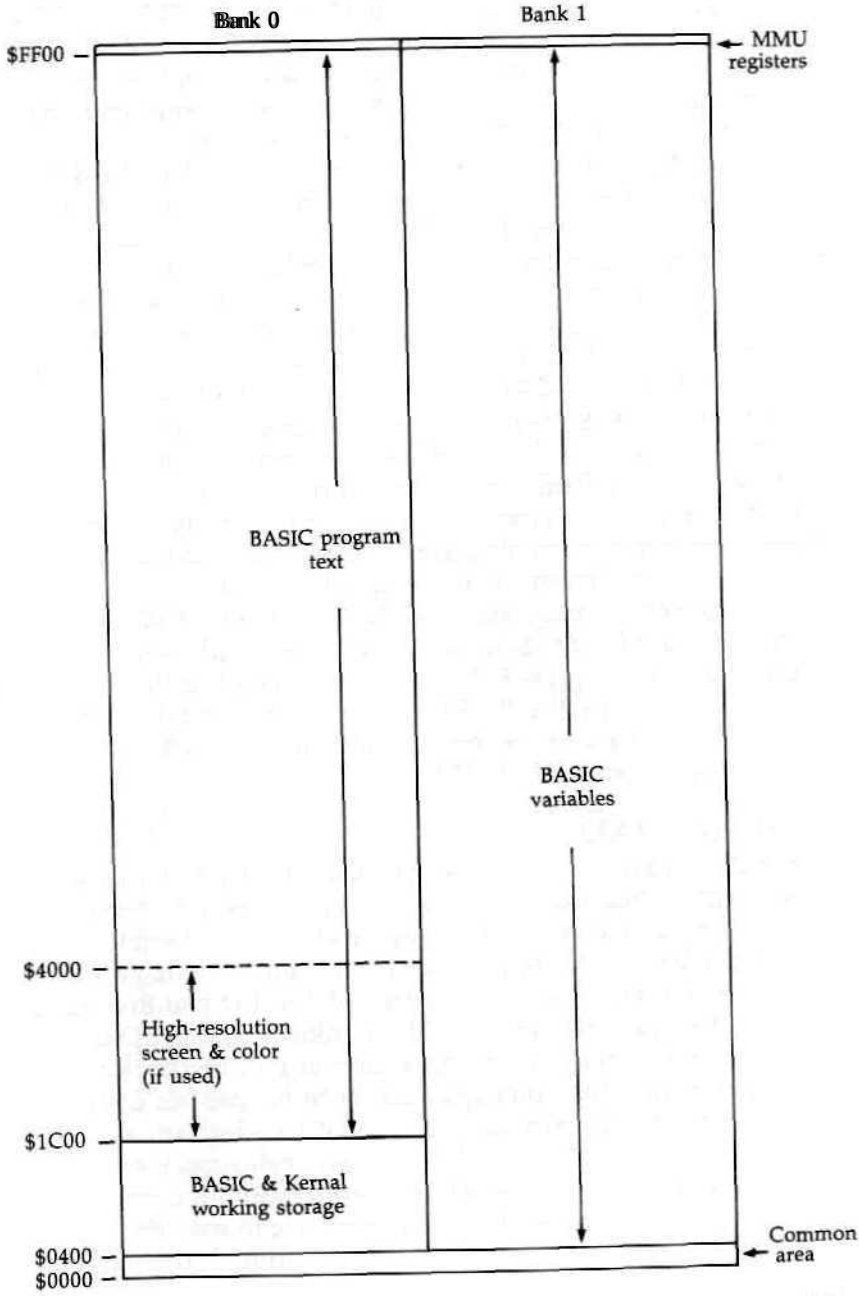
for the zero byte required by BASIC before the first program line and the two zero bytes used to mark the end of the program.)

Actually, it's a bit misleading to claim that many free bytes, since you can't write a BASIC program 120,000 bytes long. For BASIC, the free RAM is divided into two distinct segments: the 58,112 bytes in block 0 for BASIC program text and the 64,256 bytes in block 1 for variables and strings. (By comparison, the Commodore 64 offers 38,911 bytes for program text and variables combined.)

As noted in Figure 4-1, there is one additional factor which affects the amount of memory available for program text. When you use a GRAPHIC statement to set up a high-resolution screen, an additional 9K is reserved in block 0: IK at 7168-8191/$1COO-$1FFF for color information and 8K at 8192-16383/$2000-$3FFF for the high-resolution-screen bit-map. In this case, the amount of RAM available for BASIC program text is reduced to 48,896 bytes (locations 16384-65279/$4000-$FEFF in block 0). If a program is already in memory when the GRAPHIC statement is executed, the program is moved upward in memory (the starting address will be changed from 7169/$1CO1 to 16385/$4001) and relinked to work at the new addresses. Once a high-resolution memory area is established, it remains allocated until a GRAPHIC CLR statement is executed, at which time the program text is moved down to start at 7169/$1CO1 again.

Pointers in zero page and page 18/$ 12 are used to specify the length of program text and variables. BASIC program text is assumed to begin at the address in block 0 specified in locations 45-46/$2D-$2E. That pointer is initialized to 7169/$1CO1 during the BASIC cold start routine [$4023]. Unlike the Commodore 64, which sets its start-of-BASIC pointer according to the value in the system's start-of-free-memory pointer, the 128 sets the address value without regard for the value in 2565-2566/$0A05-$0A06. Locations 4626-4627/$1212-$1213 point to one byte beyond the highest available address in block 0. That pointer is initialized during BASIC cold start to 65280/$FF00, again without regard to the Kernal memory pointer value in 2567-2568/$0A07-$0A08. The actual ending address of the program text currently in memory is specified by the value in 4624-4625/$1210-$1211. That pointer is initialized during the BASIC CLR routine [$51F8] to two bytes

Figure 4-1. BASIC RAM Usage



beyond the starting address in 45-46/$2D-$2E. An OUT OF MEMORY error occurs if the address in 4624-4625/ $1210-$1211 reaches the value in 4626-4627/ $1212-$1213. The ending address pointer is set after a BASIC LOAD [$912C], and the BASIC SAVE routine [$9112] uses the values in the starting and ending address pointers as the starting and ending address for the block of memory to be saved.

The address in the pointer at locations 47-48/$2F-$30 marks the start of scalar (nonarray) variables in bank I. The pointer is initialized to 1024/$0400 during the BASIC cold start routine. A pointer at 49-50/$31-$32 marks the end of scalar variables and the beginning of arrays; another pointer at 51-52/$33-$34 marks the end of arrays and the beginning of free memory in block 1. Both of these pointers are reset to the value in 47-48/$2F-$30 during the BASIC CLR routine. The pointer at 57-58/$39-$3A holds an address which is one byte beyond the highest address of free memory in block 1. It is initialized during BASIC cold start to point to 65280/$FF00. The free memory in block 1 is used to hold strings of all types—constants, variables, and arrays. The string pool starts at the top of free memory and is filled downward toward the bottom of free memory indicated in 51-52/$33-$34. The pointer at 53-54/$35-$36 marks the current address of the bottom of the string pool. That pointer is reset to the value in 57-58/$39-$3A by the BASIC CLR routine. An OUT OF MEMORY error occurs when the value in 53-54/$35-$36 reaches the value in 51-52/$33-$34.

Reserving RAM

There are occasions when you will want to divert an area of RAM from its normal usage. For example, you may need to set aside space for a machine language routine, an alternate screen display, or a data buffer. For machine language (ML) programming, you can use any area of RAM if you are willing to learn the intricacies of the 128's banking scheme. Otherwise, it's best to restrict your programming to certain known areas. For a machine language routine to be used in conjunction with a BASIC program, you'll need to select an area which BASIC doesn't normally use, or to take away some memory that otherwise would be used for program text or variable storage. As noted in Chapter 3, locations 4864-7167/ $13OO-$1BFF are currently unused (even though they are

called "reserved" in Commodore literature). This 2304-byte area is the largest unused area of protected RAM in the 128, and it is becoming extremely popular with 128 ML programmers—much like the $C000 block in the Commodore 64. You can expect to see many ML programs using this area.

Other, shorter blocks are also available if certain BASIC features are not used. If tape is not used, the 256 bytes at 2816-3071/$0B00-$0BFF are available. However, unlike other free blocks, this page may be overwritten during a reset because disk boot sectors are read into this area. Thus, the time-honored Commodore tradition of using the cassette buffer for short ML routines is less suitable in the 128. (It's annoying to have to reload your routine after each reset.) If your program doesn't use RS-232 communications, the two RS-232 buffers at 3072-3583/$0C00-$0DFF provide a 512-byte workspace. This is probably the best area for short ML routines that you wish to use in conjunction with BASIC. (Unlike the cassette buffer, this area survives reset intact.) If your program does not use sprites, the 512-byte sprite definition area at 3584-4095/ $0E00-$0FFF is also available. Of course, if your program uses neither tape nor RS-232 nor sprites, you can use the full 1280 bytes at 2816-4095/$0B00-$0FFF or any subsection thereof.

To use a large ML program in conjunction with BASIC, there is an easy way to reserve over UK of protected RAM. However, this technique works only if neither the BASIC nor the ML program requires high-resolution graphics. The trick is to use the BASIC GRAPHIC statement to set aside a high-resolution screen area at 7168-16383/$lC00-$3FFF. As mentioned above, this area remains allocated until a GRAPHIC CLR statement is executed. Simply begin your BASIC program with a line like GRAPHIC 1:GRAPHIC 0 (or GRAPHIC 1:GRAPHIC 5 if you want to use the 80-column display). Then BLOAD the machine language program into the reserved area. In addition to the 9K screen area, you can also use the contiguous unused area just below, at 4864-7167/$1300-$lBFF. If you want to use a machine language program in conjunction with BASIC and high-resolution graphics, you'll have to resort to bank-switching techniques if the program is too large to fit in the unused area at 4864/$1300.

It's possible to reserve space above or below either the BASIC or variable/string areas. To reserve space below the BASIC program text, increase the value in the start-of-BASIC

pointer at 45-46/$2D-$2E by the number of bytes you want to reserve. (To reserve an even number of 256-byte pages, you need only change the value in 46/$2E.) Two other steps are also necessary: BASIC requires a zero byte below the first location in its program text space, and a NEW operation is required to reset other important memory pointers. For example, to reserve three pages (768 bytes) below the normal start of BASIC, you would use a statement like this:

```
POKE 46,31:POKE 31*256,0:NEW
```

After this statement is executed, the area at 7168-7935/$1COO-$1EFF is protected from BASIC until the next time the BASIC cold start routine is performed (normally during the next reset sequence). The pointer value is unaffected by RUN/STOP-RESTORE. This technique is less useful when a high-resolution screen area is allocated. In that case, the start of BASIC is moved to 16384/$4000. The technique for reserving space at the start of BASIC still works, but the reserved memory will lie above 16383/$3FFF, which is the highest address seen as RAM in bank 15—the bank in which Kernal ROM is visible and to which BASIC defaults. Thus, a routine above that boundary will be invisible unless you tinker with the MMU configuration register.

Space can be reserved at the top of the BASIC program area by reducing the value in the pointer at 4626-4627/$1212—$1213 by the desired number of bytes. (Again, if you wish to reserve an even number of 256-byte pages, you can simply reduce the value in 4627/$l213.) No additional steps are required other than changing the pointer value. This technique was often used in the Commodore 64 to reserve space for machine language routines; its usefulness is more limited in the 128 because of the 16384/$4000 boundary of RAM visible in bank 15, which was mentioned above. To easily use the reserved area for an ML routine in conjunction with BASIC, the top of memory must be lowered sufficiently to make at least a portion of the reserved area appear below the boundary of RAM visible in bank 15; this dramatically reduces the amount of memory available for program text. It's not even possible when a high-resolution screen area is allocated. The technique can, however, be useful for setting aside an area in block 0 for a buffer, a reserved area of memory for data storage.

You can also reserve space in block 1, either above or below the variable/string area. To reserve space below variables, add a value corresponding to the number of bytes to be reserved to the address in the pointer at 47-48/$2F-$30, (As with the other pointers, you can simply increase the value in 48/$30 if you are reserving an even number of 256-byte pages.) This step must be followed by a BASIC CLR statement to reset other variable pointers, so it should be performed early in the program (CLR erases all variable values). The following line reserves an additional IK at the bottom of variable space, locations 1024-2047/$0400-$07FF in block 1:

```
100 POKE 48,8:CLR
```

Once established, the reserved area will remain intact until the next time the BASIC cold start routine is executed, normally at the next reset. The setting is unaffected by RUN/STOP-RESTORE.

Since this reserved RAM is in block 1, it can't be used for ML routines as easily as the RAM from block 0. There is no standard bank configuration that makes BASIC and Kernal ROM visible in conjunction with block 1 RAM. Of course, it is possible to access Kernal or BASIC routines indirectly by using the JSRFAR or JMPFAR routine. One use for a reserved area in block 1 would be for an alternate 40-column screen. See the entry for the MMU RAM configuration register (54535/$D506) information on using block 1 for VIC-II screen memory.

To reserve space above strings, subtract a value corresponding to the number of bytes to be reserved from the address in the pointer at 57-58/$39-$3A. (As with the other pointers, you can simply increase the value in 58/$3A if you are reserving an even number of 256-byte pages.) This step must also be followed by a BASIC CLR statement to reset other string pointers, so it should be performed early in the program (CLR erases all variable values). The following line reserves 31K at the top of string space, locations 32768-65279/$8000-$FEFF in block 1:

```
100 POKE 58,128:CLR
```

Once established, the reserved area will remain intact until the next time the BASIC cold start routine is executed—normally at the next reset. The setting is unaffected by RUN/STOP-RESTORE. As mentioned above, this area can't be easily used for machine language routines since it is in block 1. One ap-

propriate use for a reserved area here would be for a data buffer—to hold downloaded text in a telecommunications program, for example.

Using ML Without BASIC

You have several options when using ML programs alone, without BASIC. The simplest, if your program is less than 9K (9216 bytes) long, is to leave the system in its default bank 15 configuration and use the visible area of block 0 RAM at 7168-16383/$lC00-$3FFF. (If you need a few more bytes, you can stretch the start of the program down to the bottom of the reserved area at 4864/$1300.) With this setup, you have full access to the I/O chip registers and all the routines in BASIC and Kernal ROM.

If you need more space, but still want access to Kernal routines, you can change the settings of bits 1-3 of the MMU configuration register to switch out BASIC ROM. In this case, you'll have access to over 43K of contiguous RAM, locations 4864-49151/$1300-$BFFF. If you want to use a high-resolution screen in conjunction with your ML routine, it's easiest to set up the screen in its normal location (7168-16383/$1COO-$3FFF). This means that—if your program is too long to fit below the screen areas—you'll need to switch out BASIC to have some RAM visible with Kernal ROM. (You could still use the Kernal JSRFAR routine to access BASIC routines—if you wanted to use some of the graphics drawing routines, for example.)

Although it is possible to set up a custom MMU configuration that makes block 1 RAM visible with either BASIC or Kernal ROM (or both), there's rarely a need for such gyrations. It's usually easiest to locate your executable machine language in block 0 and use block 1 for data storage.

Several obscure techniques are available to squeeze a few more bytes out of the 128. For example, you can gain access to the lowest IK of block 1 RAM, which is normally covered by the common area from block 0, by changing the value in the MMU RAM configuration register (54534/$D506). See the discussion of the MMU in Chapter 8 for details.

Page 255/$FF

The highest page of memory, locations 65280-65535/ $FFO0-$FFFF, in each RAM block is normally unused by BASIC and contains a few bytes of free RAM as well as some

important routines and vectors. The MMU configuration and load configuration registers always appear in the lower five bytes of this area, locations 65280-65284/$FF00-$FF04. They should never be disturbed unless you know the effect of the values you are storing there (see Chapter 8 for more information on the MMU). You should also exercise care when changing the contents of locations 65285-65348/$FF05-$FF44 in either RAM block, as these areas contain copies of the interrupt and reset handling routines. (These areas are initialized by the Kernal RESET routine [$E000].) If an interrupt or reset occurs while the system is configured for a bank where Kernal ROM is not visible—bank 0 or 1, for example—a crash will occur if the area in the visible RAM block does not contain a routine to redirect the reset or interrupt to a proper handling routine. See the entries for these addresses in Chapter 9, "Kernal ROM," for more information.

Free space in this page includes the 181 bytes at locations 65349-65529/$FF45-$FFF9 in block 0 and the 176 bytes at 65349-65524/$FF45-$FFF4 in block 1. However, locations 65488-65519/$FFD0-$FFEF in block 0 will be overwritten whenever the computer is reset. As mentioned in Chapter 1, the Z80 microprocessor has control briefly after a reset or when the computer is first powered on. The initialization steps performed by the Z80 include copying two routines into block 0 RAM. One, at 65488-65503/$FFD0-$FFDF, is an 8502 machine language routine to surrender control to the Z80; the other, at 65504-65519/$FFE0-$FFEF, is a Z80 machine language routine to surrender control to the 8502. These routines have no use in 128 mode—they can be used only in CP/M mode—but they are recopied to block 0 during each reset. {Actually, there is one situation where disturbing these routines can cause a problem. If you overwrite the routine at 65488/$FFD0 and then attempt to start CP/M with a BASIC BOOT command, the system will crash. The machine language in the CP/M boot sector expects that routine to be intact.)

Locations 65525-65529/$KFF5-$FFF9 in block 1 have a special use. The first three bytes, locations 65525-65527/$FFF5-$FFF7, are an initialization signature; after the Kernal RESET routine [$E000] has been performed at least once, these locations will contain the character codes for the letters *CBM.*

As long as the signature locations contain these codes, the initialization test subroutine will take an indirect jump to the address specified in locations 65528-65529/$FFF8-$FFF9, called the system vector or soft reset vector. This vector normally points to 57892/$E224 in Kernal ROM, a routine that does nothing more than reinitialize the signature and vector. You can change the vector to point to a routine of your own to add additional steps to the reset sequence or to initiate an entirely new reset sequence. One restriction applies: The routine you specify in the vector must be visible in the bank 15 configuration since that is how the system is set up when the jump through the vector is taken.

When tapping into the RESET routine, you need to be aware of what has happened before the vector jump is taken and what hasn't happened yet. Before entering the subroutine that takes the jump through the vector, the RESET routine [$E000] resets the stack pointer to the top of the stack, configures the system for bank 15, resets the other MMU registers to their default values, and recopies the common routines to 65285-65348/$FF05-$FF44, 674-763/$02A2-$02FB, and 1008-1020/$03F0-$03FC. However, the initialization routines IOINIT, RAMTAS, RESTOR, and CINT are normally called after the return from the jump. This means that you can't use the vector diversion to change default indirect vector settings or to alter screen parameters if your routine ends with RTS to return to the normal reset sequence. It also means that when you use the vector to substitute your own reset sequence, you may need to call one or more of these subroutines to complete system initialization. At least the IOINIT routine [$E109] or some equivalent initialization routine is necessary, since the reset signal generated by pressing the RESET button also resets the VIC and VDC (8563) video chips, clearing all chip registers to zero. IOINIT initializes the video chip registers to their standard settings.

One interesting use of this vector is to make a machine language program unstoppable by anything short of turning off the computer. To accomplish this, change the vector to point to the initialization routine of the program to be made unstoppable. That initialization step should include calls to at

least the IOINIT and CINT routines, and it should also disable RUN/STOP-RESTORE by redirecting the NMI vector. Here is a short example:

```
0C00  LDA  #$F8    ;Use Kernal INDSTA routine to
0C02  STA  $C3     ;   change system reset vector
0C04  LDA  #$FF    ;   in bank 1 to point to the
0C06  STA  $C4     ;   routine at $0C28
0C08  LDA  #$C3
0C0A  STA  $02B9
0C0D  LDA  #$28
0C0F  LDX  #$01
0C11  LDY  #$00
0C13  JSR  $FF77
0C16  LDA  #$0C
0C18  LDX  #$01
0C1A  INY
0C1B  JSR  $FF77
0C1E  LDA  #$33    ;Change the INMI indirect vector
0C20  STA  $0318   ;   to point to the interrupt return
0C23  LDA  #$FF    ;   routine (disables RUN/STOP-
0C25  STA  $0319   ;   RESTORE)
0C28  JSR  $FF84   ;Kernal IOINIT routine
0C2B  JSR  $C000   ;Kernal CINT routine
0C2E  LDX  #$00    ;Loop to print message repeatedly
0C30  LDA  $0C40,X ;   text at $0C40
0C33  BEQ  $0C2E
0C35  JSR  $FFD2
0C38  INX
0C39  BNE  $0C30
;Text for message
>0C40  49 20 43 41 4E 27 54 20
>0C48  42 45 20 53 54 4F 50 50
>0C50  45 44 21 0D0DO0
```

Use J F0C00 {from the monitor) or BANK 15:SYS 3072 (from BASIC) to set the new pointer values and start the routine. Once started, it cannot be stopped with either reset or RUN/STOP-RESTORE. Obviously, you should make sure that your ML program is fully debugged—and be sure that you have a backup copy, just in case it isn't—before you use this technique to make the program unstoppable.

The highest six addresses in each RAM block, locations 65530-65535/$FFFA-$FFFF, contain copies of the processor

reset and interrupt vectors. This area is initialized during the reset sequence, and, like the handling routines to which these vectors point, these vector addresses should be changed with care. The system will crash if a RAM vector does not contain the address of a valid handling routine when an interrupt or reset occurs while that block is visible. See the entries for these addresses in Chapter 9 for more information on the processor vectors.

BASIC ROM

The Commodore 128's BASIC 7.0 occupies the 28K of ROM between 16384-45055/$4000-$AFFF. That represents significant growth from the BASIC 2.0 of the Commodore 64, which filled only about 9K. The expansion is the result of the addition of a variety of graphics, sound, and sprite statements, as well as enhanced commands for disk operations. Because BASIC is so large, it's not practical to provide a detailed description of every routine—that would fill another book. Instead, the entry points to most of BASIC'S important routines are listed, with short explanations of what the target routines do.

Adding to BASIC

Even with all the added features of BASIC 7.0, you may find it lacking and wish to modify BASIC to add new commands. One common way to do this in the Commodore 64 is to copy BASIC ROM into RAM, then modify and use the RAM-based version. This scheme can't be used on the 128, even though it does have RAM under BASIC ROM like the 64. While it's possible to copy BASIC into ROM, there's no easy way to keep a RAM-based version of BASIC executing in RAM. The bank-switching routines in 128 BASIC ROM keep the system configured for banks 14 or 15, where BASIC ROM is visible, while BASIC routines are being executed.

The formal method of adding new statements or functions is to tap into the indirect vectors at 780-785/$030C-$0311 and 764-765/$02FC-$02FD. This allows you to add new statements or functions that use the two-byte extended tokens. Currently, extended statement tokens 39-255/$27-$FF and extended function tokens ll-255/$0B-$FF are unused and thus available for your additional keywords.

Three separate steps are required to add a new keyword. You must provide for it to be tokenized, detokenized (listed), and executed. The indirect vector at 780-781/$030C-$030D lets you patch into the CRUNCH routine to tokenize your new extended token keywords. The vector at 782-783/ $030E-$030F lets you patch into the IQPLOP routine to detokenize these new keywords when a line containing a new

keyword is listed. The other two vectors allow you to patch into the statement and function execution routines to provide for the handling of the new keywords (784-785/$0310-$0311 for statements and 764-765/$02FC-$02FD for functions). The following example shows the addition of a statement and a function. The statement, STORE, can be used to store values in the VDC chip internal registers (see Chapter 8 for details of the VDC chip). The format of the statement is STORE *register, value*. The register parameter is the VDC register number (0-36) and the value parameter is the value (0-255) to be stored in that register. The function, RAD, converts an angle value from degrees to radians, the system used in BASIC functions. The format for the function is *KAD(angle),* where angle is the angle value in degrees. Since this is a function, it must be used on the right side of an operation, as in A = RAD(45).

```
1600  LDA #$29   ;Redirect ICRNCH2 vector to $1629 to
1602  STA $030C  ; tokenize new keywords
1605  LDA #$16
1607  STA $030D
160A  LDA #$5A   ;Redirect IQPLOP2 vector to $165A to
160C  STA S030E  ; list new keywords
160F  LDA #$16
1611  STA $030F
1614  LDA #$7B   ;Redirect IGONE2 vector to $167B to
1616  STA $0310  ; handle new statement
1619  LDA #$16
161C  STA $0311
161E  LDA #$A2   ;Redirect ESC_FN vector to $16A2 to
1620  STA $02FC  ; handle new (unction
1623  LDA #$16
1625  STA $02FD
1628  RTS
                 ;Tokenize new keywords
1629  STA $02    .Stash current character
162B  LDY #$50   ;Search for keyword in table at $1650
162D  LDA #$16
162F  JSR 543E2
1632  BCC $163A  ;If no match found, try other table
1634  ADC #$A6   ;Convert index (with bit 7 set) to token (39/$27)
1636  LDX #$00   ;Set flag for extended statement token
1638  BEQ $1647
163A  LDY #$56   .Search for keyword in table at $1656
163C  LDA #$16
163E  JSR $43E2
1641  BCC $164A  ;Exit if no match found
1643  ADC #$8A   .Convert index (with bit 7 set) to token (11/$OB)
1645  LDX #$FF   ;Set flag for extended function token
```

```
1647   CLC
1648   BCC $164D  ;Exit with carry clear if keyword found
164A   SEC
164B   LDA $02     ;Restore text character
164D   JMP $4321   .Return to ICRNCH routine
>1650 53 54 4F 52 C5 00  ;STORE
>1656 52 41 C4 00        ;RAD
                         ;List (detokenize) new keywords
165A   CPX #$00    ;Was this statement or token?
165C   BNE S166C
165E   CMP #$28    ;Was statement token less than 40?
1660   BCS $1678
1662   LDY #$50    ;Use table entry at $1650 to list
1664   LDA #$16
1666   STY $24
1668   STA $25
166A   BCC $1678
166C   CMP #$0C    ;Was function token less than 12?
166E   BCS $1678
1670   LDY #$56    ;Use table entry at $1656 to list
1672   LDA #$16
1674   STY $24
1676   STA $25
1678   JMP S51CD ;Return to IQPLOP routine
                   ; Handle execution of statement
167B   CMP #$28    ;Was statement token less than 40?
167D   BCS $1685
167F   LDA #$16    ;Put address of execution routine - 1 on stack
1681   PHA         ; (execution routine is at $1866)
1682   LDA #$87
1684   PHA
1685   JMP $4BA9
                   ; STORE routine
1688   JSR $8803   ;Evaluate register number and value parameters
168B   TXA         ;Move value lo accumulator
168C   LDY $17     ;High byte of register number should be zero
168E   BNE $169D
1690   LDX $16     ;Low byte (in X) should be less than 37
1692   CPX #$25
1694   BCS $169D
1696   STY $FF00   ;Set for bank 15 so VDC chip is visible
1699   JSR $CDCC;Use screen editor routine
169C   RTS
169D   LDX #$0E    ;Illegal quantity error if incorrect value supplied
169F   JMP ($0300)
                   ;RAD routine
16A2   CMP #$0C    ;Was token less than 12?
16A4   BCS $16B4
16A6   JSR $7956   ;Check that parameter ended with a closing
                   parenthesis
16A9   LDA #$B5    ;Load value from $16B5 into FAC2
```

```
16AB  LDY  #$16
16AD  JSR  SAF5D
16B0  JSR  $AF21  ^Multiply by argument in FAC1
16B3  CLC
16B4  RTS
>16B5  7B OE FA  35  12     ;Floating point value for TT/180
```

An alternate method of adding new statements to BASIC involves creating intentional errors. To use this scheme, your new keyword must consist of an existing keyword preceded by a letter (or another keyword)—LLIST or COPYCHAR, for example. A syntax error will occur when the new keyword is encountered, but you can use the IERROR indirect vector (768-769/$0300-$0301) to trap the error and process the keyword. The advantage of this technique is that you don't have to worry about tokenizing or detokenizing the new keyword. The following example program illustrates the technique. It supports a new statement, VPOKE, which performs like the STORE statement in the example above. Use VPOKE *register, value* to store a value in a VDC chip register. After a SYS 4864 to patch in this routine, VPOKE can be used in either program or immediate mode, just like any other keyword.

```
1300   LDA #$0B   ;Redirect IERROR vector to $130B
1302   STA $0300
1305   LDA #$13
1307   STA $0301
130A   RTS
130B   CPX #$0B   ;Was this a SYNTAX error?
130D   BNE $1326
130F   CMP #$97   ;If §o, did it occur at a POKE token?
1311   BNE $1326
1313   LDA $3D    ;Calculate pointer to character
1315   SBC #$01   ; immediately before the POKE token
1317   STA $26    ; (If the keyword uses a two-byte
1319   LDA $3E    ; extended token, you must back up
131B   SBC #$00   ; two positions instead of one.)
131D   STA $27
131F   JSR $03C0  ;Retrieve character before token
1322   CMP #$56   ;Was it a V?
1324   BEQ $1329  ;If so, branch to handle VPOKE
1326   JMP $4D3F  ;Process all other errors normally
1329   JSR $0380  ;Move CHRGET pointer beyond token
132C   JSR $8803  .Evaluate parameters following VPOKE
132F   TXA        ;Move value parameter to accumulator
1330   LDY $17    ;Check that register parameter is
1322   BNE $133A  ; less than 37
```

```
1334   LDX  $16       (target register number in X)
1336   CPX  #$25
1338   BCC  $133E
133A   LDX  #$0E    ;If register number is too large,
133C   BNE  $1326   ; exit with ILLEGAL QUANTITY error
133E   STY  $FFO0   ;Configure for bank 15 (Y contains $00)
1341   JSR  $CDCC;Use screen editor register setup routine
1344   JMP  $AF90   ;Continue processing program text
```

The BASIC Jump Table

One new feature of BASIC 7.0 that will be very valuable to machine language programmers is the jump table at 44800-44967/$AFOO-$AEA7. Many of the most useful BASIC routines now have static entry points like those the Kernal jump table provides for Kernal routines. Wherever possible, you should use the jump table entry into the routine to maintain compatibility in the event that BASIC ROM is revised.

BASIC Entry Vectors

16384 $4000 JHARD-RESET

BASIC cold-start entry point; jumps to 16419/$4023, the address of the routine which performs a complete initialization of BASIC. This is the normal entry point following a system reset.

16387 $4003 JSOFT_RESET

BASIC warm-start entry point; jumps to 16393/$4009, the address of the routine which reinitializes BASIC and Kernal vectors and screen editor vectors and variables. This is the normal entry point during a RUN/STOP-RE STORE NMI interrupt.

16390 $4006 JBASIC-IR9

BASIC IRQ entry point; jumps to 43085/$A84D, the address of the routine which handles the BASIC portion of the system IRQ interrupt sequence. The target routine supports sprite movement, sprite collision detection, light pen reading, and the BASIC music statements. This is the normal entry point during the system IRQ service routine.

16393 $4009 SOFT-RESET

Performs a warm start of BASIC
This routine is the normal final step of the RUN/STOP-RESTORE sequence. It resets the SID registers and sound locations, and calls the routine at 16781/$418D to stop sprite movement. However, it does not reinitialize the BASIC vectors or pointers.

16416 $4020 Unused

Three unused bytes filled with the value 255/$FR

16419 $4023 HARD_RESET

Performs a cold start of BASIC
This routine is the normal final step of the reset sequence. It performs a complete initialization of BASIC, including resetting all vectors, pointers, and working storage locations to their default values. This routine also includes a call to the Kernal PHOENIX routine [$FF56], which will start any function ROMs that may be present, or boot a disk if one is in the drive,

16453 $4045

Initializes BASIC painters and constants
This is the main initalization routine of the cold-start sequence. It is responsible for setting all RAM working storage locations for BASIC to their default values.

16658 $4112

Initializes SID registers and sound routine locations
This routine sets all SID chip registers to 0/$00 and initializes all locations associated with the SOUND and PLAY statements.

16762 $417A

Initializes MMU preconfiguration registers

16781 $418D

Initializes sprite speed and direction table
This routine copies a 0/$00 into the speed control byte for each entry in the sprite movement table at 4478/$117E, effectively halting all sprite motion.

16795 $419B

Displays the power-on message

This routine displays the text from the following area of ROM, Note that the free memory figure is part of the ROM message, and may not reflect the actual amount of memory available to BASIC.

16827 $4IBB

Text for power-on message

{CLR}

COMMODORE BASIC V7.0 122365 BYTES FREE
(O1985 COMMODORE ELECTRONICS, LTD.
(O1977 MICROSOFT CORP.
ALL RIGHTS RESERVED

16977 $4251

Initializes BASIC indirect vectors

This routine copies the BASIC indirect vectors from the following table to locations 768-785/$0300-$0311, and initializes the vector at 764-765/$02FC-$02FD.

16999 $4267

Table of default vector values

This area contains the default addresses copied into the page 3 indirect vectors by the routine at 16977/$4251.

17017 $4279

Text for character retrieval routines

This area contains the code for CHRGET and the other page 3 character retrieval subroutines. The routines are copied into RAM.

17102 $42CE

Assorted character retrieval subroutines

17162 $430 A CRUNCH

Tokenizes keywords in lines of BASIC program text

17328 S43B0

Handles extended tokens

17356 $43CC

Deletes a character in the input buffer

17378 $43E2

Searches keyword tables for match

17431 $4417

BASIC keyword tables

The following table holds the BASIC 7.0 keywords in token number order. Bit 7 of the last character of each keyword **will** be set to %1 to indicate the end of the keyword.

Token	Keyword	Token	Keyword
128/S80	END	160/$A0	CLOSE
129/S81	FOR	161/$A1	GET
130/$82	NEXT	162/$A2	NEW
131/$83	DATA	163/$ A3	TAB(
132/$84	INPUT#	164/$A4	TO
133/$85	INPUT	165/$A5	FN
134/$86	DIM	166/$A6	SPC(
135/$87	READ	167/SA7	THEN
136/$88	LET	168/SA8	NOT
137/S89	GOTO	169/SA9	STEP
138/$8A	RUN	170/$AA	+
139/S8B	IF	171/$AB	-
140/S8C	RESTORE	172/$ AC	*
141/$8D	GOSUB	173/$AD	/
142/$8E	RETURN	174/JAE	t
143/$8F	REM	175/$AF	AND
144/$90	STOP	176/$B0	OR
145/$91	ON	177/SB1	>
146/$92	WAIT	178/$B2	—
147/$93	LOAD	179/$B3	<
148/$94	SAVE	180/$B4	SGN
149/$95	VERIFY	181/$B5	INT
150/$96	DEF	182/$B6	ABS
151/$97	POKE	183/$B7	USR
152/$98	PRINT#	184/$B8	FRE
153/$99	PRINT	185/SB9	POS
154/$9A	CONT	186/$8A	SQR
155/$9B	LIST	187/$BB	RND
156/S9C	CLR	188/SBC	LOG
157/$9D	CMD	189/$BD	EXP
158/S9E	SYS	190/$BE	COS
159/S9F	OPEN	191/$BF	SIN

Token	Keyword
192/$C0	TAN
193/$C1	ATN
194/$C2	PEEK
195/$C3	LEN
196/$C4	STR$
197/$C5	VAL
198/$C6	ASC
199/$C7	CHR$
200/$C8	LEFT$
201/$C9	RIGHT$
202/$CA	MID$
203/$CB	GO
204/$CC	RGR
205/$CD	RCLK
206/$CE	function token extender
207/$CF	JOY
208/$D0	RDOT
2O9/$D1	DEC
210/SD2	HEX$
211/$D3	ERR$
212/$D4	INSTR
213/$D5	ELSE
214/$D6	RESUME
215/$D7	TRAP
216/$D8	TRON
217/$D9	TROFF
218/$DA	SOUND
219/$DB	VOL
220/SDC	AUTO
221/$DD	PUDEF
222/SDE	GRAPHIC
223/$DF	PAINT

Token	Keyword
224/$E0	CHAR
225/$E1	BOX
226/$E2	CIRCLE
227/$E3	GSHAPE
228/$E4	SSHAPE
229/$E5	DRAW
230/$E6	LOCATE
231/$E7	COLOR
232/$E8	SCNCLR
233/$E9	SCALE
234/$EA	HELP
235/$EB	DO
236/$EC	LOOP
237/$ED	EXIT
238/$EE	DIRECTORY
239/$EF	DSAVE
240/$F0	DLOAD
241/$F1	HEADER
242/$F2	SCRATCH
243/$F3	COLLECT
244/$F4	COPY
245/$F5	RENAME
246/$F6	BACKUP
247/$F7	DELETE
248/$F8	RENUMBER
249/$F9	KEY
250/$FA	MONITOR
251/$FB	USING
252/$FC	UNTIL
253/$FD	WHILE
254/$FE	statement token extender

17929 $4609

Table of extended token statements

BASIC 7,0 has too many keywords to have a one-byte token for each. Additional statements use a two-byte token where the first byte is always 254/$FE. This table holds the extended token statement keywords in order of the second byte of the token, like the standard keywords, bit 7 of the last character of each keyword will be set to %1.

Token	Keyword	Token	Keyword
2/$02	BANK	21/$15	DCLEAR
3/$O3	FILTER	22/$16	SPRSAV
4/$04	PLAY	23/$17	COLLISION
5/$05	TEMPO	24/$18	BEGIN
6/$06	MOVSPR	25/$19	BEND
7/$07	SPRITE	26/$lA	WINDOW
8/$08	SPRCOLOR	27/S1B	BOOT
9/$09	RREG	28/$lC	WIDTH
10/SOA	ENVELOPE	29/$lD	SPRDEF
11/$OB	SLEEP	3O/$1E	QUIT
12/$0C	CATALOG	31/$1F	STASH
13/$0D	DOPEN	32/$20	(no keyword for this token)
14/$0E	APPEND	33/$21	FETCH
15/SOF	DCLOSE	34/$22	{no keyword for this token)
16/$10	BSAVE	35/$23	SWAP
17/S11	BLOAD	36/$24	OFF
18/$12	RECORD	37/$25	FAST
19/$13	CON CAT	38/$26	SLOW
20/$14	DVERIFY		

18121 $46C9

Table of extended token functions

BASIC 7.0 has too many keywords to have a one-byte token for each. Additional functions use a two-byte token where the first byte is always 206/$CE. This table holds the extended token function keywords in order of the second byte of the token. Like the standard keywords, bit 7 of the last character of each entry is set to %1.

Token	Keyword
2/$02	POT
3/$03	BUMP
4/$04	PEN
5/$05	RSPPOS
6/$06	RSPRITE
7/$07	RSPCOLOR
8/$08	XOR
9/$09	RWINDOW
10/$0A	POINTER

18172 $46FC

Table of statement dispatch addresses

This area holds the address of the routines to execute tokens 128-162/$80-$A2. Because of the way statement execution is

handled, the values here are actually one less than the address of the target routine. See Appendix F for a list of keyword execution addresses.

18242 $4742

Table of statement dispatch addresses

This area holds the address of the routines to execute tokens 213-250/$D5-$FA. Because of the way statement execution is handled, the values here are actually one less than the address of the target routine. See Appendix F for a list of keyword execution addresses.

18172 $46FC

Table of statement dispatch addresses

This area holds the address of the routines to execute extended statement tokens 2-38/$02-$26. Because of the way statement execution is handled, the values here are actually one less than the address of the target routine. See Appendix F for a list of keyword execution addresses.

18317 $478D

Table of function dispatch addresses

This area holds the address of the routines to execute tokens 180-211/$B4-$D3. See Appendix F for a list of keyword execution addresses.

18454 $4816

Table of function dispatch addresses

This area holds the address of the routines to execute extended function tokens 2-10/$02-$0A. See Appendix F for a list of keyword execution addresses.

18472 $4828

Table of operator priorities and dispatch addresses

Each mathematical operator such as +,—,*, and / has a three-byte entry in this table. The first byte is the priority of the operator for expression evaluation and the next two are the address of the routine to perform the specified operation.

18502 $4846

Prints unimplemented command message
BASIC 7.0 contains two unused keywords, QUIT and OFF. Either of those will use this routine to print the UNIMPLEMENTED COMMAND error message.

18507 $484B

Table of BASIC error messages
This area holds text for the BASIC error messages in error number order. Bit 7 in the last character of each message will be set to %1 to mark the end of the message.

Error number	Error message
1/S01	TOO MANY FILES
2/$02	FILE OPEN
3/$03	RLE NOT OPEN
4/$04	FILE NOT FOUND
5/$05	DEVICE NOT PRESENT
6/$06	NOT INPUT RLE
7/$07	NOT OUTPUT FILE
8/$08	MISSING FILE NAME
9/$09	ILLEGAL DEVICE NUMBER
10/$0A	NEXT WITHOUT FOR
11/$OB	SYNTAX
12/$0C	RETURN WITHOUT GOSUB
13/$0D	OUT OF DATA
14/$0E	ILLEGAL QUANTITY
15/$0F	OVERFLOW
16/flO	OUT OF MEMORY
17/S11	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/$19	FORMULA TOO COMPLEX
26/$lA	CANT CONTINUE
27/$lB	UNDEF'D FUNCTION
28/$lC	VERIFY
29/$lD	LOAD
30/$ IE	BREAK
31/$1F	CANT RESUME

Error number	Error message
32/$20	LOOP NOT FOUND
33/$21	LOOP WITHOUT DO
34/$22	DIRECT MODE ONLY
35/$23	NO GRAPHICS AREA
36/$24	BAD DISK
37/$25	BEND NOT FOUND
38/$26	LINE NUMBER TOO LARGE
39/$27	UNRESOLVED REFERENCE
40/$28	UNIMPLEMENTED COMMAND
41/$29	FILE READ

19074 S4A82

Sets pointer to error message

Sets locations 38-39/$26-$27 to point to the error number specified in the accmulator upon entry.

19103 $4A9F GONE

Main BASIC statement execution routine

This routine handles COLLISION processing, then falls through into the next routine to execute the current BASIC statement.

19190 $4AF6 NEWSTT

Executes the next BASIC statement

19381 $4BB5

Tests for RUN/STOP keypress

This routine tests whether the RUN/STOP key is being pressed. If so, a branch will be taken into the following routine.

19403 $4BCB STOP/END

Handles the STOP and END statements

19447 $4BF7

Handles the execution of function keywords

19587 S4C83

Displays the SYNTAX ERROR message

19590 $4C86 **OR**
Handles the OR logical operator

19593 $4C89 AND
Handles the AND logical operator

19638 $4CB6
Handles relational operators (<, =, >)

19754 $4D2A
Prints the READY prompt

19767 $4D37 READY
Enters MAIN with a READY prompt
This routine is the normal path back to immediate mode after a program or previous immediate mode line has been executed. It prints the READY prompt and falls through into the MAIN routine.

19770 $4D3A
Displays an OUT OF MEMORY error message

19772 $4D3C ERROR
Handles BASIC errors

19836 $4D7C
Prints a specified error message

19895 $4DB7 MAIN
Handles immediate mode and program line entry

19938 $4DE2
Adds or deletes BASIC program lines

20303 $4F4F LNKPRG
Relinks BASIC program lines

20371 $4F93
Reads a line of input into the buffer

20394 $4FAA
Searches for a particular token in the runtime stack

20478 $4FFE

Decrements the runtime stack pointer

20503 $5017

Checks foT available string space

This routine tests whether there is sufficient space in the string pool before a string is added. If no space is available, garbage collection is attempted.

20569 $5059

Increments runtime stack pointer

20580 $5064 FNDLIN

Searches program text for a specified line number

20640 $50AO LINGET

Creates integer value from a character string

This routine converts a string of characters at the current text pointer address into a two-byte integer value in locations 22-23/$16-$17.

20706 $50E2 LIST

Handles the LIST statement

20771 $5123

Lists a single BASIC program line

20950 $51D6 NEW

Handles the NEW statement

20984 S51F8 CLR

Handles the CLR statement

21076 $5254

Resets the CHRGET text pointer

This routine resets the CHRGET text pointer, locations 61-62/ $3D-$3E, to the beginning of the BASIC text area.

21090 $5262 RETURN

Handles the RETURN statement

21135 $528F BEND/DATA

Handles the BEND and DATA statements

21149 $529D REM
Handles the REM statement

21189 $52C5 IF
Handles the IF statement

21280 $5320
Skips a BEGIN-BEND block

21393 $5391 ELSE
Handles the ELSE statement

21411 $53A3 ON
Handles the ON statement

21446 $53C6 LET
Handles variable value assignments
This routine evaluates the expression on the right of a relational operator and assigns the resulting value to the variable on the left.

21818 $553A PRINT*
Handles the PRINT* statement

21824 $5540 CMD
Handles the CMD statement

21844 $5554 PRINT
Handles the PRINT statement

22034 $5612 GET
Handles the GET statement (also GET# and GETKEY)

22088 $5648 INPUT*
Handles the INPUT* statement

22114 $5662 INPUT
Handles the INPUT statement

22185 $56A9 READ
Handles the READ statement

22474 $57CA
Moves the CHRGET text pointer to the next DATA statement

22516 $57F4 NEXT
Handles the NEXT statement

22648 $5878 DIM
Handles the DIM statement

22661 $5885 SYS
Handles the SYS statement

22708 $58B4 TRON/TROFF
Handles the TRON and TROFF statements

22717 S58BD RREG
Handles the RREC statement

22785 $5901 MID$
Handles MID$ when used as a statement

22901 $5975 AUTO
Handles the AUTO statement

22918 $5986 HELP
Handles the HELP statement

22956 $59AC
Highlights the portion of a listed line containing an error

22991 $59CF GOSUB
Handles the GOSUB statement

23003 $59DB GOTO
Handles the GOTO statement

23069 $5A1D
Places RETURN parameters in the runtime stack

23101 $5A3D GO
Handles the GO statement

Begins by testing whether the GO token is followed by the token for TO, indicating that GOTO was entered as GO TO. The acceptance of GO TO as a synonym for GOTO is unique to Commodore.

23136 $SA60 CONT
Handles the CONT statement

23169 $5A31
Sets flags for running a program

23195 S5A9B RUN
Handles the RUN statement

23242 $5ACA RESTORE
Handles the RESTORE statement

23280 S5AF0
Table of tokens for RENUMBER

23288 S5AF8 RENUMBER
Handles the RENUMBER statement

24057 $5DF9 FOR
Handles the FOR statement

24199 $5E87 DELETE
Handles the DELETE statement

24372 $5F34 PUDEF
Handles the PUDEF statement

24397 $5F4D TRAP
Handles the TRAP statement

24418 $5F62 RESUME
Handles the RESUME statement

24544 $5FE0 DO
Handles the DO statement

24633 $6039 EXIT
Handles the EXIT statement

24714 $608A LOOP
Handles the LOOP statement

24801 $6OE1
Assigns a definition string to a programmable key

24842 $610 A **KEY**
Handles the KEY statement

24989 $619D
Table of characters for KEY

25000 $61A8 PAINT
Handles the PAINT statement

25271 $62B7 BOX
Handles the BOX statement

25643 $642B SSHAPE
Handles the SSHAPE statement

25997 $658D **GSHAPE**
Handles the GSHAPE statement

26254 $668E CIRCLE
Handles the CIRCLE statement

26448 $6750 CIRSUB
Bitmapped graphics circle-drawing subroutine

26519 $6797 DRAW
Handles the DRAW statement

26583 $67D7 CHAR
Handles the CHAR statement

26965 $6955 **LOCATE**
Handles the LOCATE statement

26976 $6960 SCALE
Handles the SCALE statement

27096 $69DS
Table of scaling factors

27106 $69E2 COLOR
Handles the COLOR statement

27212 $6A4C
Table for translating VIC color values to VDC color values

27228 S6A5C
Calculates color fill values

27257 $6A79 SCNCLR
Handles the SCNCLR statement

27482 $6B5A GRAPHIC
Handles the GRAPHIC statement

27593 $6BC9 BANK
Handles the BANK statement

27607 $6BD7 SLEEP
Handles the SLEEP statement

27693 $6C2D WAIT
Handles the WAIT statement

27727 $6C4F SPRITE
Handles the SPRITE statement

27846 $6CC6 MOVSPR
Handles the MOVSPR statement

28129 $6DE1 PLAY
Handles the PLAY statement
This routine has many suboutines to handle parsing and execution of the strings of music data. PLAY is actually a mini-language within BASIC.

28631 $6FD7 TEMPO
Handles the TEMPO statement

28644 S6FE4
Data tables for PLAY string processing

28689 $7011
Default values for ENVELOPE instrument tables

28742 $7046 FILTER
Handles the FILTER statement

28865 $7OC1 ENVELOPE
Handles the ENVELOPE statement

29028 $7164 COLLISION

Handles the COLLISION statement

This routine sets up the conditions for COLLISION checking. The actual testing for collisions occurs during the BASIC IRQ routine [$A84D].

29072 $7190 SPRCOLOR

Handles the SPRCOLOR statement

29110 $71B6 WIDTH

Handles the WIDTH statement

29125 $71C5 VOL

Handles the VOL statement

29164 $71EC SOUND

Handles the SOUND statement

29388 $72CC WINDOW

Handles the WINDOW statement

29493 $7335 BOOT

Handles the BOOT statement

If a filename is provided, the routine does the equivalent of BLOAD followed by SYS, rather than actually attempting to boot a disk.

29554 $7372 SPRDEF

Handles the SPRDEF statement

^P^V631173a.jtatement.jt,sab^-in machine language sprite-design utility program.

30444 $76EC SPRSAV

Handles the SFRSAV statement

30643 $77B3 FAST

Handles the FAST statement

30660 $77C4 SLOW

Handles the SLOW statement

30679 $77D7

Evaluates an expression with a test for type mismatch

30703 $77EF **FRMEVL**

Evaluates an expression

30935 $78D7 EVAL

Evaluates a single term of a numeric expression

31084 $796C

Displays a SYNTAX ERROR message

31096 $7978

Evaluates a variable value

This routine is also responsible for processing all the BASIC reserved variables: TI, **TI$,** ST, DS, DS$, ER, and EL.

31407 $7AAF

Finds or creates a variable

This routine searches the variable table in bank 1 for a specified variable, and creates the variable if it does not already exist.

31632 $7B90

Creates an entry in the variable table for a new scalar variable

31846 $7C66

Moves arrays upward in bank 1 to make room for a new scalar variable

31915 $7CAB

Finds or creates an array variable

32386-32767 $7E82-$7FFF

This unused area of BASIC ROM is filled with the value 255/$FE

32768 $8000 FRE

Handles the FRE function

32800 $8020

Prints designers' message

When you use the statement **SYS** 32800,123,45,6, you'll get a rather political message from the designers **of** the 128.

32842 $804A VAL
Handles the VAL function

32886 $8076 DEC
Handles the DEC function

32965 $80C5 PEEK
Handles the PEEK function

32997 $80E5 POKE
Handles the POKE statement

33014 $80F6 ERR$
Handles the ERR$ function

33090 $8142 HEX$
Handles the HEX$ function

33154 $8182 RGR
Handles the RGR function

33179 $819B RCLR
Handles the RCLR function

33283 **$8203** JOY
Handles the JOY function

33357 $824D POT
Handles the POT function

33454 $82AE PEN
Handles the PEN function

33530 $82FA POINTER
Handles the POINTER function

33566 $83 IE RSPRITE
Handles the RSPRITE function

33633 $8361 RSPCOLOR
Handles the RSPCOLOR function

33660 $837C BUMP
Handles the BUMP function

33687 $8397 RSPPOS
Handles the RSPFOS function

33761 $83E1 XOR
Handles the XOR function

33799 $8407 RWINDOW
Handles the RWINDOW function

33844 $8434 RND
Handles the RND function

33936 $8490
Table of floating-point constants for RND calculation

34000 $84D0 POS
Handles the POS function

34009 $84D9
Checks that BASIC is in run mode

34032 $84F0
Checks that BASIC is in immediate mode

34042 $84FA DEF
Handles the DEF statement

34107 S853B FN
Handles user-defined functions using FN

34222 $85AE STR$
Handles the STR$ function

34239 $85BF CHR$
Handles the CHR$ function

34262 $85D6 LEFT$
Handles the LEFT$ function

34314 $860A RIGHT$
Handles the RIGHTS function

34332 $861C MIDS
Handles the MID$ function

34408 $8668 LEN
Handles the LEN function

34423 $8677 ASC
Handles the ASC function

34437 $8685
Displays the ILLEGAL QUANTITY error message

34440 $8688
Creates space for a string in the string pool

34458 $869A
Stores a string in the string pool

34573 $870D
Performs string concatenation

34683 $877B
Evaluates a string parameter
This routine returns with locations 36-37/$24-25 set to point to the string and the Y register holding the length of the string.

34801 $87F1
Evaluates a numeric expression
This routine evaluates a numeric parameter and checks that it is in the range 0-255/$00-$FF. If the parameter is valid, it will be returned in the X register.

34819 $8803
Evaluates parameters for POKE or WAIT
This routine retrieves a pair of parameters: The first, a value in the range 0-65535/$0000-$FFFF, will be returned in locations 22-23/$16-$17, and the second, a value in the range 0-255/ $OO-$FF, will be returned in the X register.

34831 S880F
Checks that the next character is a comma

34837 $8815

Evaluates a numeric parameter

This routine retrieves a numeric parameter, checking that it is in the range 0-65535/$0000-$FFFF. If the value is valid, it will be returned in locations 22-23/$16-$17.

34862 SS82E

Subtracts value in memory from FAC1

Loads FAC2 with the five-byte floating-point value pointed to by the accumulator and Y register (low byte/high byte), then subtracts the value in FAC2 from the one in FAC1, leaving the results in FAC1.

34865 $8831

Subtracts FAC1 from FAC2

Subtracts the value in FAC2 from the one in FAC1, leaving the results in FAC1.

34885 $8845

Adds value in memory to FAC1

Loads FAC2 with the five-byte floating-point value pointed to by the accumulator and Y register (low byte/high byte), then adds the value in FAC2 to the one in FAC1, leaving the results in FAC1.

34888 $8848

Adds FAC1 to FAC2

Adds the value in FAC2 to the one in FAC1, leaving the results in FAC1.

34993 S88B1

Normalizes FAC1

35110 $8926

Forms twos complement of FAC1

35165 $895D

Displays OVERFLOW error message

35170 $8962

Performs byte alignment of FAC1

35274 S89CA LOG
Handles the LOG function

35342 $8A0E
Adds 0.5 to FAC1

35364 $8A24
Multiplies value in memory by FAC1

Loads FAC2 with the five-byte floating-point value pointed to by the accumulator and Y register (low byte/high byte), then multiplies the value in FAC2 by the one in FAC1, leaving the results in FAC1.

35367 $8A27
Multiplies value in memory by FAC1

Loads FAC2 with the five-byte floating-point value pointed to by the accumulator and Y register (low byte/high byte), then multiplies the value in FAC2 by the one in FAC1, leaving the results in FAC1.

35465 $8A89
Loads FAC2 with value from the current bank

Loads FAC2 with the five-byte floating-point value pointed to by the accumulator and Y register (low byte/high byte).

35508 $8AB4
Loads FAC2 with value from bank 1

Loads FAC2 with the five-byte floating-point value in bank 1 pointed to by the accumulator and Y register (low byte/high byte),

35607 $8B17
Multiplies FAC1 by 10

35640 $8B38
Divides FAC1 by 10

35657 $8B49
Divides value in memory by FAC1

Loads FAC2 with the five-byte floating-point value pointed to by the accumulator and Y register (low byte/high byte), then divides the value in FAC2 by the one in FAC1, leaving the results in FAC1.

35660 $8B4C

Divides FAC2 by FAC1

Divides the value in FAC2 by the one in FAC1, leaving the results in FAC1.

35796 $8BD4

Loads FAC1 from memory

Loads FAC1 with the five-byte floating-point value pointed to by the accumulator and Y register (low byte/high byte).

35840 $8C00

Copies FAC1 value into memory

Stores the value in FAC1 in five bytes pointed to by the X and Y registers (low byte/high byte).

35880 $8C28

Copies FAC2 into FAC1

35896 $8C38

Copies FAC1 into FAC2

35911 $8C47

Rounds FAC1

35927 $8C57

Determines the sign of the value in FAC1

35941 $8C65 SGN

Handles the SGN function

35972 $8C84 ABS

Handles the ABS function

35975 $8C87

Compares FAC1 against FAC2

36039 $8CC7

Converts FAC1 to a four-byte integer

36091 $8CFB INT

Handles the INT function

36120 $8D18

Fills FAC1 with the value in the accumulator

36130 $8D22

Generates floating-point value representing character string
This routine reads a character string from BASIC program text and generates the equivalent floating-point value in FAC1.

36390 $8E26

Prints IN and a line number

36398 $8E2E

Prints a line number
This routine generates a string based on the value in 59-60/$3B-$3C, then prints the results,

36418 $8E42

Generates a character string representing the value in FAC1
This routine generates a string of characters in the work area at 256/$0100 representing the value in FACL

36791 $8PB7 SQR

Handles the SQR function
This routine calculates the square root of the value in FAC1, taking advantage of the fact that SQR(X) = X t 0.5.

36801 $8FC1

Handles the exponentiation (T) operator
This routine raises the value in FAC1 to the power specified in FAC2. This routine takes advantage of the fact that At B = EXP(LOG(A) * B).

36869 $9005

Table of floating-point constants for EXP evaluation

36915 $9033 EXP

Handles the EXP function

36998 $9086

Performs series evaluation

37080 $90D8
Calls the Kernal OPEN routine

37087 $90DF
Calls the Kernal BSOUT routine

37093 S90E5
Calls the Kernal BASIN routine

37117 $90FD
Calls the Kernal CHKIN routine

37129 $9109
Calls the Kernal GETIN routine

37138 $9112 SAVE
Handles the SAVE statement

37161 $9129 VERIFY
Handles the VERIFY statement

37164 $912C LOAD
Handles the LOAD statement

37261 $918D OPEN
Handles the OPEN statement

37274 $919A CLOSE
Handles the CLOSE statement

37294 $91AE
Evaluates parameters for SAVE, LOAD, and VERIFY

37366 $91F6
Evaluates parameters for OPEN and CLOSE

37433 9243
Clears DS$ after disk operations

37457 $9251
BASIC calls to Kernal routines

The subroutines in this area are BASIC'S formal calls to Kernal routines:

37457/$9251 READSS
37463/$9257 SETLFS
37469/$925D SETNAM
37475/$9263 BASIN
37481/$9269 BSOUT
37487/$926F CLRCH
37493/$9275 CLOSE
37499/$927B CLALL
37505/$9281 PRIMM
37511/19287 SETBANK
37517/$928D PLOT
37523/$9293 STOP

37529 $9299
Creates space in the string pool for a temporary string

37610 $92EA GARBA2
Performs garbage collection on string pool

37897 $9409 COS
Handles the COS function
This routine takes advantage of the fact that COS(X) = SIN (X + Ti/2).

37904 $9410 SIN
Handles the SIN function

37977 $9459 TAN
Handles the TAN function
This routine takes advantage of the fact that TAN(X) = SIN(X) / COS(X).

38021 $9485
Table of constants for trig function evaluation

38067 $94B3 ATN
Handles the ATN function

38115 $94E3
Table of constants for trig function evaluation

38176 $9520 PRINT USING
Handles the PRINT USING statement

39361 $99C1 **INSTR**
Handles the INSTR function

39692 $9B0C RDOT
Handles the RDOT function

39728 $9B30 DRAWLN
Bitmapped graphics line-drawing routine

39931 $9BFB
Bitmapped point-plotting routine

40010 $9C4A
Scales graphics parameters

40366 $9DAE
Applies scaling factor to a specified parameter

40557 $9E6D
Evaluates graphics parameters

40712 $9F08
Handles relative graphics parameters

40783 $9F4F
Allocates the bitmapped graphics area

40903 $9FC7
Adjusts BASIC program pointers for graphics area allocation or de-allocation

40994 $A022
De-allocates the bitmapped graphics area

41076 SA074
Confirms that the graphics area has been allocated

41086 $A07E CATALOG/DIRECTORY
Handles the CATALOG and DIRECTORY statements

41245 SA11D DOPEN
Handles the DOPEN statement

41268 $A134 APPEND
Handles the APPEND statement

41303 SA157
Finds an available secondary address

41327 $A16F DCLOSE
Handles the DCLOSE statement

41347 SA183
Closes all open files for a specified device

41356 $A18C DSAVE
Handles the DSAVE statement

41380 $A1A4 DVERIFY
Handles the DVERIFY statement

41383 $A1A7 DLOAD
Handles the DLOAD statement

41416 $A1C8 BSAVE
Handles the BSAVE statement

41496 $A218 BLOAD
Handles the BLOAD statement

41575 $A267 HEADER
Handles the HEADER statement

41633 $A2A1 SCRATCH
Handles the SCRATCH statement

41687 $A2D7 RECORD
Handles the RECORD statement

41762 $A322 DCLEAR
Handles the DCLEAR statement

41775 $A32F COLLECT
Handles the COLLECT statement

41798 $A346 COPY
Handles the COPY statement

41826 $A362 CONCAT
Handles the CONCAT statement

41838 $A36E RENAME
Handles the RENAME statement

41852 $A37C BACKUP
Handles the BACKUP statement

41923 $A3C3
Evaluates parameters for disk commands

42535 SA627
Table of disk command templates

42599 SA667
Sets up disk command buffer

42872 $A778
Reads disk status string (DSS)

42977 $A7E1
Provides ARE YOU SURE query

43021 SA80D
Clears disk status string

43077 $A845
Switches to bank 15 configuration

43085 $A84D
BASIC IRQ service routine
This routine supports the MOVSPR sprite movement statement, the COLLISION statement, and the PEN function. It is also responsible for updating the duration timers for the SOUND and PLAY statements,

43504 $A9F0
Common exit point from BASIC IRQ routine

43551 $AA1F STASH
Handles the STASH statement

43556 SAA24 FETCH
Handles the FETCH statement

43561 SAA29 **SWAP**
Handles the SWAP statement

43630-44642 $AA6E-SAE62 Unused
All locations in this unused area of ROM are filled with the value 255/lFF.

44643-44799 $AE63-$AEFF
This area contains a heavily encoded message from the designers of the 128.

BASIC Jump Table

The Commodore 128's BASIC 7.0 includes a feature not found in previous versions: a jump table. Like the Kernal and screen editor jump tables, the BASIC table provides stable entry points to a number of important BASIC routines. If you want to call a BASIC routine from within one of your own machine language programs, you should use the jump table entry if one is provided. If you call a BASIC ROM routine directly, your program will not work if the address of the routine is changed in a future version. Presumably, Commodore will update the jump table if BASIC ROM is ever revised, so that jump table calls will remain valid.

In the discussions below, FAC1 refers to floating-point accumulator #1, locations 99-103/$63-$67, and FAC2 refers to floating-point accumulator #2, locations 106-110/$6A-$6E.

44800 $AF00 JAYINT
Entry point for the AYINT routine, currently at 33972/$84B4. This routine converts the contents of FAC1 into a two-byte signed integer value in locations 102-103/$66-$67 (high byte in 102/$66, low byte in 103/$67). The routine tests the original value and generates an ILLEGAL QUANTITY error message if it is not in the range -32768-32767.

44803 $AF03 JGIVAYF
Entry point for the GIVAYF routine, currently at 31036/$793C. This routine converts the two-byte signed integer

value in the Y register and accumulator (low byte in Y, high byte in the accumulator) into a floating-point value in FAC1.

44806 $AF06 JFOUT

Entry point for the FOUT routine, currently at 36418/$8E42. This routine creates a string of characters representing the floating-point value in FAC1. The string starts at location 253/$0100 and is terminated with a zero byte. The first character of the string is a space (code 32/$20) if the value was positive, or a minus sign (—) if the value was negative.

44809 $AF09 JVAL_1

Entry point for the VAL_1 routine, currently at 32850/$8052. This routine reads a string of characters from bank 1 and generates the equivalent floating-point value in FAC1. Locations 36-37/$24-$25 point to the the starting address of the string and the accumulator holds the length of the string. This routine will leave the system in BASIC'S alternate bank 14 configuration in which block 1 RAM is visible, so it shouldn't be called by a routine in bank 0.

44812 $ AFOC JGETADR

Entry point for the GETADR routine, currently at 34837/ $8815. This routine converts the current value in FAC1 into a two-byte unsigned integer in locations 22-23/$16-$17 (low byte in 22/$ 16, high byte in 23/$17). The integer value will also be in the Y register (low byte) and accumulator (high byte) upon return. Before performing the conversion, the routine checks that the value FAC1 is in the range 0-65535, and generates an ILLEGAL QUANTITY error message if it is not.

44815 $AF0F JFLOATC

Entry point for the FLOATC routine, currently at 35957/ $8C75. This routine converts the two-byte unsigned integer in locations 100-101/$64-$65 (low byte in 101/$65, high byte in 100/$64) into a floating-point value in FAC1. For this routine to function properly, you must also load the X register with the value 144/$90 and make sure the status register carry bit is set.

44818 $AF12 JFSUB

Entry point for the FSUB routine, currently at 34862/S882E. This routine subtracts the floating-point value in FAC1 from the five-byte floating-point value from the address in bank 1 specified in the accumulator (low byte) and Y register (high byte). (The bank 1 value will be loaded into FAC2.) The result will be left in FAC1.

44821 $AF15 JFSUBT

Entry point for the FSUBT routine, currently at 34865/$8831. This routine subtracts the value in FAC1 from the value in FAC2. The result will be left in FAC1.

44824 $AF18 JFADD

Entry point for the FADD routine, currently at 34885/S8845. This routine adds the floating-point value in FAC1 to the five-byte floating-point value from the address in bank 1 specified in the accumulator {low byte) and Y register (high byte). (The bank 1 value will be loaded into FAC2.) The result will be left in FAC1.

44827 $AF1B JFADDT

Entry point for the FADDT routine, currently at 34888/$8848. This routine adds the value in FAC1 to the value in FAC2. The result will be left in FAC1,

44830 $AF1E JFMULT

Entry point for the FMULT routine, currently at 35364/$8A24. This routine multiplies the floating-point value in FAC1 by the five-byte floating-point value from the address in bank 1 specified in the accumulator (low byte) and Y register (high byte), (The bank 1 value will be loaded into FAC2.) The result will be left in FAC1.

44833 $AF21 JFMULTT

Entry point for the FMULTT routine, currently at 35367/ $8A27. This routine multiplies the value in FAC1 by the value in FAC2. The result will be left in FAC1.

44836 $AF24 FDIV

Entry point for the FDIV routine, currently at 35657/$8B49. This routine divides the five-byte floating-point value from the address in bank 1 specified in the accumulator (low byte) and Y register (high byte) by the floating-point value in FAC1. (The bank 1 value will be loaded into FAC2.) The result will be left in FAC1.

44839 $AF27 JFDIVT

Entry point for the FDIVT routine, currently at 35660/$8B4C. This routine divides the value in FAC2 by the value in FAC1. The result will be left in FAC1.

44842 $AF2A JLOG

Entry point for the LOG routine, currently at 35274/$89CA. This routine calculates the natural logarithm of the value currently in FAC1, the log to the base *e*. The result will be left in FAC1.

44845 $AF2D JINT

Entry point for the INT routine, currently at 36091/$8CFB. This routine calculates the whole number portion of the current value of FAC1, removing any fractional portion. The fractional portion is simply truncated; no rounding is performed. The result is a floating-point value in FAC1, not an integer value.

44848 $AF30 JSQR

Entry point for the SQR routine, currently at 36791/S8FB7. This routine calculates the square root of the current value in FAC1. The result will be left in FAC1.

44851 $AF33 JNEGOP

Entry point for the NEGOP routine, currently at 36858/ $8FFA. This routine switches the sign of the current value in FAC1, making the value negative if it was positive, or positive if it was negative.

44854 $AF36 JFPWR

Entry point for the FPWR routine, currently at 36798/$8FBE. This routine raises the value in FAC2 to the power specified in

the five-byte floating-point value from bank 1 beginning at the address specified in the accumulator and Y register (low byte in the accumulator, high byte in the Y register). The exponent value will be loaded into FAC1. The result of the operation will be left in FAC1.

44857 $AF39 JFPWRT

Entry point for the FPWRT routine, currently at 368O1/$8FC1. This routine raises the value in FAC2 to the power specified in FAC1, effectively FAC2 t FAC1. The result of the operation will be left in FAC1.

44860 $AF3C JEXP

Entry point for the EXP routine, currently at 36915/$9033. This routine calculates the natural exponential of the value in FAC1, effectively *e* T FAC1, where *e* = 2.71828. This is the inverse of the LOG operation. The result will be left in FAC1.

44863 $AF3F JCOS

Entry point for the COS routine, currently at 37897/$9409. This routine calculates the cosine of the current value in FAC1, which will be interpreted as an angle in radians. The result will be left in FAC1.

44866 $AF42 JSIN

Entry point for the SIN routine, currently at 37904/$9410. This routine calculates the sine of the current value in FAC1, which will be interpreted as an angle in radians. The result will be left in FAC1.

44869 $AF45 JTAN

Entry point for the TAN routine, currently at 37977/$9459. This routine calculates the tangent of the current value in FAC1, which will be interpreted as an angle in radians. The result will be left in FAC1.

44872 $AF48 JATN

Entry point for the ATN routine, currently at 38067/$94B3. This routine calculates the inverse tangent (arctangent) of the current value in FAC1. The result, which can be interpreted as an angle in radians, will be left in FAC1.

4487S $AF4B JROUND

Entry point for the ROUND routine, currently at 35911/ $8C47. This routine will round the least significant bit of FAC1 according to the value in the FAC1 rounding byte, location 113/S71.

44878 $AF4E **JABS**

Entry point for the ABS routine, currently at 35972/$8C84. This routine will calculate the absolute value of the current value in FAC1, making the value positive regardless of its previous sign.

44881 SAF51 JSIGN

Entry point for the SIGN routine, currently at 35927/$8C57. This routine sets the accumulator (and processor status register) according to the current value in FAC1. If the value is zero, the accumulator will hold 0/$00 upon return (and the status register Z bit will be set). If the FAC1 value is positive, the accumulator will hold l/$01 (and the status register Z and N bits will both be clear). If the FAC1 value is negative, the accumulator will hold 255/$FF (and the status register N bit will be set).

44884 $AF54 JFCOMP

Entry point for the FCOMP routine, currently at 35975/ $8C87, This routine compares the floating-point value in FAC1 against the five-byte floating-point value from the address in bank 1 specified in the accumulator (low byte) and Y register (high byte). The accumulator (and processor status register) will be set according to the result of the comparison. If the two values are equal, the accumulator will hold 0/$00 upon return (and the status register Z bit will be set). If the FAC1 value is greater than the value in bank 1, the accumulator will hold l/$01 (and the status register Z and N bits will both be clear). If the FAC1 value is less than the value in bank 1, the accumulator will hold 255/$FF (and the status register N bit will be set).

44887 $AF57 JRND-0

Entry point for the RND_0 routine, currently at 33857/$8437. This routine generates a pseudorandom floating-point value

according to the setting of the status register N and Z bits upon entry. The resulting value will be left in FAC1. If the N bit is set upon entry, the value in FAC1 will be used as the seed, producing a predictable result. If the Z bit is set, the value in the CIA. #1 time-of-day clock is used as a seed. Otherwise, the previous random number in locations 4635-4639/ $121B-$121F is used as a seed for the next value.

44890 $AF5A JCONUPK

Entry point for the CONUPK routine, currently at 35508/ $8AB4. This routine loads FAC2 with the five-byte value at the address in bank 1 pointed to by the accumulator and Y register (low byte/high byte).

44893 $AF5D JROMUPK

Entry point for the ROMUPK routine, currently at 35465/ $8A89. This routine loads FAC2 with the five-byte value at the address in the current bank pointed to by the accumulator and Y register (low byte/high byte).

44896 $AF60 JMOVFRM

Entry point for the MOVFRM routine, currently at 31365/ $7A85. This routine loads FAC2 with the five-byte value at the address pointed to by locations 36-37/$24-$25.

44899 $AF63 JMOVFM

Entry point for the MOVFM routine, currently at 35796/ $8BD4. This routine loads FAC1 with the five-byte value at the address in the current bank pointed to by the accumulator and Y register (low byte/high byte).

44902 $AF66 JMOVMF

Entry point for the MOVMF routine, currently at 35840/ $8C00. This routine copies the contents of FAC1 into a five-byte area beginning at the address in the current bank pointed to by the X and Y registers (low byte/high byte).

44905 $AF69 JMOVFA

Entry point for the MOVFA routine, currently at 35880/ $8C28. This routine copies the contents of FAC2 into FAC1.

44908 $AF6C JMOVAF

Entry point for the MOVAF routine, currently at 35896/ $8C38. This routine copies the contents of FAC1 into FAC2.

44911 $AF6F JOPTAB

This table entry is not a jump vector. Location 44911/$AF6F does contain a JMP instruction, but the target address is not a valid routine. Instead, locations 44912-44913/$AF70-$AF71 provide a fixed reference to the address of the BASIC operator table. This table, currently at 18472/$4828, holds the priorities and dispatch addresses for the mathematical operators such as +, -, *, and /.

44914 $AF72 JDRAWLN

Entry point for the DRAWLN routine, currently at 39728/ $9B30. This is the basic bitmapped graphics line-drawing routine.

44917 $AF75 JGPLOT

Entry point for the GPLOT routine, currently at 39931/$9BFB. This routine plots a point on the bitmapped screen using the currently specified color source.

44920 $AF78 JCIRSUB

Entry point for the CIRSUB routine, currently at 26448/$6750. This is the basic bitmapped graphics circle-drawing subroutine.

44923 $AF7B JRUN

Entry point for the RUN routine, currently at 23195/$5A9B.

44926 $AF7E JRUNC

Entry point for the RUNC routine, currently at 20979/S51F3. RUNC is actually an alternate entry point into NEW to reset the text pointer to the start of program text and perform CLR.

44929 $AF81 JCLR

Entry point for the CLR routine, currently at 20984/$51F8.

44932 $AF84 JNEW

Entry point for the NEW routine, currently at 20950/S51D6.

44935 $AF87 **JLNKPRG**

Entry point for the LNKPRG routine, currently at 20303/ $4F4F. This program updates the line links for all lines in the current program.

44938 $AF8A JCRUNCH

Entry point for the CRUNCH routine, currently at 17162/ $430A. This routine is the one responsible for converting lines of text into tokenized BASIC statements.

44941 $ AF8D JFNDLN

Entry point for the FNDLN routine, currently at 20580/$5064. This routine searches through program text for the line number specified in locations 22-23/S16-17. Upon exit, the carry bit will be clear if no match was found, or set if the specified line was located.

44944 SAF90 **JNEWSTT**

Entry point for the NEWSTT routine, currently at 19190/ $4AF6. This routine prepares for the execution of the next BASIC statement.

44947 $AF93 JEVAL

Entry point for the EVAL routine, currently at 30935/$78D7. This routine evaluates a single numeric term or variable into a value in FAC1.

44950 $AF96 **JFRMEVL**

Entry point for the FRMEVL routine, currently at 30703/ $77EF. This routine evaluates a numeric expression, leaving the results in FAC1.

44953 $AF99 JRUN_A_PROGRAM

Entry point for the RUN routine, currently at 23206/$5AA6. This routine performs the portion of the RUN routine normally executed for running a program after it has been loaded from disk. The extra steps in this case include relinking the program before it is run,

44956 $AF9C JSETEXC

Entry point for the SETEXC routine, currently at 23169/ $5A81, This routine sets BASIC flags to indicate that a program is running.

44959 $AF9F JLINGET

Entry point for the LINGET routine, currently at 20640/ $50A0. This routine reads a string of characters and generates a two-byte integer number in locations 22-23/$16-$17, The value must be less than 64000 or a SYNTAX ERROR will occur.

44962 $AFA2 JGARBA2

Entry point for the GARBA2 routine, currently at 37610/ $92EA. This routine performs a garbage collection, removing inactive strings from the string pool to increase the amount of available string space.

44965 $AFA5 JEXECUTE_A_LINE

Entry point for the MAIN routine, currently at 19917/S4DCD. This routine is BASIC'S primary immediate mode loop.

44968-45055 $AFA8-$AFFF Unused

All locations in this unused area of ROM are filled with the value 255/$FF.

Machine Language Monitor ROM

The duplicate use of the word *monitor* in computer terminology may be confusing at first. The term can refer to either hardware, a dedicated video screen used to display information from the computer, or software, a program used to examine and modify the contents of memory. Once you understand the difference, the meaning of monitor is usually obvious from the context. This chapter describes the 128's built-in software monitor, which resides in the 4K block of ROM from 45056-49151/$B000-$BFFF. In addition to examining and changing memory, this monitor allows you to assemble, disassemble, and execute ML routines; examine and change microprocessor register contents; and copy, compare, save, load, and verify blocks of memory.

Like the 128's BASIC, its machine language monitor has a long heritage from previous Commodore models. All of the original CBM models (except for very early PETs) included a rudimentary monitor in ROM which allowed users to examine and modify memory and registers, execute ML programs, and load and save data, but had no provision for assembling or disassembling machine language.

The VIC-20 and Commodore 64 had no monitor in built-in ROM, but sophisticated monitors for both were available on cartridge. A number of public-domain RAM-resident monitors were also available, most notably *Superman* and *Micromon.* Finally, the Plus/4 and 16 once again included a monitor program in ROM, a version called *Tedmon.*

The 128's monitor shares many characteristics with all of its predecessors, but it includes a number of enhancements as well. One of the most notable is that it allows the entry of numbers in decimal, octal, or binary in addition to hexadecimal. Whenever the monitor expects a number, you can use a decimal value if it's prefixed with a + character, or a binary number if it's prefixed with a % character (in the rare case when you might want to use an octal—base 8—number, pre-

fix it with an ampersand, &). If no prefix is used, then hexadecimal is assumed. (Hex can also be explicitly specified by using a $ character as a prefix.)

Moving Between BASIC and the Monitor

In general, the monitor is aloof from the rest of the 128 ROM routines. Unlike BASIC—which in a number of places bypasses the Kernal jump table and calls Kemal routines directly—the monitor calls all the Kernal routines it uses through their formal jump table entries. Neither BASIC nor the Kernal calls any monitor routines other than through jump table entries. The monitor does not make use of any BASIC ROM routines or data tables and for the most part does not interfere with memory locations used by BASIC. Thus, you may pass freely back and forth between the monitor and BASIC without fear of upsetting the BASIC program currently in memory. This greatly enhances the monitor's function as a debugging tool.

One notable—and highly unfortunate—exception to this independence from BASIC is that the monitor uses addresses in the range 96-104/$60-$68 as working pointers in most operations. This area includes the addresses used by BASIC for its floating-point accumulator 1 (FAC1), where the results of mathematical operations are stored. As a result, it is impossible to use the 128's monitor to directly examine or change the contents of FAC1. This severely limits the usefulness of the monitor for experimenting with BASIC floating-point routines.

One other overlap between BASIC and the monitor is that the two share the same input buffer area for accepting and processing commands (512-673/$0200-$02Al). Thus, it is not possible to use the monitor to examine the BASIC input buffer contents or to manipulate data in the input buffer, since the buffer will be at least partially overwritten by the monitor command to display or change the memory area.

Memory Management

Another particularly attractive feature of the 128's monitor is the ease with which it interfaces with the computer's memory management system. Addresses in monitor commands are specified as five-digit hexadecimal values, where the first digit refers to the bank and the remaining four specify the address within the bank. Monitor commands that accept a range of ad-

dresses can span banks—T ED000 EDFFF 1C000, for example. Thus, the monitor can effectively see the 16 separate banks as a single 1024K (16 * 64K) block of memory. One exception is the F (fill memory) command, which cannot cross bank boundaries, because doing so would overwrite the vital contents of locations $00 and $01.

The fact that the monitor can see the 128's memory as a continuous block has an important consequence for the H (hunt for byte pattern) command. Remember that the banks are not really 16 separate blocks of memory, but rather 16 different arrangements of the available RAM and ROM. The lowest IK of memory (including the input buffer at 512/$0200) is common to all banks, and at least the lower 4K of block 0 RAM {including the buffer at 2688/$0A80, where the search pattern is stored) appears in all even-numbered banks and in banks 13/$D and 15/$F.

Thus, if you search any bank from beginning to end (for example, H 10000 1FFFF 'C-128), you'll always find at least one match for your search pattern—in the input buffer at $0200, where the search command is stored. If you search any even-numbered bank from beginning to end, you'll find at least two matches for your search pattern—once in the input buffer at $0200 and again in the search buffer at $0A80. And if you search all banks from beginning to end (for example, H 00000 FFFFF 'COMPUTE!), then you'll always find at least 26 matches because of all the times the memory areas used by the input buffer and search buffer appear in the different banks. It's important to choose your address range carefully when searching for a byte pattern.

A final note on banks and the monitor: If no bank is explicitly specified, bank 0 is assumed. This is different from BASIC, which retains the setting specified in a previous BANK statement, starting with a default of bank 15. So when using the monitor, you must always explicitly specify the bank if you wish to use any bank other than bank 0, It is particularly important to remember this when using the G and J commands, lest you send the processor off to some uncharted region of memory.

For the programmer wishing to make use of ROM routines, those in the monitor are generally less useful than those in the BASIC, screen editor, and Kernal portions of the ROM. The monitor JMPs rather than JSRs to the routines used

to perform monitor commands, so most major routines end by jumping directly back to the monitor's main loop rather than with an RTS opcode. You probably wouldn't want to incorporate calls to such routines in your own programs because the routines never return from the monitor. However, the main command execution routines use a number of subroutines that do end with RTS opcodes, and you may find some of these useful, particularly the routines to convert and print byte values as decimal numbers (at 47623/$BA07 and 47687/SBA47) or as hexadecimal numbers (at 47250/$B892). An example is provided at the end of the chapter.

Monitor Jump Table

Like the BASIC screen editor, and Kernal jump tables, each three-byte entry in the following table consists of a JMP opcode followed by the address of an important routine.

45056 $B000 JMONINIT

Monitor cold-start entry point; jumps to 45089/$B021, which enters the monitor with default microprocessor register values. This is the entry point when the RUN/STOP key is held down during power-on/reset, or when the MONITOR command is executed in BASIC.

45059 $B003 JMONBRK

Monitor break entry point; jumps to 45065/$B009, which enters the monitor with the current program counter, bank, and microprocessor A, X, and Y register values preserved. The monitor is normally called via this entry point whenever a BRK opcode is executed because the Kernal RESTOR routine [$E056], part of the RESET sequence, initializes the CBINV vector at 790-791/$0316-$0317 to point here. CBINV determines where control is passed after a BRK.

45062 $B006 JIMONRTN

Reentry point from the IMON indirect vector. Like the BASIC and Kernal indirect vectors, the monitor's command execution routine has an indirect vector, IMON (814-815/$032E-$032F), which is initialized by the Kernal RESTOR routine [$E056] to point here. From this point, control is transferred back to 45234/$B0B2 in the main loop, the address immediately fol-

lowing the IMON jump. See IMON for information on how the indirect vector can be used to wedge in additional monitor commands.

45065 $B009 MONBRK

Monitor entry routine when BRK instruction encountered.
Prints BREAK and a {BELL} character, then retrieves and stores the bank number, program counter, and microprocessor register values that were placed on the stack by the IRQ/BRK handling routine [$FF17], then branches into the following routine to fall through to the register display and main loop.

45089 $B021 MONINIT

Cold-start routine for monitor.
Switches to bank 15, loads all register storage locations with zeros, sets the program counter storage to $B000 and bank storage to 15, and prints MONITOR. Next (at $B046) the stack pointer is stored and Kernal error messages are enabled. The routine then falls through to display the stored register values and enter the main loop.

45136 $B050 SHOWREG

Handles R (register display) command.
Prints a heading for the register display, then displays the contents of the storage locations (2-9/$02-$09) that represent the program counter (prefixed with the current bank number); status; and A, X, Y, and stack pointer register values. The storage locations are filled upon entry to the monitor and can be changed with the register change (;) command. To simplify the process of changing register values, this routine adds a semicolon before the displayed values so that you can change the stored values by typing over the displayed values and pressing RETURN. The routine ends by falling through into the main loop.

45195 $B08B MONMAIN

Main command execution loop for the monitor.
Clears a line for input, then gets a command line into the input buffer (512-672/$0200-$02AO). The routine accepts characters until RETURN is pressed. Characters are then retrieved from the buffer until one is found that is not a space. This character is assumed to be a monitor command (all monitor

commands consist of a single character). With this character in the accumulator, the routine jumps through the IMON indirect vector at 814-815/$032E-$032F. Normally, this vector points to the jump table entry at $B006, which immediately returns to the address following the indirect jump. However, this vector can be changed to allow additional commands to be added to the monitor. See the discussion at IMON for more details.

The routine then compares the command character in the accumulator against characters from the command table at 45286/$B0E6. If no match is found, an error is assumed, and (at $B0BC) the error signal (a question mark) is printed following the command. The routine then loops back to process another command. If a match is found among the first 15 characters in the command table, then the command is executed by pushing an address from the table at 45308/$B0FC onto the stack, then jumping to read the parameter following the command. The RTS at the end of the parameter decoding routine [$B7A7] will cause control to be passed to the command execution address stored on the stack. If the matching character is among characters 16-19 in the table ($-%), then a jump is taken to the base conversion routine [$B9B1]. If the matching character is among characters 20-22 in the table (L-V), then a jump is taken to the routine that prepares for load, save, or verify [$B337].

45283 SB0E3 EXITMON

Handles X {exit to BASIC) command.
Leaves the monitor and returns to BASIC by jumping indirectly through BASIC'S restart vector at 2560-2561/ $0A00-$0A01.

45286 $B0E6 COMTBL

Table of monitor commands.

Each of the 22 commands consists of a single character:

A C D F G H J M R T X @ . > ; $ + & % L S V

45308 $B0FC EXECTBL

Table of execution addresses for the monitor commands.
Each two-byte entry in the table consists of the address minus 1 of the routine to perform the corresponding command. The entries are one less than the actual address because of the way the RTS opcode behaves: When RTS pulls a return address

from the stack, it adds 1 to the address value before placing the value in the 8502's program counter. The actual execution addresses for each of the commands handled by this table are as follows:

A	{Assemble instruction)	SB406
C	(Compare memory blocks)	SB231
D	{Disassemble instruction)	$B599
F	{Fill memory)	SB3DB
G	(Go to routine)	$B1D6
H	(Hunt for byte)	$B2CE
J	(Jump to subroutine)	$B1DF
M	(Memory display)	$B152
R	(Register display)	$B050
T	(Transfer memorv)	$B234
X	(eXit to BASIC)	$B0E3
@	(send disk command)	$BA90
	(same as A)	$B406
>	(change memory)	$B1AB
	(change register)	$B193

45338 SB11A MINDFET

INDFET call for the monitor.

Calls the Kernal INDFET routine [$FF74] to retrieve a character into the accumulator from the bank specified in 104/$68 at the address pointed to in 102-103/$66-$67, and with the offset specified by the contents of the Y register. The use of $66-$68 as working addresses makes it impossible to use the monitor to examine the contents of floating-point accumulator 1 (FAC1), since the value in FAC1 will be changed by the monitor M command (which uses this routine).

45354 $B12A MINDSTA

INDSTA call for the monitor-

Calls the Kernal INDSTA routine [$FF77] to store the value in the accumulator into the bank specified in 104/$68 at the address pointed to in 102-103/$66-$67, and with the offset specified in the Y register. The use of $66-$68 as working addresses makes it impossible to use the monitor to load values directly into floating-point accumulator 1 (FAC1), since the value in FAC1 will be changed by the > (memory change) command (which uses this routine).

45373 $B13D MINDCMP

INDCMP call for the monitor.
Calls the Kernal INDCMP routine [$FF7A] to compare the value in the accumulator against the value at the address pointed to in 102-103/$66-$67, with the offset specified in the Y register, from the bank specified in 104/$68. The only monitor routine that uses indirect comparison is the compare/transfer routine [$B321], and that routine calls INDCMP directly, using instead 96-97/$60-$61 as the address pointer and 98/$62 for the bank value.

45394 $B152 SHOWMEM

Handles M (memory display) command.
Displays the contents of a specified area of memory as hexadecimal values and ASCII characters. The routine functions by repeatedly calling the subroutine display lines of byte and character values [$B1E8].

The format depends on the screen mode: 8 bytes per line in 40-column mode or 16 bytes per line in 80-column mode. No parameters are required, but either one or two parameters can be specified. If no parameters are specified, the display begins at whatever address is currently in 102-103/$66-$67 from the bank specified in 104/$68. Before other operations are performed, the value in those locations is not predictable. After another M command, these locations will hold an address one line (8 or 16 bytes) higher than the previous ending address. Twelve lines of data will be displayed, representing either 96 bytes (40-column mode) or 192 bytes (80-column mode). If one address is specified, 12 lines are displayed starting at the specified address. If two addresses are specified, all bytes between the addresses are displayed; the NO SCROLL key may be used to pause the screen and the STOP key will halt the process. The routine always displays full lines, so a few bytes beyond the specified ending address may also be shown. It is possible to wrap from bank to bank; the next address after $FFFF in one bank is $0000 in the next higher bank. Thus, the M command treats the 16 banks like a continuous block of memory. However, the address will not wrap from $FFFFF to $00000. The routine ends by jumping back to the main loop

45460 SB194 CHNGREG

Handles ; (change register) command.
Allows you to change the contents of the bank, program counter, and register storage locations (2-9/$02-$09). Since the values in the storage locations are reloaded into the corresponding registers by the G and J commands, this allows you to change values that the various microprocessor registers will hold when ML routines are executed from the monitor. The register contents are usually changed by editing the values displayed by the R {register display) command [$B050]. Since that routine automatically provides the semicolon (;) in front of the values, you may not even have realized that this is a separate command. You are free to use the ; command independently of the register display, but it is somewhat less convenient.

The routine expects to read the values in order, so you must supply values for all registers with storage locations lower than the one you wish to change. This isn't a problem if you're editing the register display, but if you're using the ; (semicolon) command independently, you must supply values for all registers that are normally displayed to the left of the value you wish to change. For example, even if you want to change only the Y register value, you must still also supply address, status register, accumulator, and X register values (in order) before the Y register value.

45483 6B1AB CHNGMEM

Handles > (change memory) command.
Allows you to change the contents of one or more memory locations—to a maximum of either 8 or 16 locations, depending on whether the 40- or 80-column screen is in use. Memory contents are usually changed by editing the lines of byte values displayed by the M (memory display) command [$B152]. Since that routine automatically provides the > in front of the address and values, you may not even have realized that this is a separate command. You are free to use the > command independently of the memory display; in fact, it's more convenient when you need to change only one or two bytes.

If no parameters are found following the > command, then a line of byte values is displayed beginning at the address in 102-103/$66-$67 from the bank in 104/$68. If no

monitor commands involving ranges of addresses—for example, M, T, or C—have yet been performed, the value in these locations is unpredictable. Following the M command, the locations will hold an address one line (8 or 16 bytes, depending on the display width) higher than the address of the last line displayed.

Only hexadecimal byte value parameters can be interpreted; you cannot change memory by changing the ASCII characters displayed at the end of each line. The routine to display a line of memory [$B1E8] is called to redisplay the changed locations. A full line (8 or 16 bytes) is always redisplayed, even if you have changed only one or two bytes. The routine ends by jumping back to the main loop [$B08B].

45526 SB1D6 GOTOLOC

Handles C (go to routine) command.

Loads the bank and program counter storage locations (2-4/$02-$04) with the specified values if a target address is supplied. The stack pointer is restored to the value it had upon entry to the monitor (stored in 9/$09). This negates the effects of any stack operations the monitor routines may have performed. Finally, the Kernal JMPFAR routine [$FF71] is called to transfer control to the address specified in 3-4/$03-$04, and in the bank specified in 2/$02, with the microprocessor registers loaded from 5-8/$05-$08.

There's normally no returning from a JMPFAR. If you want to get back to the monitor after executing an ML routine using G, then the routine at the target address must end with a BRK ($00) opcode. Execution of the BRK will return you to the monitor via the break entry point [$B003],

If you use G to go to a routine that ends with an RTS, you'll be returned to BASIC at the end of the routine. If you'd prefer to be returned to the monitor when a program terminates with RTS, use J instead of G.

45535 $B1DF JMPSUB

Handles J (jump to subroutine) command.

Loads the bank and program counter storage locations (2-4/$02-$04) with the specified values if a target address is supplied. The Kernal JSRFAR routine [$FF6E] is called to transfer control to the address specified in 3-4/$03-$04, and in the bank specified in 2/$02, with the microprocessor registers

loaded from 5-8/$05-$08. Upon return from the JSRFAR, the routine jumps back to the main loop [$B08B].

To get back to the monitor cleanly after using J, you'll need to be sure that the routine at the target address ends with an RTS opcode. Execution of a BRK opcode will also cause a return to the monitor (via the break entry point [$B003]), but in that case the JSRFAR return address will be left on the stack.

45544 $B1E8 SHOWLIN

Displays a line of memory as hex bytes and ASCII characters. Clears a screen line, then prints a > character to facilitate use of the memory change command [$B1AB], Next, the bank and address of the first location in the current display line (in 102-104/$66-$68) are printed. A loop reads bytes from memory and prints two-digit hexadecimal numbers representing the byte values. The loop repeats for either 8 or 16 bytes, depending on the screen width (determined by checking the value in 215/$D7). After this, the routine prints a colon (so that the following ASCII characters will not be counted as part of the input for the memory change command) and an {RVS} character (so that the following ASCII characters will be displayed in reverse video). Finally, a second loop is used to read the same 8 or 16 bytes again, but this time to display the equivalent ASCII character for the byte value. To prevent cursor or color control characters from being printed and upsetting the screen display, the character code for the period (.) is substituted if the byte value to be displayed is less than 32/$20 or between 128-159/$80-$9F.

45617 SB231 **CMPXFR**

Compares or transfers blocks of memory.

Begins by loading an operation flag (147/$93) with a value that indicates which function is being performed: 0/$00 if the routine is entered at $B231 for C (compare) or 128/$80 if entered at $B234 for T (transfer). The transfer operation might more properly be called a copy, because the contents of the source block of memory are not changed unless the blocks overlap. A direction flag (2739/$0AB3) is also used during transfers to indicate whether bytes are being copied downward in memory (flag value 0/$00) or upward (flag value 128/$80). The direction of a transfer is significant—downward moves

must work from the starting address toward the ending address, while upward moves must work from end to start. Otherwise, if the source and destination ranges overlap, a single value would be rippled through the destination range.

The routine loads bytes from the source address range and, if a transfer operation is indicated, stores the value in the corresponding destination address using the Kernal INDSTA routine [$FF77]. Next, it compares the byte in the source range against the byte in the destination range using the Kernal INDCMP routine [$FF7A]. This performs the compare operation (the bytes should always be equal for a transfer operation). If the bytes do not match, the address of the mismatch is printed. Then the source and destination addresses are either incremented (for a compare or downward transfer) or decremented (for an upward transfer). This loop is repeated until all bytes in the range have been compared or transferred; then the routine jumps back to the main loop [$B08B], However, the loop also includes a call to the Kernal STOP routine [$FFE1], so the RUN/STOP key can be used to halt the compare or transfer.

45774 $B2CE SEARCH

Searches memory for byte pattern.

Evaluates the address parameters and calculates the number of bytes to search, then fills the buffer at $0A80 with the search pattern. If the first nonspace character following the ending address parameter is the apostrophe ('), then the following characters are copied directly from the input buffer into the search buffer, so the search will be for the actual ASCII characters. If the apostrophe is not found, the characters following the ending address are converted into byte values before being placed in the buffer.

Once the search buffer is prepared, a byte is loaded from memory and compared against the first byte in the search buffer. If the two bytes match, the next byte in memory is compared to the next byte in the buffer, and so on, until either a mismatch occurs or the end of the search buffer is reached (which indicates that the pattern has been matched). In the case of a match, the starting address of the match is printed, followed by two spaces. The testing process normally repeats until all bytes in the specified address range have been checked, but the loop includes a call to the Kemal STOP rou-

tine [$FFE1], so the RUN/STOP key can be used to be used to terminate the search. The routine ends by jumping back to the main loop [$B08B],

Be sure to see the warning in the introduction to this chapter about using the H command for searching wide address ranges.

45879 $B337 MONLSV

Prepares for load, save, or verify.

Begins by setting default values for the device number, secondary address, bank, filename length, and filename address. Setting these values directly (rather than using Kernal routines like SETLFS, SETNAM, and SETBANK) is an exception to the monitor's otherwise strict use of Kernal jump table calls. If no characters follow the command, then a branch to 45995/$B3AB allows the L and V commands to be used alone to load or verify using "nameless" tape files.

If anything follows the command, it is assumed to be a filename and must start with a quotation mark character ("), or else an error will be signaled. Characters following the opening quotation mark are copied into the monitor buffer at 2688/$0A80 until a closing quotation mark is found. An error is signaled if no closing quotation mark is found or if more than 16 characters are used in the filename. If no other parameters are found following the closing quotation mark, a branch to 45995/$B3AB attempts to load or verify a tape file with the specified name.

If a parameter value follows the closing quotation mark, the low byte of the value is copied into the device number location (186/$BA). The next parameter value is assumed to be a starting address, A load or verify is attempted if the parameter is absent. The final parameter, if any, is assumed to be the ending address. If it's missing, a branch to 46O33/$B3D1 attempts a relocating load or verify. If it is present, the command is checked, and an error is signaled if it is not S (an ending address cannot be specified for a load or verify).

45983 $B39F MONSAVE

Handles save for monitor.

Changes the secondary address setting to zero to specify a relocatable file if the device is the tape drive. The Kernal SAVE

routine [$FFD8] is used to write the data to the specified device. The routine ends by jumping back to the main loop [SB08B].

45995 $B3AB MONLOAD

Handles load and verify.

Checks that the command is either V or L, and signals an error if not. For V, the character code for V (86/$56) is left in the accumulator; for L, a zero is placed in the accumulator. The Kernal LOAD routine [$FFD5] is then called, which will perform a load if the accumulator holds zero (for L) or a verify if the accumulator holds a nonzero value {for V). The routine tests for a verify error; if one occurs, the message ERROR is printed to signal that the data in memory does not match the file on disk or tape. The routine ends by jumping back to the main loop [$B08B],

46033 6B3D1

Prepares for relocating load or verify.

Loads the specified ending address value and changes the secondary address (185/$B9) from 1 to 0, indicating a relocating load is to be attempted. The routine then branches to attempt load or verify [$B3AB],

46043 $B3DB FILLMEM

Fills memory with specified byte value.

Evaluates the starting and ending addresses, and calculates the number of bytes to fill. If the starting and ending banks are not the same, an error is signaled to prevent the fill operation from crossing bank boundaries and overwriting the important values in locations $00 and $01. If the address range is valid, the fill byte value is read, and a loop begins to store fill byte in all memory locations in the specified range. The loop normally repeats until the specified number of bytes have been filled, but it includes a call to the Kernal STOP routine [$FFE1], so the RUN/STOP key can be used to halt the fill. The routine ends by jumping back to the main loop [$B08B].

When filling, you must be careful not to overwrite page 0 or page 2, both in the area of memory common to all banks. Page 0 contains the pointer to the byte to fill (102-104/$66-$68), and page 2 contains the INDSTA routine used to store

bytes in the specified addresses. Overwriting either of these areas will likely result in a system lockup.

46086 $B406 ASSMBLE

Handles A (assemble) command or its equivalent (.). Checks for values following the A or period (.) and signals an error if none is found. The period is accepted as a synonym for A to simplify the assembly process by allowing you to edit the lines displayed by the D (disassemble) command. Since the D command automatically provides the period before each line, you may not have realized that it is treated as a separate command, but you can substitute it freely for A.

If only an address is found following the command, the routine simply returns to the main loop [$B08B], Next, the routine searches for the first group of three nonspace characters. Any values on the input line with fewer than three characters are ignored;this explains why the two-digit hexadecimal byte values displayed in front of the three-character mnemonic by the disassemble routine are ignored when the instruction is edited. It also explains why changes to the two-digit byte values are ignored by this routine. The three-character pattern is then packed into a two-byte value. This packing scheme is a holdover from the RAM-resident monitors of earlier Commodore computers. It's really unnecessary in the 128, which has room to spare in this block of memory, but Commodore's programmers probably found it easier to reuse the existing code. All 8502 ML mnemonics consist of combinations of the alphabetic characters A-Z. Since there are only 26 different valid characters, any single character can be represented by a five-bit value (which can hold 0-31), and three five-bit values can fit nicely into two eight-bit bytes.

As an example of how this packing works, suppose the pattern found is LDA—corresponding to hex bytes $4C $44 $41. First, the value 63/$3F is subtracted from each byte, yielding $0D $05 $02. The binary equivalents are %00001101 %00000101 %00000010. The rightmost five bits of each value are shifted rightward into two bytes. The resulting packed mnemonic in these locations is %01101001 %0l000100, or $69 $44.

Next, the routine infers an addressing mode from the parameter following the three-character pattern. The packed pattern is compared against those in the table at 46881/$B721.

(The packing scheme does make the testing for valid mnemonics slightly faster, since only two bytes need to be compared instead of three.) An error is signaled if no match is found; otherwise, the position of the matching mnemonic in the table is used along with the addressing mode to calculate the proper opcode for the specified instruction. The opcode and its associated parameter (if any) are then stored in memory, and the routine at 46556/$B5DC is called to disassemble the line just assembled. This provides the hex values of the ML bytes. Finally, the routine loads the input buffer with an A and the next address value so that these will be found when the routine ends by jumping back to the main loop [$B08B], This greatly simplifies the assembly of further instructions.

46489 8B599 DISASSM

Handles D (disassemble) command.

Calculates the number of bytes to disassemble, then calls 46548/$B5D4 as many times as necessary to disassemble that many bytes. If no parameter is specified, 20 bytes are disassembled beginning at the address in 102-104/$66-$68. (Actually, up to 22 bytes may be disassembled, depending on how many are necessary for the last full instruction.) If no previous commands have been executed, the address value is unpredictable. After an earlier D command, the value will be the next address beyond the last one previously disassembled. If only a starting address is provided, 20 bytes are disassembled beginning at the specified address. If both starting and ending address parameters are provided, all instructions between those addresses will be disassembled. However, the disassembly loop includes a call to the Kernal STOP routine [$FFE1], so the RUN/STOP key can be used to halt the disassembly. You may also use the NO SCROLL key to pause the disassembly. The routine ends by jumping back to the main loop [$B08B],

46548 $B5D4 DISASM1

Disassembles a single instruction.

Prints a period (to simplify editing of instructions), then the bank and address of the opcode byte to be disassembled. A call to 46681/SB659 calculates the addressing mode and offset into the packed mnemonic table for this opcode. The hex value of the opcode and up to two associated data bytes are then printed. They're padded with spaces to align the mne-

monies column. A call to 46753/SB6A1 unpacks and prints the mnemonic. The associated parameter value is then printed, along with such characters as #, (, and), to identify the addressing mode for the instruction. For relative branching instructions, the target address of the branch is printed instead of the relative offset value.

46681 $B659 CALCMN

Calculates mnemonic and addressing mode.

Manipulates the specified opcode (in the accumulator upon entry) to provide the offset into the table for the corresponding packed mnemonic, an addressing mode identifier value, and a count of associated data bytes (0-2).

46753 $B6A1 PRNTMN

Prints mnemonic for opcode.

Unpacks and prints a mnemonic from the table at 46881/$B721. Upon entry, the accumulator holds the offset into the table for the mnemonic to be printed. As an example of how the unpacking works, the first table entry is $1C $D8. The binary equivalent is %0001110011011000. Divided into three five-bit groups (and ignoring the rightmost bit), that's %00011 %10011 %01100, or $03 $13 $0C. Adding $3F to each yields $42 $52 $4B, corresponding to the character codes for the letters BRK. You would expect this to be the first table entry, since the BRK instruction has the lowest possible opcode ($00).

46787 $B6C3 OPCDTBL

Opcode decoding table.

The values in this table are used by the mnemonic and mode calculation routine [$B659] to determine the packed mnemonic table offset for the specified opcode value.

46855 6B707 INDCTBL

Table of addressing mode indicators.

Each 8502 mnemonic may have several possible addressing modes, each with a different opcode. The values from this table are used to indicate the mode which should be associated with a mnemonic to represent the current opcode.

46869 $B715 MODETBL

Table of mode identification characters.

The characters in this table are used by the assemble routine [$B406] to determine the addressing mode being used, and by the disassembly routine [$B5D4] to disassemble an instruction to add the proper characters to indicate the mode being used.

46881 $B721 MNEMTBL

Table of mnemonics in packed form.

Each three-character mnemonic is packed into two bytes (see the assemble routine [$B406] for details). The table is in two halves: entries at 46881-46944/$B721-$B760 are the first byte for the corresponding packed mnemonic, and entries at 46945-47102/$B761-$B7A0 are the second byte.

47013 $B7A5 GETPARM

Evaluates a parameter in the input buffer.

Interprets the next numeric parameter in the input buffer (using the parameter conversion routine [$B7CE]), then sets the status register accordingly: carry clear if a parameter has been found and the following character is a space, comma, or the end of the input line, or carry set if no parameter has been found. Upon exit, the accumulator holds the number of digits in the parameter.

47054 $B7CE CVTPARAM

Transforms numeric parameter into byte value.

Reads a parameter value from the input buffer and converts it into a three-byte value in 96-98/$60-$63 (in low-byte to high-byte order). The parameter can be in any one of four different numeric bases: hexadecimal, decimal, octal, or binary. Hex is assumed unless another base is specifically indicated by a prefix character: $ for hexadecimal, + for decimal, & for octal, and % for binary. Upon exit, location 2740/$0AB4 will hold a count of digits in the parameter (zero if no parameter was found). The status register's carry bit will be set if the parameter value is too large to be interpreted (greater than 1048575/$FFFFF), and clear otherwise.

47242 $B88A BASETBL

Table of bases and bits-per-digit.

This table is used by the parameter transformation routine [$B7CE] to determine how many bits to read for each digit of a number in the corresponding base.

47250 $B892 PRNTHEX

Prints a hexadecimal value.

Prints a five-digit hexadecimal value followed by a space. The first digit comes from the low nybble of 104/168, the next two from 103/$67, and the final two from 102/S66. When entered at 47263/SB89F, the routine prints a four-digit hexadecimal value representing the value in the accumulator (low byte) and X register (high byte). When entered at 47269/$B8A5, a two-digit hex value representing the value in the accumulator is printed. When entered at 47272/$B8A8, only a space is printed.

47277 $B8AD

Moves cursor to start of current line.

Uses the Kernal PRIMM routine [$FF7D] to print a carriage return, CHR$(13), followed by a cursor up, CHR$(145).

47284 8B8B4

Moves cursor to start of next line.

Uses the Kernal BSOUT routine [$FFD2] to print a carriage return, CHR$(13).

47289 $B8B9 CLRLIN

Clears a screen line.

Uses the Kernal PRIMM [$FF7D] routine to print a carriage return, CHR$(13), followed by ESC Q (to clear a screen line) and a space.

47298 $B8C2 PRNTBYT

Prints two ASCII characters for a byte value.

Calls the byte conversion routine [$B8D2] to generate two ASCII characters representing the hex digits for the byte in the accumulator, then uses the Kernal BSOUT routine [$FFD2] to print these characters.

47314 8B8D2 CVTBYT

Converts a byte value into two ASCII characters.

Generates two ASCII characters representing the hexadecimal digits for the byte value in the accumulator upon entry. The characters are returned in the accumulator (high nybble) and X register (low nybble).

47335 $B8E7 TESTCHR

Tests next character in the input buffer.

Tests the character in the input buffer at the previous pointer position (or, if entered at 47337/$B8E9, at the current pointer position). If the character is a colon, question mark, or the zero byte marking the end of input, the carry bit will be set upon exit to indicate that the end of the input line has been reached. Otherwise, carry will be clear to indicate that characters remain to be read.

47361 $B901 MOVEVAL

Transfers address and bank values to working pointer.

Loads the working address pointer (102-103/$66-$67) with the parameter value calculated by the routine to transform input characters into byte values [$B7CE] in 96-97/$60-$61. The calculated bank value in 98/$62 is loaded into the bank pointer at 104/$68,

47374 $B90E CALCCNT

Calculates number of bytes and banks to display or move.

Calculates the number of bytes and banks in an address range by subtracting the starting address value in 102-103/$66-$67 from the ending address value in 96-97/$60-$61. The result of the subtraction is left in 96-97/$60-$61. The number of whole banks in the range is found by subtracting the starting bank, in 104/$68, from the ending bank, in 98/$62, with the result left in 98/$62.

47394 8B922

Decrements pointer/counter

Decrements the contents of 96-98/$60-$62 by 1 if entered at 47394/$B922, or by the value in the accumulator if entered at 47396/SB924. This is the line count for the display routine, the byte count for the disassemble routine, or the target address pointer for compare/transfer routine.

47420 $B93C DECCNT

Decrements byte count.

Decrements the count of bytes for the current operation (99-100/$63-$64). If the value there rolls over from $0000 to $KFFF, the count of banks (101/$65) is also decremented. Upon exit, carry bit in the status register will be clear if the countdown is complete; otherwise, the carry bit will be set.

47440 $B950 **INCPTR**

Increments address pointer.

Increments the address pointer (102-103/$66-$67) by 1 if entered at 47440/$B950, or by the value in the accumulator if entered at 47442/$B952. If the incrementing causes the address value to roll over from $FFFF to $0000, the bank pointer (104/$68) is also incremented.

47456 8B960 DECPTR

Decrements address pointer.

Decrements the address pointer (102-103/$66-$67). If the decrementing causes the address value to roll over from $0000 to $FFFF, the bank pointer (104/$68) is also decremented.

47476 SB974 **CHNGADD**

Changes bank and address.

If the parameter evaluation routine [$B7A7] finds an address parameter for the command (indicated by a clear carry status bit), the address is loaded into the program counter storage locations, 3-4/$03-$04, and the bank value is loaded into the bank storage location, 2/$02. If no address parameter is provided, the values in the storage locations remain unchanged.

47491 $B983 PREPPTR

Prepares pointers for dual-address operations.

Sets up the required pointers for those commands that require both starting and ending address parameters (C, F, H, and T). The starting address is loaded into 102-103/$66-$67, with the starting bank in 104/$68. The ending address is loaded into 2743-2744/$0AB7-$0AB8, with the ending bank in 2745/ $0AB9. The number of bytes and banks affected, the difference between the starting and ending addresses, is calculated and stored in 99-101/$63-$65. If either address is missing or

if the ending address is less than the starting address, the carry status bit will be set upon exit to signal the error.

47537 $B9B1 NUMXVRT

Performs number base conversion.

Evaluates the parameter following the base symbol, then prints the value of the number in four different number bases: hexadecimal, decimal, octal, and binary. The routine can convert values in the range 0-1,048,575. For the hex value, a dollar sign is printed followed by four hex digits (five if the value is greater than 65535/$FFFF). For decimal, octal, and binary, this routine provides the leading character (+ for decimal, & for octal, and % for binary), then uses the subroutine at 47687/$BA47 to display the values. The routine ends by jumping back to the main loop "

47623 SBA07 HEXDEC

Converts a hexadecimal value to decimal.

Converts the three-byte hexadecimal value in 96-98/$60-$62 into a decimal value in BCD (binary coded decimal) format in 2720-2723/$0AA0-$0AA3. For example, converting the value 258/$102, stored in 96-98/$60-$62 as $02 $01 $00, will result in $00 $00 $02 $58 being left in 2720-2723/ $0AA0-$0AA3. This routine takes advantage of the 8502's rarely used decimal mode. To prevent the complications that would likely be caused by interrupts occurring while the processor is set for decimal mode, interrupts are disabled until the 8502 is reset to its normal (binary) mode.

47687 $BA47 PNTOBD

Prints octal, binary, or decimal values.

Prints the octal or binary equivalent of a three-byte value. Enter with 96-98/$60-$62 holding the three-byte value, the accumulator holding the maximum number of digits allowed (8/$08 for octal, 24/$18 for binary), and the Y register holding the number of bits to interpret per printed digit minus 1 (2/$02 for octal, 0/$00 for binary). To print a decimal value, enter at 47709/$BA5D with the decimal value in 2720-2723/ $OAA0-$OAA3 in BCD format (see the routine 47623/$BA07),

and with 8/$08 (maximum number of digits) in the accumulator and 3/$03 (number of bits-per-digit minus 1) in the Y register.

47760 8BA90 DISKCMD

Handles @ (disk) commands.
Checks whether a parameter was found following the command. If so, the low byte of the value specifies the device number for the disk command. If no parameter is specified, a device number of 8—the most common number for disk drives—is used. If the first nonspace character following the device number is a dollar sign, then the routine branches to 47875/$BB03 to display a disk directory. Any other characters following the device number parameter are sent over the serial device as a command to the specified device. The routine ends by jumping back to the main loop [$B08B],

Because the first item following the @ is always interpreted as a device number, you must specify some device number value (@8 10 is a valid command, but @I0 isn't because the 10 will be interpreted as an invalid device number). However, an alternate syntax is possible. The parameter evaluation routine [$B7A7] exits when it encounters a comma, so a comma can be used alone to specify a null parameter. Thus, a command of the form @,I0 is acceptable—the default device number (8) will be used.

47875 $BB03 SHOWDIR

Displays disk directory.
If the first nonspace character in the command string following the @ command is a dollar sign, this routine is branched to because a disk directory requires more elaborate screen formatting than the simple disk status messages otherwise returned by the @ command. This routine is a good model for a directory display using machine language. (You can't use this routine directly because it ends with a branch that returns to the monitor main loop [$B08B].)

47986-49151 $BB7 2 -$BFFF Unused

The final 1166 bytes in the monitor ROM block are unused and contain 255/SFF (except the last two bytes, which are $00 $3A).

Programming Example

The following program, which prints a countdown in both decimal and hexadecimal, illustrates how several of the monitor routines can be incorporated into your own ML programs. Use the command J $F0B00 to execute the routine:

```
0B00  LDA  #$10     ;Load $60-$62 with the
0B02  STA  $60      ; initial count
0B04  LDA  #$27
0B06  STA  $61      ; $002710 (10,000 decimal)
0808  LDA  #$00     ; in this example
0B0A  STA  $62
0B0C  JSR  $B8B4    ;Clear a line for the display
OBOF  JSR  $B8AD    ;Move print position to clear line
0B12  JSR  $BA07    ;Convert byte value to decimal
0B15  LDA  #$00     ;Set up for decimal printing
0B17  LDX  #$08
0B19  LDY  #$03
0B1B  JSR  $BA5D    ;Print decimal value
0B1E  JSR  $B8A8    ;Print a space
0B21  JSR  $B901    ;Transfer value to working storage
0B24  JSR  $B892    ;Print hexadecimal value
0B27  JSR  $B922    ;Decrement count value
0B2A  BCS  $0B0F    ;Loop until value reaches zero
0B2C  RTS
```

Screen Editor ROM

If the 128 is your first computer, or if you have upgraded to the 128 from an earlier Commodore model, then you may not appreciate the power and versatility of the 128's full-screen editor. The editor is so well integrated into the system that you may take it for granted. However, if you've ever worked with an Apple, Radio Shack, TI-99, or other computer with limited editing features, you'll appreciate the ability to move the cursor to any position on the screen, insert and delete characters, and reenter entire lines just by pressing RETURN anywhere on the line. (Imagine the frustration of having to re-type an entire line of input just because a single character was mistyped.) And, with the addition of ESC key sequences, the 128's screen editor is even more powerful and versatile than the earlier Commodore editors from which it is descended.

In earlier Commodore models, the screen editing routines were an integral part of the Kernal ROM, but in the 128 the routines have been separated and moved to their own block of ROM, occupying locations 49152-53247/$C000-$CFFF. The routines in this block handle keyboard input and all aspects of text screen output for both the 40- and 80-column displays. Special features such as bitmapped graphics and sprites are handled in BASIC ROM, but editor routines set up the bit-mapped and multicolor bitmapped screens.

The 40- and 80-column text displays constitute the main output device for the 128, and thereby its primary mechanism for communicating with the user. For the fundamentals of how each display is generated, refer to the description of the respective 40- and 80-column video chips in Chapter 8. Note that both video sources remain on at all times, so it's possible to connect both a composite and an RGBI or monochrome monitor and have simultaneous 40- and 80-column displays. At any given time, however, one of the screens will be consid-ered active and the other inactive, That is, the cursor will "live" on only one of the displays, and all printing will be di-rected to that screen. The active screen is considered device 3, the default output device for the system.

Windows

One of the most significant new features of the 128's screen output is that it's window-oriented. That is, printing and cursor movement are confined to the boundaries of a window—a defined area of the screen—rather than to the absolute height and width of the screen. The window can be, and most often is, the full size of the screen, but it can be as small as a single character. Once you change the window boundaries, you must think in terms of windows rather than screens. For example, you should think of the SHIFT-CLR/HOME key as "clear window" rather than "clear screen." Only the currently defined window area will be cleared when that key combination is pressed. You can also limit printing to certain areas of the screen by modifying the window margins. The 128's windowing capabilities are no match for those of more advanced computers like the Macintosh and Amiga, but they do offer exceptional control over display formatting.

In order to understand how characters in the window are manipulated, you must first understand the distinction between physical lines and logical lines. A physical line is just one horizontal row of the window—the characters between the left and right margins. However, when you print characters on the screen and the printing overflows from one row into the next, the editor will consider the physical lines to be linked together into a single logical line. Lines will continue to be linked until either the RETURN or SHIFT-RETURN characters are encountered. A logical line can be as short as a single physical line or as long as the entire window, depending on how many characters are printed before the RETURN or SHIFT-RETURN. (By contrast, the Commodore 64 allows logical lines to span a maximum of two physical lines.) Some screen editor routines operate on physical lines, others on logical lines. The entries in this chapter indicate when an operation affects an entire logical line.

Line Links

The system for linking lines on the 128 is considerably different from that of earlier Commodore models. For example, the Commodore 64 maintains a table at 217-242/$D9-$F2 which contains the high bytes of the first address on each screen line, with the high bit of each address byte used to indicate whether the line is linked. The 128 has no such table; instead, it con-

tains a four-byte line link bitmap at 862-865/ $035E-$0361. Each screen line has a corresponding bit in the map. A bit set to %1 indicates that the corresponding line is linked to the one above. See the entries for location 862/$035E and for the routines at 52084-52144/$CB74-$CBB0 for more information.

The 128 also implements a tab feature using a similar bit-mapping scheme. The ten bytes at 852-861/$0354-$035D provide a bit corresponding to each screen column, A bit in the map set to %1 corresponds to a tab stop. See the entry for location 852/$0354 and for the routines at 51535-51597/ $C94F-$C98D for more information.

The other major function of the screen editor is to handle input from the keyboard. Although you tend to think of the keyboard as an integral part of the computer, the system sees it as just another peripheral, device 0. The keyboard is the default input device for the 128. It is normally scanned during the jiffy IRQ sequence. See the entry for the SCNKEY routine [$C55D] for details. The system's two major Kernal input routines, GETIN [$EEEB] and BASIN [$EF06], both call screen editor routines to retrieve input from the keyboard.

An enhanced feature of the 128's keyboard is programmable keys. Definition strings can be assigned to the eight function keys, F1-F8, and to the SHIFT-RUN/STOP combination and the HELP key. The definitions can be of variable lengths, with the restriction that the combined lengths of all ten definitions cannot exceed 246 characters.

The screen editor has another feature that makes redefining the keyboard much easier than on previous Commodore models. The tables for decoding keyscan codes into character codes are still in ROM, but the pointers to the tables are now maintained in RAM. Thus, redefining the keyboard is as simple as redirecting the pointer to a new decoding table set up in RAM. See the entry for locations 830-841/$033E-$0349 for details.

Screen Editor Jump Table

Most major screen editor routines can, and should, be entered through their respective entries in the following jump table. Each three-byte table entry consists of a JMP opcode followed by the address of the target routine.

49152 $C000 JCINT

Entry point for the routine at 49275/$C07B, which establishes the default characteristics for both the 40- and 80-column displays. This entry is part of both the reset and RUN/STOP-RESTORE sequences. No preliminary setup is required, but the behavior of the routine is affected by the setting of the initialization status flag (2564/$0A04). The keyboard decoding table pointers and screen editor indirect vectors are initialized only when bit 6 of the flag is %0. If you wish to preserve decoding table pointers or indirect vectors while resetting other screen editor characteristics, set this bit to %1 before you call the routine.

49155 $C003 JDISPLY

Entry point for the routine at 52276/SCC34, which deposits a screen code and attribute value at the current cursor position. Call this entry with the accumulator holding the screen code (not the character code) for the desired character and the X register holding the attribute value for the character.

49158 SC006 JKEYIN

Entry point for the routine at 49716/SC234, which retrieves a single character from the keyboard. (In official Commodore literature, this routine has the rather nondescriptive name LP2.) This entry is used by the Kernal GETIN routine [SEEEB] when the keyboard is the input device. (The keyboard, device 0, is the 128's default input device.) Upon return, the accumulator will contain the retrieved character. Be sure to see the warning in the KEYIN entry about calling this routine directly.

49161 C009 JGETSCRN

Entry point for the routine at 49819/$C29B, which retrieves a character from a line of keyboard or screen input. (In official Commodore literature, this routine has the rather nondescriptive name LOOP5.) This entry is used by the Kernal BASIN routine [$EF06] when input is requested from the keyboard (device 0) or the screen (device 3). The input source is determined by the value in the flag at 214/SD6. When the flag has a nonzero value, the character at the current cursor position is read and returned in the accumulator. When the end of the line of input is reached, the routine returns the code for the

RETURN character, the value 13/$0D. If the flag is zero, the routine accepts characters from the keyboard and displays them on the screen until RETURN is pressed. It then returns the first character of the input in the accumulator and sets the source flag to a nonzero value so that subsequent calls retrieve characters from the input line displayed on the screen. In either case, the source flag should be set only before the first call to this routine for any given line of input.

For keyboard input, the starting and ending columns and the row for the input line are set automatically. For screen input, the character is read from the row specified in 235/$EB at the column specified in 236/$EC. These locations normally hold the current cursor position and are advanced one position to the right after each call to this routine. However, the locations can also be reset to any row and column you desire. The screen input line ends on the row specified in 2608/$0A30 at the column specified in 234/$EA. Neither of these locations is set automatically, so you must specify the ending position before calling for input from the screen. The X and Y register contents will be preserved during this routine.

49164 $C00C JPRINT

Entry point for the routine at 50989/$C72D, which prints a character at the current cursor position using the current attribute (in 241/$F1). This routine is used by the Kernal BSOUT routine [$EF79] when the screen is specified as the output device. (The screen, device 3, is the 128's default output device.) Call the entry with the accumulator containing the character code (not the screen code) for the character to be displayed. The accumulator and X and Y register contents are preserved during this routine.

49167 $C00F JSCRORG

Entry point for the routine at 52315/$CC5B, which returns information about the current window size. This entry is the target of the Kernal jump table entry JSCRORG [$FFED], Upon return, the X register will hold the number of columns (minus 1) in the current window, and the Y register will hold the number of rows (minus 1). The accumulator will hold the maximum column number for the active screen (39 for the 40-column display or 79 for the 80-column display).

49170 $C012 JSCNKEY

Entry point for the routine at 50525/$C55D, which scans the keyboard matrix for a keypress. This entry is the target of the Kernal jump table entry JSCNKEY [$FF9F]. If a key is found to be pressed, its corresponding character code will be placed in the keyboard buffer for retrieval by the GETIN or BASIN routines, unless the key is a shift key or programmable key. Both of those get special handling (see the SCNKEY routine for details).

49173 $C015 JREPEAT

Entry point for the routine at 50769/$C651, actually an alternative entry into the SCNKEY routine. At this point, the routine expects location 212/$D4 to contain the keyscan matrix code for the current key, 211/SD3 to hold the status of the shift keys, and 204-205/$CC-$CD to point to the keyboard decoding table selected according to the shift key status. This table entry is most useful when intercepting the KEYVEC indirect vector (see the entry at 82 6/$ 033A for details).

49176 $C018 JPLOT

Entry point for the routine at 52330/$CC6A, which reads or sets the cursor position. This entry is the target of the Kernal jump table entry JPLOT [$FFF0], If called with the carry bit clear, the cursor is moved to the row specified in the X register and the column specified in the Y register (the row and column settings are relative to the current home position of the window). The carry bit will be set upon return if the specified position is outside the current window boundaries. If called with the carry set, the routine returns with the row number of the current cursor position in the X register and the column number in the Y register (again, the numbers will be relative to the current home position of the window).

49179 $C01B JCRSR80

Entry point for the routine at 52567/$CD57, which moves the cursor on the 80-column display to the row and column specified in 235/$EB and 236/$EC, respectively. The 40-column display's cursor is maintained by software, so it follows these pointers automatically, but the position of the 80-column cursor must be set explicitly.

49182 $C01E JESCAPE

Direct entry point into the escape sequence handling routine [$C9BE], Normal entry into the handler begins with an indirect jump through the ESCVEC vector (824-825/$0338-$0339). That vector normally points to this table entry, which in turn jumps back into the handling routine at the point immediately following the indirect jump.

49185 $C021 JKEYSET

Entry point for the routine at 52386/$CCA2, which redefines a programmable key. This entry is the target of the Kernal jump table entry JPFKEY [$FF65]. Call this entry with the X register containing the key number (1-10), the Y register containing the length of the definition string, and the accumulator holding the address of a two-byte zero-page pointer to the string. The zero-page byte immediately following the pointer should contain the number of the memory bank in which the string resides.

49188 $C024 JSCNIR9

Entry point for the routine at 49556/SC194, which handles the screen editor portion of the IRQ handling sequence. This includes setting up the screen display mode, scanning the keyboard, and managing the cursor for 40-column display.

49191 $C027 JIN1T80

Entry point for the routine at 52748/SCE0C, which initializes the character patterns for the 80-column display. This entry is the target of the Kernal jump table entry JINIT80 [$FF62]. The routine copies the contents of the 40-column display's character ROM ($D000-$DFFF in bank 14) into the character definition area of the 8563 chip's private block of RAM.

49194 $C02A JSWAPPER

Entry point for the routine at 52526/$CD2E, which switches active screen displays. This entry is the target of the Kernal jump table entry JSWAPPER [$FF5Fj. The routine exchanges the screen variable tables, tab stop tables, and line link tables, and toggles the active screen flag (215/$D7). This will redirect printing to whichever screen was previously inactive.

49197 SC02D JWINDOW

Entry point for the routine at 51739/$CA1B, which sets the position of a corner of the output window. To set the top row and left column of the window, call this entry with the carry bit clear, the accumulator holding the row number, and the X register holding the column number. To set the bottom row and right column of the window, call this entry with the carry bit set, the accumulator holding the row number, and the X register holding the column number.

49200 SC030 Unused

Three unused bytes filled with the value 255/$FR

49203 8C033 SADDRTBL

Table of screen line starting addresses.

Locations 49203-49227/$C033-$C04B hold the low bytes of the address in 40-column screen memory for the column 0 position in each of the 25 screen lines. Locations 49228-49252/$C04C-$C064 hold the high bytes for the line addresses for the default 40-column screen position (at 1024/S0400). These table values are used in conjunction with the screen base addresses, 2619/$0A3B for the 40-column screen or $2606/$0A2E for the 80-column screen, to calculate the actual starting address for each line of screen and attribute memory.

49253 $C065 SCNVCTRS

Table of **default screen editor indirect vectors.**

The five two-byte values here are copied to the screen editor indirect vector table at 820-829/$0334-$033D when the system is reset. In each case, the default indirect vector merely returns control to the location immediately following the indirect jump,

49263 8C06F KEYPTRS

Table of default keyboard decoding table pointers.

The six two-byte values here, the addresses of the ROM keyboard decoding tables, are copied to the keyboard decode pointer table at 830-841/$033E-$0349 when the system is reset

49275 $C07B CINT

Initializes screen editor constants, variables, tables, and vectors. (This routine has a screen editor jump table entry at 49152/$CO00 and a Kernal jump table entty at 65409/SFF81.)

Sets the default values for all working storage memory locations used by screen editor routines. This routine is part of both the reset and RUN/STOP-RESTORE sequences. The CIA chip register (56576/$DD00) that controls VIC-II chip memory banking is set to have the VIC-II see memory from its bank 0, corresponding to addresses 0-16383/$0000-$3FFF in block 0 of system RAM (VIC-II banks are different from MMU banks; see Chapter 8 for more details). Bit 1 of the processor I/O register (l/$01) is set to %1 to make block 1 of color memory (the block for 40-column text screen color) visible at 55296/$D800. The Kernal CLRCH routine [$FFCC] is called to reset normal I/O sources: input from the keyboard and output to the screen. The SID chip volume register (54296/$D418) is set to zero to silence any sound output (a step not present in the Commodore 64's CINT routine). Screen editor RAM variables and constants are initialized as follows:

Location	Value	Meaning
208/$D0	0	Keyboard buffer empty
209/$D1	0	No function key pending
214/$D6	0	Input from the keyboard
216/$D8	0	40-column display set for text mode
217/SD9	0	CHAREN shadow cleared
2592/$0A20	10/SOA	Maximum of 10 keys in buffer
2593/$0A21	0	Printing pause flag cleared
2594/$0A22	128/$80	All keys repeat if held down
2595/$0A23	4/$04	Delay between repeats initialized
2596/$0A24	10/$0A	Delay before repeats initialized
2598/$0A26	0	40-column cursor will blink
2599/$0A27	10/$0A	Cursor blink switch initialized
2600/$OA28	10/$0A	Cursor blink countdown initialized
2603/$0A2B	96/$60	80-column cursor set for blinking block
2604/$0A2C	20/$14	40-column text screen and character base initialized (screen at 1024/$0400, character set seen at 4096/$1000)
2605/$0A2D	120/$78	40-column bitmap and matrix base initialized {matrix at 7168/$1COO, bitmap at 8192/$2000)
2606/$OA2E	0	Base page for screen memory in 8563 RAM (screen at 0/$0000)

2607/$0A2F	8/$08	Base page for attribute memory in 8563 RAM (attributes begin at 2048/$0800)
2612/$0A34	208/$DO	Default raster line for split screen (text portion starts at line 20)
2619/$0A3B	4/$04	Base page for 40-column screen memory (value read from 49228/$C04C)

Also during this setup, the active screen flag (215/$D7) is set for 40-column mode, and bit 7 of the MMU mode configuration register (54533/$D505) is set to %1 to enable reading of the 40/80 DISPLAY key. While the 40-column display is active, default tab stops are set and default screen editor variables are copied from the table at 52852/$CE74 into 224-249/$E0-$F9. At the same time, default variables for the 80-column display are copied from the table at 52878/$CE8E into 2624-2649/$0A40-$0A59. However, there is a bug in this portion of the routine: The copy loop uses an index of 26/$1A instead of the proper value 25/$19, so the byte following the 40-column table is also copied to 250/$FA, and the byte following the 80-column table is also copied to 2650/$0A5A. The screen editor indirect vectors (820-829/$0334-$033D) are initialized from the table at 49253/$C065.

Next, bit 6 of the initialization status flag (2564/$0A04) is checked. If the bit is set to %1, the following steps to load keyboard table pointers and key definition strings are skipped. If the bit is clear, the keyboard decoding table pointers (830-841/$033E-$0349) are initialized from the table at 4926/$C06F, and the default programmable key definitions are copied to the definition area at 4096/$1000 from the table at 52904/$CEA8. After the values are copied, bit 6 of the flag is set to %1 to indicate that these steps have been performed. The flag is cleared before this routine is called by the reset routine [$EOO0], so these steps will always be performed during a reset. However, the RUN/STOP-RESTORE sequence [$FA53] does not affect the flag, so any changes to programmable key definitions or keyboard decoding table pointers will remain intact after RUN/STOP-RESTORE.

Default tab stops are then established for the 80-column screen, and the 80-column window is reset to full screen size and cleared. The 40-column window is also reset to full screen size and cleared. Finally, the position of the 40/80 DISPLAY key is read (by checking bit 7 of 54533/$D505), and the selected display will become the active screen upon exit.

49474 SC142 CLEAR

Clears the current window and homes the cursor. Also handles clear screen character, CHR$(147).

Sets the cursor to the home position and clears rows until the bottom of the window is reached, then falls through into the following routines to return the cursor to the home position and set pointers.

49488 $C15O HOME

Moves the cursor to the home position of the current window. Sets the cursor row (235/$EB) and starting row for input (232/$E8) to the top row of the current window (229/$E5), and the cursor column (236/SEC) and starting column for input (233/$E9) to the left margin of the current window (230/$E6), then falls through into the next routine to set pointers to that line.

49500 $C15C SETLINE

Sets starting address pointers for the current line. Loads the line number where the cursor currently resides (235/$EB) into the X register, then falls through into the next routine to set pointers to that line.

49502 $C15E SETADDR

Sets starting address pointers for a specified line. Calculates the screen memory address corresponding to the leftmost column of the screen line specified in the X register and places the address in the pointer at 224-225/$EO-$E1. The low byte of the address comes from the table at 49203/$C033 (multiplied by 2 if the 80-column display is active). The high byte is calculated by taking a value from the table at 49228/$C04C, masking off all but the lowest two bits (and multiplying the result by 2 if the 80-column screen is active), then ORing the result with the starring page value for the active screen—2619/$0A3B for the 40-column display or 2606/$0A2E for the 80-column display.

Next {at 49532/$C17C), the attribute memory address corresponding to the leftmost column of the specified screen line is calculated and placed in the pointer at 226-227/$E2-$E3. The low byte of the address is taken directly from the low byte of the screen memory pointer (224/$E0). For the

40-column screen, the high byte is calculated by masking off all but the lowest two bits of the screen memory pointer high byte (225/$El), then ORing the result with the value 216/$D8 (40-column color memory always starts at $D80O). For the 80-column screen, the high byte is calculated by masking off all but the lowest three bits of the screen memory pointer high byte (225/$El), then ORing the result with the attribute memory starting page value (2607/$0A2F).

49556 $C194 SCNIRQ

Performs screen and keyboard portion of IRQ functions. (This routine has a jump table entry at 49188/$C024.)

Begins by checking whether the current IRQ is the result of a raster interrupt (see Appendix A for a discussion of interrupts). If it's not, then this can't be a normal jiffy IRQ, so the routine skips the screen setup steps and just scans the keyboard and blinks the cursor before exiting. If it is a raster interrupt, the routine checks whether the screen mode flag (216/$D8) contains the value 255/$FF. If so, the routine again skips ahead to scan the keyboard, blink the cursor, and exit. The 128 itself never puts that value into the flag, but you can use this feature when you set up your own custom screen mode or raster interrupt (see the entry for location 216/$D8 for details).

The routine then checks whether the raster is currently on scan line 256 or higher. If so, the interrupt occurred while the raster was beyond the visible portion of the screen. This offscreen raster interrupt is the normal jiffy IRQ source for the 128—quite a different technique from the CIA timer-driven jiffy IRQ for the Commodore 64. (The interrupt is actually triggered at line 255, but the raster will have moved to the next scan line by the time its position is checked here.) For an offscreen interrupt, the VIC-II chip registers will be set according to the screen mode flag value, using the step at 49631/$C1DF for mode 0 (a full text screen) or the one at 49587/$C1B3 for a bitmapped or split-screen mode.

If the interrupt occurred above scan line 255, bit 6 of the screen mode flag is tested. If that bit is set to %1, a split screen mode is in use, so a branch is taken to 49631/$C1DF to set VIC-II registers for the text portion of the display. Otherwise, the registers are merely reset for whichever mode is currently in use, which produces no obvious effect.

At 49587/$ClB3, the bitmapped or multicolor bitmapped

screen is established. If a split screen is in use, the scan line value for the split, stored in 2612/$0A34, is loaded into the VIC-II's raster compare register (53266/SD012). The processor I/O register (l/$01) is set to make block 0 of color memory—the block for multicolor source 3—visible at 55296/SD800, and the VIC memory control register (53272/$D018) is loaded with the bitmap base address, stored in 2605/$0A2D. Bit 5 of VIC control register 1 (53265/$D011) is set to %1 to enable the bitmapped display. If the mode flag indicates that multicolor bitmapped mode was selected, bit 4 of VIC control register 2 (53270/SD016) is also set to %1 to enable that feature.

At 49631/$C1DF, the 40-column text screen (or the text area of a split screen) is established. The VIC-II's raster compare register (53266/$D012) is loaded with 255/$FF to trigger the offscreen raster interrupt. The processor I/O register (l/$01) is set to make block 1 of color memory—the block for 40-column text color—visible at 55296/$D800, and the VIC memory control register (53272/$D018) is loaded with the screen and character memory base value, stored in $2604/$0A2C. The VIC control register bits for bitmapped and multicolor are cleared to disable those display modes. Finally, a short delay loop is executed if the text portion of a split screen is being established. This is to insure that the switch from bitmapped to text modes occurs while the raster is off the right edge of the visible screen area, which prevents the flicker that would occur if the change occurred in the middle of scanning a line.

If the IRQ source is a raster interrupt which occurred on the visible screen (to set up the text portion of a split screen), the routine exits at this point with the status register carry bit clear. In all other cases, the routines to scan the keyboard for a keypress [$C55D] and handle 40-column cursor blinking [$C6E7] are called, and the carry bit will be set upon exit.

49716 3C234 KEYIN

Performs GETIN from keyboard.
(This routine has a jump table entry at 49158/$C006.)
Checks 2O9/$D1 to see whether any characters remain to be read from the definition string for the most recently pressed programmable key. If so, a character is read from the definition area, with the value in 210/$D2 used as an offset from the start of the definition area (4106/$100A). The count of re-

maining characters is decremented and the offset into the definition area is incremented; then the routine exits with the character in the accumulator, the status register carry bit clear, and IRQ interrupts reenabled.

If no programmable key string characters are pending, the character at location 842/$034A, the first character in the keyboard buffer, is returned in the accumulator. Any remaining characters in the buffer are shifted one position to the left to remove the retrieved character from the buffer; then the count of characters in the buffer (208/$D0) is decremented. Upon exit, the status register carry bit will be clear and IRQ interrupts will be reenabled.

Unlike other major screen editor routines with jump table entries, you should avoid calling this one directly. To retrieve keypresses, call the Kernal GETIN routine via its jump table entry (65508/$FFE4) with the current input device (153/$99) set to zero to indicate keyboard input. The Kernal GETIN routine [$EEEB] verifies that a character is waiting to be read from either a programmable key string or the keyboard buffer before it calls this routine. If you call this routine with no characters available (with 208/$D0 and 2O9/$D1 both containing zero), you'll cause serious problems. The contents of all page 3 locations above the keyboard buffer (and part of 40-column screen memory in page 4) will be garbled. This area includes the vital indirect fetch and store routines; you'll have to reset the computer to recover.

49752 8C258 KEYLIN

Accepts a line of keyboard input and returns the first character. (The normal entry point for this routine is 49755/$C25B.)

Turns on the cursor, then waits until a character is available in the keyboard buffer or from a programmable key definition string. The routine above is called to retrieve the character, and unless it's a RETURN character, code 13/$0D, the routine loops to display the character on the screen at the current cursor position (using the PRINT routine [$C72D]), move the cursor to the next position, and wait for another character to become available. When RETURN is encountered, the routine falls out of this loop and stores the RETURN code in the input source flag (214/$D6) so that subsequent calls to the keyboard BASIN routine will retrieve characters from the line of input that has been displayed on the screen. The routine then sets

pointers to the input screen line and jumps to 49852/$C2BC in the following routine to exit with the accumulator holding the first character of the input—unless no other characters were typed before RETURN, in which case the routine branches to 49829/$C2A5 in the following routine to exit with the RETURN character code in the accumulator.

49819 8C29B GETSCRN

Performs BASIN from screen or keyboard.
(This routine has a jump table entry at 49161/$C009.)

Stashes the contents of the X and Y registers on the stack for later restoration, then checks the input source flag (214/$D6). If the flag value is zero, indicating input from the keyboard, the preceding routine is used to accept a line of input from the keyboard. If the flag is nonzero, but with bit 7 clear (%0), the routine branches ahead to 49852/$C2BC to retrieve a character from a screen line. If bit 7 is set (%1), the end of the input line has been reached, so (at 49829/$C2A5) the input source flag is reset to zero and a RETURN character is printed unless output is to a device other than the screen. The routine then exits with the RETURN character code, 13/$0D, in the accumulator (the X and Y registers will be restored to their original values).

At 49852/$C2BC, the screen code at the current cursor position is retrieved and converted into the corresponding character code (disassemble $C2C2-$C2D8 to see the conversion process). The routine then checks whether the end of input has been reached. If so, bit 7 of the input flag is set to %1 to indicate that all characters have been read from the current input line.

The cursor is then moved right to the next character position in the line. If the calculated character code is 222/$DE, the pi *(K)* character, it's replaced with 255/$FF, an alternative value (and BASIC token) for that character. Before exiting, the original X and Y register values placed on the stack at the start of the routine are restored, and retrieved character code is loaded into the accumulator. The status register carry bit will be clear upon exit.

Although there are no bugs in this routine, its calling Kernal routine (BASIN [$EF06]) will not correctly retrieve a line of input from the screen (keyboard input functions properly). The screen input setup portion of BASIN has two major

flaws: It does not set the ending row for input (2608/$0A30), and it resets the input source flag (214/$D6) before each character is retrieved (so that the setting of bit 7 is lost). Together, these bugs make it impossible to determine when the end of the input is reached. The following is an example of the proper technique for retrieving a line of input from the screen:

```
        LDA   $EB       ;Set ending row for input
        STA   $0A30     ; (same as starting row)
        LDA   $E7       ;Set ending column for input
        STA   $EA       ; (to right window margin)
        LDA   #$03      ;Set screen as input source
        STA   $D6       ; (any nonzero value will do)
        LDX   #$00      ;Initialize storage offset
LOOP    JSR   $C009     ;Read character from the screen
        CMP   #$0D      ;Is it RETURN?
        BEQ   EXIT      ;Exit if so
        STA   $0C00,X   ;Stash character from screen
        INX             increment the offset
        BNE   LOOP      ;Loop for another character
EXIT    RTS
```

49919 $C2FF QUOTECK

Handles quote mode flag.

Checks whether character value in the accumulator is 34/$22, the quote (") character. If so, bit 1 of the quote mode flag (244/$F4) is toggled, turning quote mode on if it was previously off or off if it was previously on. The character value will still be in the accumulator upon exit.

49932 $C30C PRNTEXIT

Provides common exit for screen BSOUT subroutines.

Stores the current character code (239/SEF) as the previous character code (240/SFO). The subroutine to move the cursor on the 80-column screen [$CD57] is called to actually move the cursor if that display is active. This is necessary because the 80-column screen's cursor is controlled by hardware (not by software, like the 40-column screen's cursor), so it doesn't automatically move when its corresponding row and column pointers are updated. The pending insert count (245/$F5) is checked. If it's nonzero, the quote mode flag is cleared. As a result, you won't go into quote mode if you type the quote character for a pending insert unless the quote is typed for the last insert. Before exiting, the routine reloads the accumulator

and X and Y registers with the values stored on the stack when the BSOUT routine was entered [$C72D], so all register values are preserved during BSOUT. The status register carry bit will be clear upon exit.

Because the address of this routine is pushed onto the stack at the start of the screen BSOUT routine [$C72D], all screen printing subroutines will return to the calling routine via this routine.

49952 $C320 SETCHAR

Handles character printing for screen BSOUT.

Handles all character display functions for the screen BSOUT routine [$C72D], including quote, reverse, and autoinsert modes. The routine has several entry points, depending on the desired effect. For any of these entry points, the code to be printed should be in the accumulator upon entry:

49952/$C320: Entry point for printing character codes 160-254/$A0-$FE. Prior to entry, bit 7 of the code will have been cleared to %0. This step forces bit 6 to %1, so character codes 160-191/$A0-$BF become screen codes 96-127/ $60-$7F, and character codes 192-254/$C0-$FE become screen codes 64-126/$40-$7F.

49954/$C322: Entry point for printing character codes 3 2-12 7/$ 20-$ 7F. Prior to entry, the character code will have been converted to a screen code by the screen BSOUT control routine [$C72D]. This step checks whether reverse mode is active (indicated by a nonzero value in 243/$F3). If that mode is not active, the next step is skipped.

49958/$C326: Entry point for printing characters in reverse video; also used for character codes less than 32 and for codes 128-159 when those characters are printed in quote mode or at a pending insert. For codes 1-31/$O1-$1F, the character code is used unchanged, so those characters are represented by the reverse images of screen codes 1-31/$O1-$1F. For characters 128-159/$80-$9F, bit 7 will have been cleared to %0, and bit 6 set to %1 prior to entry, so those characters are represented by the reverse images of screen codes 64-95/$40-$5F.

If autoinsert mode is active (indicated by bit 7 of 246/$F6 set to %1), a space is inserted at the current cursor position before the character is printed. (The count of inserts pending is also cleared; you can't have inserts pending in autoinsert mode.) The subroutine at 52271/$CC2F is used to place the

character on the screen at the current cursor position using the current attribute value from 241/$F1; then the routine falls through into the following one to move the cursor to the next position.

49982 $C33E UDCRSR

Updates the cursor position.

Compares the current cursor column (in the Y register upon entry) against the right window margin (231/$E7). If the cursor is at the right edge of the window, the routine checks whether the cursor is also already on the bottom line of the window. If so, the scrolling enable flag (248/SF8) is checked. The cursor will not be moved if scrolling is not allowed when the cursor reaches the lower right corner of the window. Otherwise, the cursor is moved right one position. If the cursor moves beyond the right window margin, a new blank line is inserted and linked to the one above, and the cursor is moved to the left margin of this new line. However, linking can be disallowed by setting bit 6 of 248/$F8 to %1. In this case, the cursor will simply wrap around to the left margin without inserting a new line. This link-disable feature is not directly supported on the 128—there's no character or ESC sequence to disable linking—so you must set or clear the bit directly. See the entry for location 248/$F8 for details.

50019 8C363 NEXTLIN

Moves the cursor down one line.

Moves the cursor down to the next row in the window. If the cursor is already on the bottom row of the window, the scrolling enable flag (248/$F8) is tested. If scrolling is allowed, the window contents are scrolled upward by one row, and the bottom row (where the cursor now resides) is cleared. If scrolling is disabled, the cursor is wrapped to the top row of the window.

50044 $C37C OPENLIN

Inserts a new line linked to the one above.

Copies all lines in the window below the one on which the cursor currently resides down one row (the bottom line will be lost). The line on which the cursor resides is then cleared and linked to the one above.

50086 $C3A6 SCROLL

Scrolls the window up one line.

Copies all lines in the window up one row. The top line will be lost and the bottom line will be cleared. If the new top line was originally linked to the line scrolled off the window, it will be unlinked.

50140 $C3DC SCRLUP

Copies lines up one row and clears bottom line.

Copies all lines in the window below the one specified in the X register up one row. The specified line will be overwritten, and the bottom line will be cleared and unlinked. The line link bits for each line are also copied to the new positions so that the links will be maintained intact.

This routine also checks column 7 of the keyboard matrix (see the entry at 50525/$C55D) to test the Commodore key. If that key is found to be pressed, a delay loop is executed. Since this is the subroutine used to open new lines at the bottom of the window while printing, this feature allows the Commodore key to slow screen scrolling. There is no simple way to defeat this feature.

50189 $C40D MOVLINE

Copies a line.

Copies characters from the line specified in the X register to the one pointed to in 224-225/$E0-$El, and copies the attributes for the line to the attribute line pointed to in 226-227/$E2-$E3. The copying begins at the column specified in the Y register and extends to the left window margin. The copy is performed on whichever display, 40- or 80-column, is currently active. The X register will still contain the source line number upon exit.

50341 $C4A5 CLRLINE

Clears a line.

Writes a space character with the current attribute (from 241/SF1) at every character position between the left and right window margins of the line specified in the X register. The operation is performed on whichever display, 40- or 80-column, is currently active. The link bit for the line is also cleared, unlinking the line. Upon exit, the X register will still contain the

line number and the Y register will hold the column number of the right window margin.

50492 $C53C

Fills or copies a **block of** 8563 RAM.

Uses the hardware fill/copy feature of the 8563 chip to fill or copy a block of the 8563's private 16K of RAM. The block starts at the address currently in 8563 registers 18-19/$12-$13 and ends at the address in 2620-2621/$0A3C-$0A3D. The operation is determined by the state of bit 7 of 8563 register 24/$18. If the bit is %0, the block is filled with the value in 8563 register 31/$1F. If the bit is %1, the contents of locations starting at the address in registers 32-33/$20-$21 are copied to the block.

50525 $C55D SCNKEY

Scans keyboard matrix for keypress.

(This routine has a screen editor jump table entry at 49170/SC012 and a Kernal jump table entry at 65439/$FF9F.)

Begins by checking bit 6 of location 1/$01 to determine the position of the CAPS LOCK key. If that key is down, the shift key flag (211/$D3) will be set to 16/$10 (bit 4 set to %1); otherwise, it will be cleared to zero. Next, the keyscan code (212/$D4) is initialized to 88/$58, the value for no key pressed. The keyboard matrix is checked to see whether any keys are pressed. If not, the routine jumps to 50839/$C697 in the next routine to set 88 as the matrix code value. If a key has been pressed, the routine loads the keyboard decode table pointer (204-205/$CC-$CD) from 830-831/$033E-$033F, which holds the address of the default table, and begins the process of determining which key is being pressed.

The keyboard is arranged electrically as a matrix of 11 columns and eight rows (Figure 7-1). To scan for a keypress, the CIA or VIC port bit for one of the 11 columns is set to %0, while the bits for all other columns are held at %1. While the column bit is %0, the CIA port for the eight rows (56321/$DC01) is read, and, one by one, the bits corresponding to the rows are tested. The count of scanned keys is incremented after each row bit is tested, so each key in the matrix has a keyscan code corresponding to the number of keys scanned when that key's position is tested. The codes range from 0, for

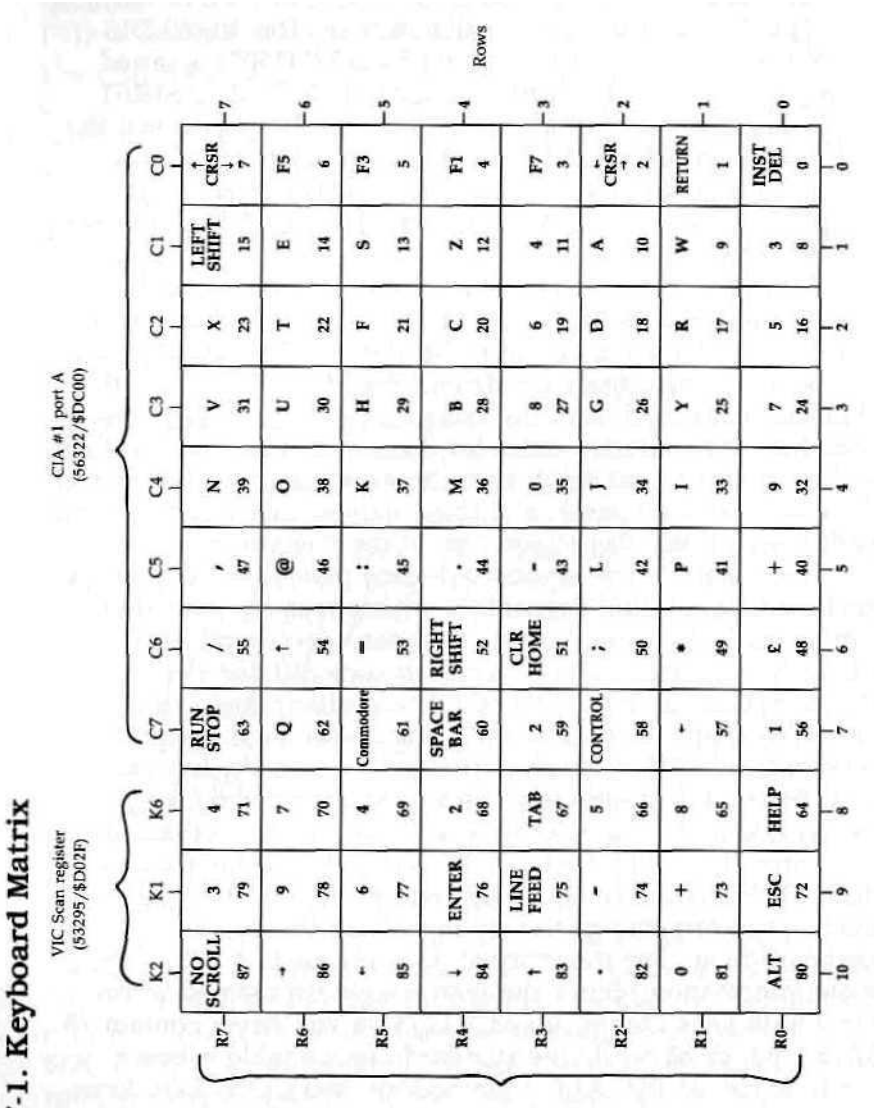
the key at row 0 of column 0 (INST/DEL), to 87, for the key at row 7 of column 10 (NO SCROLL).

Several keys on the 128 keyboard are not included in the keyscan matrix. The CAPS LOCK key (read at bit 6 of location l/$01) is tested at the start of this routine. The 40/80 DISPLAY key (read at bit 7 of location 54533/$D505) is tested during the screen initialization routine [$C07B]. The SHIFT LOCK key is a switch that has the effect of holding down the left SHIFT key. The RESTORE key isn't read at all; it's connected to circuitry which generates an NMI interrupt signal to the processor when the key is pressed (see the NMI handling routine [$FA40]).

If the key at the intersection of the row and the column being tested is pressed, the row bit will be %0; otherwise, it will be %1. If a key is found to be pressed, the character code for the key is read from the default decode table. Unless the character code read from the table is 8, 4, 2, or 1, the keyscan code (not the character code) for the key is stored in 212/$D4, and the routine goes on to scan the next position. However, if the character code has one of these values, that value is ORed with the shift key flag to set a bit in the flag value.

Note that it's the keyboard decode table, not the physical keyboard layout, that determines which keys are treated as shift keys. In the default ROM keyboard decode table at 64128/$FA80, the ALT key (keyscan code 80) decodes as character code 8, the CONTROL key position (keyscan code 58) holds character code 4, the Commodore logo key position (keyscan code 61) holds character code 2, and the left and right SHIFT key positions (keyscan codes 15 and 52, respectively) both hold character code 1. Thus, pressing either SHIFT key normally sets bit 0 of 211/$D3 to %1, and the Commodore, CONTROL, and ALT keys set bits 1, 2, and 3, respectively. (You can change this by modifying the default keyboard table. See the entry at 830-831/$033E-$033F for more information.) Since the matrix code isn't stored when these shift keys are processed, 212/$D4 will never contain 15, 52, 58, 61, or 80 while the standard decode table is being used. So the SHIFT, ALT, Commodore, and CONTROL keys can't normally be detected by checking 212/&D4. Their states must be determined from 211/$D3.

Figure 7-1. Keyboard Matrix



All eight rows in each of the 11 columns are scanned, whether keypresses are detected or not. Since there's only one location (212/$D4) for storing the matrix code, only the key with the highest code value will be recorded if more than one key is detected during the scan. However, because of the way the keyboard is wired, a false value may be returned if three or more keys are held down simultaneously. For example, if you hold down the J, K, and L keys simultaneously, the matrix code returned will be 45, the code for the colon (:) key. The special shift keys are an exception: 211/$D3 will show the state of all five (SHIFT, Commodore, CONTROL, ALT, and CAPS LOCK) on every keyscan.

Once the keyscan is complete, the matrix code of the keypress is loaded into the accumulator. At this point, the routine takes an indirect jump through the KEYVEC indirect vector (826-827/$033A-$033B). Normally, this vector points to 5O657/$C5E1—the address immediately following the indirect jump. However, if you wish to manipulate the keyscan in any way, you can redirect the vector to your own routine in RAM. See the KEYVEC entry for details.

Next, the routine checks whether the matrix code was 87/$57, the code for the NO SCROLL key. If so, the routine checks bit 6 of the flag at 247/$F7 to determine if pausing is allowed. If the bit is set, printing cannot be paused. This enable/disable feature isn't directly supported—there's no character or ESC sequence to set or clear bit 6 of the flag location—so y_{OU} must change the location directly (see the entry for location 247/$F7 for details). If pausing is allowed, the pause flag (2593/S0A21) is toggled; so pressing NO SCROLL will alternately halt and resume printing to the screen.

NO SCROLL is a misnomer. The key doesn't stop scrolling; it halts printing. PAUSE would have been a better name for this key. The pause feature is implemented in the screen BSOUT routine [$C72D].

The shift key flag (211/$D3) is tested to determine whether the Commodore and SHIFT keys are both pressed. If so, bit 7 of the case switching flag (247/$F7) is tested to determine whether this combination is allowed to switch character sets If so, the character set in use is switched, either by toggling bit 7 of the attribute flag (241/$F1) if the 80-column display is active, or by toggling bit 1 of the character set base address (2604/$0A2C) if the 40-column display is active.

If the shift key flag indicates that the CONTROL key is pressed, the routine checks whether the current matrix code is 13/$0D, indicating that CONTROL and the S key have been pressed simultaneously. The CONTROL-S combination provides a printing pause similar to NO SCROLL. Bit 6 of the flag at 247/$F7 is checked before the pause flag value is set so pausing with CONTROL-S can be disabled as well. If pausing is allowed, the pause flag (2593/$0A21) is loaded with the S key's matrix code value, 13/$0D.

A keyboard decoding table is then selected from the table of addresses at 830-841/$033E-$0349, based on the value in the shift key flag (211/$D3). The table address is loaded into the pointer at 204-205/$CC-$CD before falling through into the next routine. Since only one table can be used, the following order of precedence applies if more than one shift key has been pressed: CONTROL has the highest priority; when it is pressed, the CONTROL decoding table is selected regardless of which other shift keys are pressed. For example, ALT-SHIFT-CONTROL-W is the same as CONTROL-W. CAPS LOCK and ALT are effective only if pressed when no other shift keys are pressed. For example, ALT-SHIFT-S is the same as SHIFT-S. If ALT is pressed while CAPS LOCK is down, both are ignored and the unshifted table is used. Likewise, if the SHIFT and Commodore keys are pressed simultaneously while case switching is disabled, both are ignored and the unshifted table is selected.

50769 SC651 REPEAT

Decodes key matrix value into character value and handles key repeating.

(This routine has a jump table entry at 49173/$C015.)

Uses the keyscan matrix code in 212/$D4 as an offset into the keyboard decode table pointed to by 204-205/$CC-$CD to find the character code for the current keypress. The current matrix code is then tested against the previous matrix code. If they are not the same, the countdown before repeating begins (2596/$0A24) is reset to 16/$10 and the following test for repeating keys is skipped.

If the current code is the same as the previous one, it means that the key is being held down. In this case, the repeat enable flag (2594/$0A22) is tested to see whether any key repeats are allowed. If bit 7 of the flag is set to %1, all keys re-

peat (the default setting for the 128). If bit 6 is set to %1, no keys repeat, so the routine exits. If bits 6 and 7 of the repeat enable flag are both %0, only the cursor, space, and INST/DEL keys repeat, so the routine exits if the character is not one of these.

To keep repeats from happening too fast, a countdown location (2596/$0A24) is decremented once per call to this routine. The countdown, initialized to 16, must reach zero before repeating begins, so a key must be held down long enough for 16 IRQ keyscans to pass before repeat processing begins. After the initial delay expires, a second countdown (2595/$0A23), initialized to 4 between each repeat, must also expire before the keypress is counted a second time. Thus, the delay before repeating begins is (16 + 4)/60 = 1/3 second, and the delay between subsequent repeats is 4/60 = 1/15 second. These values are fixed in the ROM routine and can't be changed as long as the standard keyscan is used. For repeating keypresses, the character code is added to the keyboard buffer only if the buffer is currently empty—a feature that prevents repeats from filling up the buffer.

At 50839/$C697, a special countdown flag (2597/$0A25) used to debounce NO SCROLL, CONTROL-S, and SH1FT-Commodore keypresses is decremented. Then the current matrix code (212/$D4) is stored as the previous matrix code (213/$D5). If the value read from the keyboard decoding table is 255/$FF, the routine exits at this point, since that value signals that there was not a valid character code for the key. Otherwise, the pause flag (2593/$0A21) is cleared to zero, so printing will resume upon completion of this IRQ if it was previously halted.

The character code is then loaded into the accumulator, and the value in the shift key flag (211/SD3) is loaded into the X register. Then the routine takes an indirect jump through the KEYCHK indirect vector (828-829/$033C-$033D). Normally, this vector points to 50861/$C6AD, the address immediately following the indirect jump. However, if you wish to manipulate the character code in any way, you can redirect the vector to your own routine in RAM. See the KEYCHK entry for details.

Next, the routine checks whether one of the programmable keys has been pressed by comparing the character code in the accumulator values from the table at 50909/$C6DD. If a

match is found, the count of pending programmable key string characters (2O9/$D1) is loaded with the length of the corresponding definition string from the table at 4096-4105/$1000-$1009, and the offset into the definition area at 4106/$100A for the string is calculated and stored in 210/$D2. (A simple way to defeat this feature and make the function keys simply generate character codes as they do in the Commodore 64 is to change the KEYCHJC vector to point to 50871/$C6B7 so that this portion of the routine is skipped,)

If the character isn't one of the programmable keys, the number of characters currently in the keyboard buffer (208/$D0) is compared against the maximum number the buffer can hold (2592/$0A20). If the buffer is full, the current keypress is ignored. Otherwise, the character code is stored in the next available position in the buffer at 842-851/$034A-$0353. From there, it can be retrieved by the Kernal GETIN and BASIN routines.

Before exiting, the value 127/$7F is stored in the CIA I/O port register for keyboard columns. This value (%01111111) sets the line for column 7 of the matrix low, while all other lines are high, so that the keys in that column can be detected by reading the row register (56321/$DC01). This feature is provided to allow testing for the RUN/STOP and Commodore keys, but other keys on that row can also be detected outside the normal keyscan. See the routine at 63037/$F63D and the entry for location 145/$91 for more information.

50909 $C6DD PFKCHRS

Table of programmable key character values.
This table is used to identify programmable keys and to determine the offset into the definition length table at 4096/$1000 for the length of the corresponding string.

Offset	Character Code	Key
0	133/$85	Fl
1	137/$89	F2
2	134/$86	F3
3	138/S8A	F4
4	135/$87	F5
5	139/$8B	F6
6	136/$88	F7
7	140/$8C	F8
	131/$83	SHIFT-KUN/STOP
9	132/$84	HELP

50919 6C6E7 CRSR40

Handles cursor blinking for 40-column screen.
Begins by checking the active screen flag (215/$D7), exiting immediately if the 80-column display is active. The routine also exits if the cursor blink switch (2599/$0A27) holds a non-zero value. Then, the cursor blink countdown value (2600/$0A28) is decremented. The countdown is necessary because the cursor can't be blinked during every interrupt—that would result in blinking too fast to see. If the cursor countdown doesn't reach zero, the routine exits. If the count does reach zero, the cursor flag (2598/$0A26) is tested. If that flag indicates that the solid cursor is selected and the cursor is reversed, the routine exits (the solid cursor just stays reversed). Otherwise, it's time to switch the cursor's phase.

The first step in switching phases is to reset the countdown to its starting value—20/$14. That value will result in the cursor changing phase every 20/60 second. Because it takes two phase changes (on-to-off and off-to-on) to blink the cursor, the cursor blinks every 40/60, or 2/3, second. Since this value is part of this ROM routine, it's not possible to change the cursor blink rate while using the standard screen IRQ routine.

If the cursor is currently in its reversed phase, the position is restored to its original color and screen code. If the cursor is in its normal phase, the screen code and color for the position are stashed (in 2601/$0A29 and 2602/$0A2A, respectively). Then the high bit of the screen code is toggled, and the reversed screen code is written back to the current cursor position using the current cursor color (in 241/$F1). Finally, the cursor phase bit (bit 6) in the cursor flag (2598/$0A26) is toggled.

This routine is part of the normal IRQ sequence. There is no corresponding routine to blink the cursor on the 80-column screen because the 8563 80-column chip handles that operation automatically.

50989 8C72D PRINT

Handles BSOUT to the screen.
(This routine has a jump table entry at 49164/$C00C.)
Begins by storing the current character code (in the accumulator upon entry) in 239/$EF. The current accumulator and X and Y register values are then stashed on the stack for restoration upon exit.

The next portion of the routine implements the pause printing feature. If the pause flag (2593/$0A21) contains a nonzero value, the routine immediately loops to check the flag again. It remains in the flag testing loop until the flag value is reset to zero. If you're wondering how the routine ever breaks out of this loop, remember that normal processing is suspended during interrupts. The flag value is set during the IRQ-driven SCNKEY routine. This means that if you ever call BSOUT while interrupts are disabled, you must be sure that the pause flag contains a zero. Otherwise, the flag value will never change, and the routine will be stuck in the loop until you reset the computer. It also means that, as long as the normal IRQ is enabled, NO SCROLL and CONTROL-S can be used to pause screen printing in any program that calls BSOUT—even in your own machine language programs. Remember, however, that since the pause loop is implemented only in this routine, NO SCROLL and CONTROL-S can halt printing to the screen only.

Next, the input source flag (214/$D6) is reset to zero, making the keyboard the input source. The value 49931/$C30B is pushed onto the stack. This will cause the RTS at the end of the printing subroutine—or at the end of the special handling subroutine in the case of screen and color control characters—to jump to 49932/$C30C, the common exit routine for all screen BSOUT subroutines.

If the code to be printed is 13/$0D (RETURN) or 141/$8D (SHIFT-RETURN), a branch is taken to 51055/$C76F to handle these special cases. Otherwise, 240/$F0 is checked to see whether the last character printed is the ESC (escape) character, 27/$lB. If so, the code is the second character of an ESC sequence, so a jump is taken to the ESC sequence handler at 51646/$C9BE. Character codes greater than 127/$7F are sent to a special handling routine at 51202/SC802, and codes less than 32/$20 are sent to 51126/$C7B6.

Remaining characters, those with codes in the range 32-127/$20-$7F, are converted to screen codes as follows: If the code is less than 96/$60, bit 6 is cleared. Character codes 32-63/$20-$3F are unchanged, becoming screen codes 32-63/$20-$3F, while character codes 64-95/$40-$5F become screen codes 0-31/$00-$1R If the code is 96 or greater, only bit 5 is cleared, so character codes 96-127/$60-$7F become screen codes 64-95/$40-$5F. Next, the subroutine at 49919/$C2FF

is called to handle the quote (") character. Then the routine jumps to 49954/$C322 to display the calculated screen code at the current cursor position using the current attribute (241/$F1).

Remember that the address pushed onto the stack causes the final printing or character handling subroutine called by this routine to return to 49932/$C30C. That routine restores the accumulator and X and Y registers from the stack, so they contain the same values upon exit that they did upon entry.

51055 8C76F RTRN

Handles RETURN and SHIFT-RETURN characters, CHR$<13> and CHR$(141).

Finds the position of the last nonspace character in the logical line; then clears the link bit for the physical line just below that position, making the line following the current logical line the start of a new logical line. The cursor position is set to the left margin of that line, and the screen line pointers are updated to reflect the new cursor position. The routine then falls through into the following one.

51069 $C77D MODESOFF

Cancels quote and reverse modes and clears pending inserts (ESC-O and ESC-ESC).

Bits 4 and 5 of the current attribute value (241/$F1) are cleared, canceling flashing or underlined modes for subsequent characters on the 80-column display. The pending insert (245/$F5), reverse mode (243/$F3), and quote mode (244/$F4) flags are all reset to zero, canceling any pending inserts and turning off reverse and quote modes.

51084 $C78C

Tables of screen control codes and dispatch addresses.

Locations 51084-51097/$C78C-$C799 comprise a table of all the character codes less than 32 which have a special function (except for RETURN and ESC, codes 13 and 27, which are detected in the main screen BSOUT routine [$C72D]). The subroutine to handle codes less than 32 [$C7B6] compares the current character against these values. If a match is found, the corresponding dispatch address from the table at 51098-51125/$C79A-$C7B5 is placed on the stack to be called by the next RTS. Because of the way the RTS opcode behaves, adding 1 to the address placed on the stack, the address table values are 1

less than the actual address of the target routine. The table shows the characters handled and the actual addresses of the handling routines:

Offset	Character Code	Handling Routine	Description
0	2/$02	51399/$C8C7	Turns on underline mode
1	7/$07	51598/$C9BE	Generates bell tone
2	9/$09	51535/$C94F	Moves cursor to next tab stop
3	10/$0A	51633/$C9B1	Performs linefeed
4	11/$OB	51366/$C8A6	Disables case switching
5	12/$0C	51372/$C8AC	Enables case switching
6	14/$0E	51328/$C880	Switches to lowercase set
7	15/$0F	51413/IC8D5	Turns on flashing mode
8	17/$11	51290/$C85A	Moves cursor down one line
9	18/$12	51394/$C8C2	Turns on reverse mode
10	I9/$13	51379/$C8B3	Moves cursor to home position
11	20/$14	51483/$C91B	Deletes a character
12	24/$18	51553/$C961	Sets or clears a tab stop
13	29/$lD	51284/$C854	Moves cursor right one column

51126 6C7B6

Interprets character codes less than 32.

Jumps indirectly through the CTLVEC vector (820-821/ $0334-$0335), which is initialized on system reset to point to 51129/10769, the location immediately following the indirect jump. However, the vector can be redirected to your own routine in RAM, allowing you to modify the effects of printing any of the characters with codes less than 32. See the entry at CTLVEC for details.

If the value in the accumulator is the ESC character, 27/$lB, the routine exits without performing any other checking. Otherwise, the routine checks whether an insert is pending. If so, the routine branches to handle the character just as it would if quote mode were active. If no inserts are pending, the code is compared with the delete character value 20/$14, If the code matches, the routine branches past the following test for quote mode. Because delete character handling is sandwiched between the test for pending inserts and the test for quote mode, it's possible to type deferred deletes when inserts are pending, but not during quote mode. Delete is the only character singled out for this treatment: All other control characters (except RETURN and ESC) are deferred when printed while quote mode is active.

If quote mode is active, the routine clears the character storage location (239/$EF) and branches to 49958/SC326 in the character printing routine to display the character code (0-31) as a screen code in reverse mode. If quote mode is not active, the accumulator contents are compared against the 14 control character codes in the table at 51084/$C78C. If a match is found, the dispatch routine [$C7F6] is used to execute the corresponding subroutine. Otherwise, the routine falls through into the next routine to check whether the character is a color change code.

51162 $C7DA COLORSET

Handles color change characters.

Compares the character code in the accumulator with the table of color character codes at 52812/$CE4C. If no match is found, the routine exits. If the accumulator value matches a table entry, the X register will contain the table offset for the match, a value in the range 0-15. For the 40-column display, this value is placed in the current attribute flag value (241/SF1). For the 80-column display, the offset value is used as an index into the table at 52828/$CE5C to retrieve the corresponding 8563 color code. That value is then placed in the lower four bits of the attribute flag.

51190 8C7F6

Calls control code execution routines.

Uses the control code index, in the X register upon entry, as an offset into the dispatch table at 51098/$C79A. The corresponding subroutine address is retrieved from the table and pushed onto the stack, so the RTS at the end of this routine will transfer control to the subroutine.

51202 SC802

Interprets character codes greater than 127.

Jumps indirectly through the SHFVEC vector (822-823/ $0336-$0337), which is initialized on system reset to point to 51205/$C805, the location immediately following the indirect jump. However, the vector can be redirected to your own routine in RAM, allowing you to modify the effects of printing any of the characters with codes greater than 127. See the entry at SHFVEC for details.

Next, the high bit of the character code (in the accumu-

lator upon entry) is masked off, and the resulting value is compared against 32. If the value is less, indicating that the original character code was in the range 128-159, a branch is taken to the following routine to handle those characters. Otherwise, the routine tests whether the result was 127, indicating that the original code was 255, the pi (rc) character. If not, the routine jumps to 49952/$C320 in the character printing routine to display the character corresponding to the specified code. However, if the code was pi, it's changed to 94/$5E, the screen code for the pi character, before jumping to the printing routine.

51220 $C814

Handles character codes 128-159.

Checks whether quote mode is in effect and, if so, sets bit 6 of the character value in the accumulator (converting the adjusted character code value 0-31 into a screen code value in the range 64-95), then jumps to 49958/$C326 in the character printing routine to display the control code as a reverse character.

If quote mode is not in effect, the code tests against the value 20/$14 to determine if it was originally the insert character— CHR$(148). If so, the routine jumps to 51427/$C8E3 to handle the insert. Otherwise, the count of pending inserts is checked. If inserts are pending, the character is treated as if quote mode were in effect—it's converted to a screen code and printed in reverse video. Performing this test after checking for the insert character allows multiple insertions to be made.

If no insert is pending, the accumulator value is compared against a series of character values, branching or jumping to the appropriate handling routine if a match is found. The table shows the address of the routine to which control is transferred if a match is found. (The table shows the original character code value before the high bit is masked off; the value actually tested for in this routine will be 128 less):

Character Code	Handling Routine	Description
145/$9l	51303/$C867	Moves cursor up one line
157/$9D	51317/$C875	Moves cursor left one position
142/$8E	51346/$C892	Switches to uppercase/graphics set
146/$92	51391/$C8BF	Turns off reverse mode
130/$82	51406/$C8CE	Turns off underlining
143/$8F	51420/$C8DC	Turns off flashing characters
147/$93	49474/$C142	Clears the window

If the code is not among this group, bit 7 of the accumulator is reset to %1 to restore the character to its original value. Then the routine branches to 51162/SC7DA to test whether this is a color change character.

51284 $C854

Handles cursor right character, CHR$(29).

Calls the subroutine to move the cursor one position to the right [$CBED], then tests whether the cursor is wrapped to the left margin of a new line. If so, a branch is taken to the routine which determines whether the cursor has moved down onto a new logical line.

51290 **$C85A**

Handles cursor down character, CHR$(17).

Calls the subroutine to move the cursor down one row [$C363]. Upon return, the routine falls through into the next routine to see whether the cursor has moved onto a new logical line.

51293 $C85D

Checks whether cursor moved onto a new logical line.

Checks whether the link bit for the current physical line is set to %1, indicating that the cursor is still on the same logical line. If not, bit 7 of 232/SE8 is set to %1 to indicate that the input begins on the first line in a logical chain,

51303 $C867

Handles cursor up character, CHR$(145).

Exits without moving if the cursor is already on the top line of the window. Otherwise, the previous routine is used to determine if the cursor has moved onto a new logical line. Then the row (235/$EB) is decremented, and screen line pointers are updated to reflect the change.

51317 $C875

Handles cursor left character, CHR$(157).

Calls the subroutine to move the cursor one position to the left [$CC00]. Upon return, the routine exits with the status register carry bit set if the cursor is already at the upper left corner of the window (the cursor will not be moved in this

case). Otherwise, the routine exits immediately with carry clear, unless the move wrapped the cursor to the right margin of the line above. In that case, a test of whether the cursor moved onto a new logical line must be performed, and the cursor row value (235/$EB) and screen line pointers must be updated.

51328 $C880

Handles switch-to-lowercase character, CHR$(14).

Sets bit 1 of the character base address shadow register (2604/$0A2C) to %1 if the 40-column screen is active. This value will be copied into the VIC-II chip register at 53272/$D018 during the next IRQ interrupt (see the screen IRQ routine [$C194]). If the bit has previously been cleared, this has the effect of adding 2048/$0800 to the character base address. When the default register value is being used (standard ROM-based characters), this selects the lowercase/uppercase character set at 55296/$D800. Everything currently displayed on the 40-column text screen will be affected. If the 80-column screen is active, the routine instead sets bit 7 of the current attribute Hag <241/$F1) to %1. This bit, which has no effect on 40-column printing, determines which character set will be used for any characters subsequently printed on the 80-column display using this attribute value. Characters already on the screen are not affected, so it's possible to mix character sets on the 80-column display.

51346 $C892

Handles switch-to-uppercase character, CHR$(142).

Clears bit 1 of the character base address shadow register (2604/$0A2C) to %0 if the 40-column screen is active. This value will be copied into the VIC-II register at 53272/$D018 during the next IRQ interrupt (see the screen IRQ routine [$C194]). If the bit has been previously set to %1, this has the effect of subtracting 2048/$0800 from the character base address. When the default register value is being used (standard ROM-based characters), this selects the uppercase/graphics character set at 53248/SD000. Everything currently displayed on the 40-column text screen will be affected. If the 80-column display is active, the routine instead clears bit 7 of the attribute

flag (241/$F1) to %0. This bit, which has no effect on 40-column printing, determines which character set will be used for any characters subsequently printed on the 80-column screen using this attribute value. Characters already on the screen are not affected, so it's possible to mix character sets on the 80-column display.

51366 $C8A6

Handles case switching disable character, CHR$(11).

Sets bit 7 of the case switching flag (247/$F7) to %1, This flag is checked during the keyscan routine [$C55D] to determine whether the character set should be changed when the SHIFT and Commodore keys are held down simultaneously. If bit 7 of the flag is %1, case switching is not allowed. Note that this disables case switching only via the SHIFT-Commodore key combination. There is no provision for preventing case switching by printing characters 14 or 142.

51372 $C8AC

Handles case switching enable character, CHR$(12).

Clears bit 7 of the case switching flag (247/$F7) to %0. This allows case switching via the SHIFT-Commodore key combination.

51379 $C8B3

Handles cursor home character, CHR$(19).

Checks the previous character code value (stored in 240/$F0). If that character is also 19/$13, the window is reset to full screen size before moving the cursor home. The routine at 49488/$C150 is called to set the cursor to the home position.

The special effect of HOME-HOME, resetting the window to full screen size, is a feature you must keep in mind if your programs use resized screen windows. If a program uses the BSOUT routine to display characters in the window, you should avoid printing the {HOME} character twice in a row. If the program accepts user input and displays it on the screen, you must guard against the chance of having your window boundaries reset. See the entry for the CTLVEC indirect vector (820-821/$334-$335) for information on disabling this feature.

51391 $C8BF

Handles reverse off character, CHR$(146).
Loads the accumulator with 0/$00, then uses a BIT opcode to fall through into the following routine and store the value in the reverse video flag (243/$F3).

51394 $C8C2

Handles reverse on character, CHR$(18),
Stores the value 128/$80 in the reverse video flag (243/$F3). As long as this flag contains a nonzero value, all characters printed to the screen using the BSOUT printing subroutine [$C320] will be displayed in reverse video.

51399 $C8C7

Handles underline on character, CHR$(2).
Sets bit 5 of the current attribute (241/$F1) to %1. This affects only the 80-column screen; the upper four bits of the value are meaningless for the 40-column screen. Any characters subsequently printed on the 80-column screen with this attribute will appear underlined.

51406 $C8CE

Handles underline off character, CHR$(130).
Clears bit 5 of the current attribute (241/$F1) to %0. This affects subsequent printing only. Any underlined characters already on the screen will remain underlined.

51413 8C8D5

Handles flash on character, CHR$(15).
Sets bit 4 of the current attribute (241/$F1) to %1. This affects only the 80-column screen; the upper four bits of the value are meaningless for the 40-column screen. Any characters subsequently printed on the 80-column screen with this attribute will flash at the same rate as cursor blinking.

51420 SC8DC

Handles flash **off** character, CHR$(143).
Clears bit 4 of the current attribute (241/$F1) to %0. This affects subsequent printing only. Any flashing characters already on the screen will continue to flash.

51427 $C8E3

Handles insert character, CHR$(148).
Stashes the current cursor position. The routine then moves to the last nonspace character in the logical line and copies all characters between there and the original cursor position one position to the right. It then places a space character at the original cursor position. The pending insert flag is incremented, then tested to see if it rolled over from 255/$FF to 0/$00. If so, the flag is reset to 255/$FF; so that value is the maximum number of pending inserts allowed. The cursor is restored to its original position upon exit.

When autoinsert mode is active, the screen printing subroutine [$C320] calls this routine before each character is printed to insert a space in which to print the character.

51483 $C91B

Handles delete character, CHRSUO).
Moves the cursor one position to the left, then checks to see whether the move wrapped the cursor to the right margin of a new logical line. If so, the routine exits, since no characters need be moved in this case. Next (at 51491/$C923), the routine enters a recursive loop with the routine at 51517/$C93D to copy all characters to the end of the logical line one position to the left, overwriting the character to the left of the original cursor position,

51506 $C932 RSTRPOS

Restores the cursor row and column positions.
Loads the cursor column position (236/SEC) from temporary storage in 222/$DE and the current cursor row (235/$EB) from 223/$DF, then jumps to the routine at 49500/$C15C to set pointers to the new cursor position. The corresponding routine to stash the cursor row and column values is at 52254/$CClE.

51517 $C93D DELCHAR

Deletes a character in a logical line.
Copies the character and attribute to the right of the current cursor position into the current position, then moves the cursor right and jumps back to the subroutine at 51491/$C923, which calls this routine recursively until the end of the logical line is reached.

51535 $C94F

Handles tab character, CHR$(9).

Checks whether the tab stop bit for the position to the right of the current cursor column is set to %1, or whether the position is beyond the right window margin. If neither, the routine continues checking columns to the right until either a tab stop is found or the right margin is reached. The cursor position is then reset to the column where the tab stop is found, or to the right margin if none is found. Thus, it's impossible to tab past the right edge of the window.

51553 $C961

Handles clear/set tab stop character, CHR$(24).

Toggles the bit in the tab stop bitmap (852-861 /$0354-$035D) corresponding to the current cursor position. If the bit is %0, it will be set to %1, setting a tab stop at the current position. If the bit is %1, it will be cleared to %0, clearing the tab stop previously set at the position.

51564 8C96C TESTTAB

Tests tab stop bit for current cursor position.

Calculates the byte offset and bit mask into the tab stop bitmap (851-861/$0354-$035D) for the cursor column specified in the Y register, then checks the corresponding bit position in the bitmap. If the bit is set to %1, indicating that a tab stop is set at the specified cursor column, the status register Z bit will be clear upon exit. If the bitmap bit is %0, the Z status bit will be set. In either case, the Y register will still contain the specified column value upon exit.

51584 $C980

Clears all tab stops (ESC Z).

Loads the accumulator with the value 0/$00, then uses a BIT instruction to fall through into the next routine and write the value to all ten bytes of the tab stop bitmap (852-861/ $0354-$035D). This eliminates any set bits in the bitmap, thus clearing all tab stops.

51587 $C983

Sets default tab stops (ESC Y).

Loads the accumulator with the value 128/$80, then uses a loop to write the value to all ten bytes of the tab stop bitmap (852-861/$0354-$035D). Since a bit set to %1 indicates a tab stop, filling the bitmap with this value (%10000000) has the effect of setting a tab stop every eighth character position.

51598 $C98E

Handles bell character, CHR$(7).

Checks the bell disable flag (249/$F9) and exits immediately if bit 7 is set to %1. If the tone is enabled, SID chip registers are set to produce a sawtooth waveform tone of approximately 750 hertz, using a moderately low volume setting (5). The duration of the tone is controlled by judicious selection of the ADSR envelope values. The sound is never turned off, but it quickly decays to an inaudible level.

51633 $C9B1

Handles linefeed character, CHR$(10).

Finds the position of the last character in the current logical line and moves the cursor to the row below that character position at the same column it previously occupied. If the logical line extends onto the last physical line in the window, a new blank line will be scrolled onto the bottom of the window, and the cursor will be moved onto the blank line (or, if scrolling is disabled, the cursor will wrap to the top line of the window). Remember that linefeed moves the cursor to the next logical line, not the next physical line (that's what cursor down does).

51646 $C9BE ESCAPE

Handles ESC sequences.

Jumps indirectly through the ESCVEC vector (824-825/$0338-$0339), which is initialized to point to the JESCAPE jump table entry [$C01E]. This in turn returns control to 51649/$C9C1, the address immediately following the indirect jump. Thus, the jump normally has no effect; however, the vector can be redirected to point to your own routine in RAM, allowing you to add your own ESC sequences or to modify the action's existing sequences. See the entry at ESCVEC for more information.

Next, the routine checks whether the character code (in the accumulator upon entry) is also ESC. If so, the current character is shifted right one bit. This changes the character from 27/$1B to 13/$0D, the RETURN character, so that dual ESCs are not read repeatedly. The routine then jumps to the routine to cancel quote mode. This is an undocumented feature of the ESC sequences: ESC ESC is a handy shortcut for ESC O.

If the character is not ESC, bit 7 is masked off. This means that shifted letters are treated the same as unshifted ones. ESC SHIFT-A has the same effect as ESC A, for example. (Remember, however, that only the alphabetic letter keys have this relationship. ESC SHIFT-@ is not the same as ESC @.)

Next, 64/$40 is subtracted from the character values. This will translate the character codes for @ and A-Z into an index in the range 0-26. If the result is outside this range, there is no standard ESC sequence for the specified character, so the routine exits without taking further action. If the character is within the valid range, the address of the subroutine to perform the corresponding sequence is loaded from the table at 51678/$C9DE. It is pushed onto the stack so that the subroutine will be called when the RTS opcode at the end of this routine is executed.

51678 $C9DE ESCTBL

Table of ESC key dispatch addresses.

Each two-byte entry in this table consists of the address minus 1 of the subroutine to perform the corresponding ESC key sequence. The routine to handle ESC sequences [$C9BE] pushes a table entry on the stack so that the RTS at the routine causes a jump to the subroutine. Each entry is one less than the actual address because of the way RTS behaves: When RTS pulls an address from the stack, the address value is incremented before being placed in the 8502's program counter. The actual execution addresses for each of the subroutines are as follows:

ESC @	51871/$CA9F	Clears to end of screen
ESC A	51949/$CAED	Enters autoinsert mode
ESC B	51734/$CA16	Sets bottom right corner of window
ESC C	51946/$CAEA	Cancels autoinsert mode
ESC D	51794/$CA52	Deletes an entire logical line

ESC E	51979/$CB0B	Sets nonblinking cursor mode
ESC F	52001/$CB21	Sets blinking cursor mode
ESC G	52023/$CB37	Enables bell tone for CHR$(7)
ESC H	52026/$CB3A	Disables bell tone for CHR$(7)
ESC I	51773/$CA3D	Inserts a blank screen line
ESC J	52145/$CBB1	Moves cursor to start of logical line
ESC K	52050/$CB52	Moves cursor to end of logical line
ESC L	51938/ICAE2	Enables screen scrolling
ESC M	51941/$CAE5	Disables screen scrolling
ESC N	52040/$CB48	Sets normal 80-column screen
ESC 0	51069/$C77D	Cancels quote mode
ESC P	51851/$CA8B	Erases to start of logical line
ESC Q	51830/$CA76	Erases to end of logical line
ESC R	52031/$CB3F	Reverses 80-column screen
ESC S	51954/$CAF2	Sets block cursor (80-column)
ESC T	51732/$CA14	Sets top left corner of window
ESC U	51966/SCAFE	Sets underline cursor (80-column)
ESC V	51900/$CABC	Scrolls screen up one line
ESC W	51914/$CACA	Scrolls screen down one line
ESC X	52524/$CD2C	Switches active displays
ESC Y	51587/$C983	Sets default tab stops
ESC Z	51584/$C980	Clears all tab stops

51732 $CA14 SETTOP

Defines the upper left corner of the window (ESC T>.

Clears the carry bit, then uses a BIT opcode to fall through into the following routine to load the current cursor position and set the window boundary.

51734 $CA16 SETBTM

Defines the lower right corner of the window (ESC B).

Sets the carry bit, loads current cursor position, then falls through into the following routine to set the window boundary.

51739 $CA1B WINDOW

Sets window boundaries.

(This routine has a jump table entry at 49197/$C02D.)

Establishes a window boundary according to the values in the accumulator and X register and the setting of the status register carry bit. If entered with carry clear, the accumulator value defines the new top row of the window, and the X register value defines the new left margin. If entered with carry set,

the accumulator value defines the new bottom row, and the X register value defines the new right margin. The newly defined window is not cleared, but the routine does fall through to the portion of the following routine that clears the line link bitmap, so all lines in the window will initially be unlinked.

When using this routine, remember that row and column numbering begins with zero (column 0 of row 0 is the upper left corner of the screen). Thus, SYS 51739,9,19,0,0 would set the upper left corner at the twentieth column over on the tenth line down. The routine does not check the validity of your entries; you are responsible for making sure that the lower right corner values are at least equal to the upper left corner values. If you set a lower right corner that is above or to the left of the upper left corner, the screen display will be garbled.

51748 $CA24 FULLW

Resets the window to full screen size.

Sets the bottom right corner of the window to the maximum row and column settings, held in 237/$ED and 238/SEE, respectively; then sets row 0, column 0 as the upper left corner. Finally, the routine clears all bytes in the line link bitmap, effectively unlinking all screen lines.

51773 $CA3D

Inserts a blank line (ESC I).

Copies all lines in the window starring at the row on which the cursor currently resides one row lower (the bottom row will be lost); then clears the row where the cursor resides and moves the cursor to the left margin of the cleared line. The link bit for the line below the new one (the line that originally occupied the new line's position) is tested. If the link bit is set to %1, the new line has been inserted within an existing logical line, so the link bit for the new line will also be set to add it to the logical chain. If the new line is not linked to a previous line, bit 7 of the input starting line value (232/$E8) is set to %1 to indicate that the cursor is at the start of a logical line.

51794 $CA52

Deletes the current logical line (ESC D).

Determines which row is the first of the current logical line, then scrolls all lines in the window beginning with the first

t

row of the next logical line upward to overwrite the current line (blank lines will be added at the bottom of the window). The cursor is then moved to the left margin of the original top row of the deleted logical line. Bit 7 of the input starting line value (232/SE8) is set to %1, indicating that the cursor is at the start of a logical line.

51830 $CA76

Erases to the end of **the** current logical line (ESC Q).

Clears to the end of the current row, then checks whether the next row is linked to the current one. If so, that line is cleared as well. This continues until the last row of the logical chain is reached. The cursor is restored to its original position upon exit.

51851 $CA8B

Erases to the start of the current logical line (ESC P).

Prints a space at the current cursor position, then checks whether the cursor is at the left margin of the first row in a logical chain. If not, the cursor is moved left and another space is printed, repeating until the start of the logical line is reached. The cursor is then restored to its original position.

51871 $CA9F

Erases to the end of the window (ESC @).

Clears to the right margin of the current row, then moves down to the next row and clears the remainder of the window one logical line at a time until the bottom of the window is reached. The cursor is then restored to its original position.

51900 $CABC

Scrolls the display up one **line** (ESC V).

Copies all lines in the window up one row (the top row will be lost) and clears the bottom row. The cursor is then restored to its original position,

51914 $CACA

Scrolls the display down one line (ESC W).

Copies all lines in the window down one row (the bottom row will be lost) and clears the top row. The cursor is then restored to its original position.

51938 $CAE2

Enables screen scrolling (ESC L).

Loads the accumulator with 0/$00, then uses a BIT opcode to fall through into the following routine and store this value in the scrolling enable flag (248/$F8).

51941 $CAE5

Disables screen scrolling (ESC M).

Stores the value 128/$80 in the scrolling enable flag (248/$F8), setting bit 7 to %1. As long as bit 7 of that flag is set, the printing routines will not scroll new lines onto the window, and the cursor will wrap at window margins.

51946 $CAEA

Cancels autoinsert mode (ESC C).

Loads the accumulator with 0/S00, then uses a BIT opcode to fall through into the following routine and store this value in the autoinsert enable flag (246/SF6).

51949 SCAED

Enables autoinsert mode (ESC A),

Stores the value 128/$80 in the autoinsert enable flag (246/$F6). As long as that flag contains a nonzero value, the printing routines will insert a space before each character is printed.

51954 $CAF2

Changes 80-column cursor to solid block (ESC S).

Checks the active screen flag, exiting immediately if the 80-column display is not active. If it's active, the routine clears bits 0-4 of the 80-column cursor flag (2603/$0A2B) to %00000, then copies the flag value to the 8563 chip cursor register (RIO). This causes the cursor to begin on raster line 0 of the screen line, making the cursor block the same height as the character patterns.

51966 $CAFE

Changes 80-column cursor to underline (ESC U).

Checks the active screen flag, exiting immediately if the 80-column display is not active. If it's active, the routine sets bits 0-4 of the 80-column cursor flag (2603/$0A2B) to %00111,

then copies the flag value to the 8563 chip cursor register (RIO). This causes the cursor to begin on raster line 7 of the screen line, the scan line below the character patterns.

51979 $CB0B

Disables cursor blinking (ESC E).

Begins by checking which display is currently active. If the 80-column display is active, bits 5 and 6 of the 80-column cursor flag (2603/$0A2B) are cleared to %0. Then the flag value is copied to the 8563 chip cursor register (RIO). This halts the blinking of the cursor on the 80-column screen. If the 40-column display is active, bit 6 of the 40-column cursor flag (2598/$0A26) is set to %1 to disable cursor blinking.

52001 $CB21

Enables cursor blinking (ESC F).

Begins by checking which display is currently active. If the 80-column display is active, bits 5 and 6 of the 80-column cursor flag (2603/$0A2B) are set to %1. Then the flag value is copied to the 8563 chip cursor register (RIO). This causes the cursor on the 80-column screen to blink. If the 40-column display is active, bit 6 of the 40-column cursor flag (2598/$0A26) is cleared to %0 to enable cursor blinking.

52023 $CB37

Enables tone for bell character (ESC G).

Loads the accumulator with 0/$00, then uses a BIT opcode to fall through into the following routine and store this value in the bell disable flag (249/$F9).

52026 8CB3A

Disables tone for bell character (ESC H).

Stores the value 128/$80 in the bell disable flag (249/$F9), setting bit 7 to %1. As long as bit 7 of that flag is set to %1, no tone will be sounded when character code 7 is printed.

52031 $CB3F

Switches 80-column screen to reverse mode (ESC R).

Sets bit 6 in 8563 register 24 (R24) to %1. This sets the 80-column screen to reverse mode: Characters appear in the background color specified in the lower four bits of R26, and the

screen background for each character position takes the color specified in the corresponding attribute memory position.

52040 $CB48

Switches 80-column screen to normal mode (ESC N).

Clears bit 6 in 8563 register 24 (R24) to %0. This sets the 80-column screen to normal mode: Characters appear with the attribute specified in the corresponding attribute memory position, and the screen background takes the background color specified in the lower four bits of R26.

52050 $CB52

Moves the cursor past the last character on the current logical line (ESC K).

Finds the position of the last nonspace character in a logical line, then sets the cursor pointers to the column to the right of that position. If the existing logical line completely fills its last physical line, a new blank line will be inserted (scrolling all lines below the current one down one line), and the cursor will move to the left margin of the new line. However, if the logical line in question completely fills the last physical line in the window, all lines in the window will instead be scrolled up one line to open a new blank line at the bottom, unless scrolling is disabled. In this case, the cursor will simply be moved to the bottom right corner of the window.

52056 $CB58 READCHR

Reads character and attribute at current cursor position.

Stores the attribute at the current cursor position in 242/$F2 and returns the screen code at the current cursor position in the accumulator.

52084 $CB74 TESTLINK

Tests whether a line is linked.

Checks the bit in the line link bitmap (862-865/$035E-$0361) corresponding to the line specified in the X register. If the line is linked to the one above, the carry bit will be set upon exit; if not, it will be clear. The line number is preserved in the X register upon exit.

52097 SCB81 SETLINK

Links or unlinks the current screen line.

Loads the X register with the current row number (235/$EB), then enters one of the two following routines depending on the status of the carry bit. If carry is set, a branch is taken to 52115/SCB93 to link the current screen line to the one above. If carry is clear, this routine falls through to the next one to unlink the current line.

52101 $CB85 UNLINK

Unlinks a screen line.

Clears the bit in the line link bitmap corresponding to the line specified in the X register. The line number is preserved in the X register upon exit,

52115 $CB93 LINK

Links a screen line.

Sets the bit in the line link bitmap corresponding to the line specified in the X register. The line number is preserved in the X register upon exit.

52127 $CB9F FINDLINK

Calculates offsets into the line link bitmap.

Calculates the position of the bit in the line link bitmap corresponding to the line specified in the X register upon entry, returning the byte offset into the line link bitmap in the X register and the mask for the corresponding link bit within that byte in the accumulator.

52145 $CBB1

Moves the cursor to the start of logical line (ESC J).

Sets all cursor position pointers to the left margin in the first row of the current logical line.

52163 $CBC3 FINDEND

Finds the position of the last character in a line.

Calculates the position of the last nonspace character in the current logical line and returns the column value of that position in 234/$EA and the row value in 235/$EB.

52205 8CBED FORWARD

Moves the cursor one position to the right.

Checks whether the move would place the cursor beyond the right margin of the window. If not, the new position value is set in 236/$EC. If the cursor is already at the right window margin, the subroutine at 50019/$C363 is called to move the cursor to the left margin of the next line, scrolling the window if the cursor is on the bottom line (or wrapping the cursor to the top of the window if scrolling is not allowed). Upon exit, the carry bit will be set if the move caused the screen to scroll (or the cursor to wrap). The accumulator contents are preserved unchanged during this routine; the cursor column will be in the Y register upon exit.

52224 $CC00 RETREAT

Moves the cursor one position to the left.

Checks whether the move would place the cursor beyond the left margin of the window. If not the new position value is set in 236/$EC, and the carry bit is cleared. If the cursor is already at the left window margin, the routine checks whether the cursor is on the top row of the window. If so—if the cursor is currently in the home position of the window—the routine exits with the carry bit set and without moving the cursor. Otherwise, the cursor position is set to the right margin of the screen line above the current one. In this case, the Z bit in the status register will be set upon exit. The accumulator contents are preserved unchanged during this routine; the cursor column will be in the Y register upon exit.

52254 **$CC1E SAVEPOS**

Stores the cursor position for later restoration.

Stashes the current cursor column value in temporary storage at 222/SDE and the current cursor row value at 223/$DF. The corresponding routine to retrieve these values and restore the cursor to the saved position is at 51506/$C932.

52263 $CC27 SPACE

Prints a space.

Loads the X register with the current character color from the attribute flag (241/$F1), masking out bits 4-6, and loads the accumulator with 32, the screen code for a space. BIT opcodes

then allow the routine to fall through to 52276/$CC34 to display the space.

52271 $CC2F DISPLY1

Displays a character using the current attribute.

Loads the X register with the current character color from the attribute flag (241/SF1), then uses a **BIT** opcode to allow the routine to fall through to 52276/$CC34 to display the screen code in the accumulator.

52274 $CC32 **DISPLY2**

Displays a character using the previous attribute.

Loads the X register with the current character color from the previous attribute flag (242/$F2), then falls through to display the screen code in the accumulator.

52276 $CC34 DISPLY

Displays a character at the current cursor position.
{This routine has a jump table entry at 49155/$C003.)

Places a character on the screen by putting the screen code value (in the accumulator) into the screen memory location corresponding to the current cursor position. The attribute value (in the X register) is then stored into the attribute memory location corresponding to the current cursor position. (Note that the accumulator should contain a screen code, not a character code.) The screen memory address is determined by using the current cursor column (236/$EC) as an offset from the first screen memory position for the screen line (224-225/$E0-$El), and the attribute memory address is determined by using the current column value as an offset from the first attribute memory position for the screen line (226-227/$E2-$E3). For the 40-column screen, the screen code and attribute values are stored directly into the memory locations. For the 80-column screen, the values must be stored in the 8563 chip's private 16K of RAM indirectly, via the 8563 registers.

52315 $CC5B SCRORG

Returns height and width of current screen window.
(This routine has a screen editor jump table entry at 49167/$C00F and a Kernal jump table entry at 65517/$FFED.)

Calculates the screen height by subtracting the top margin (229/$E5) from the bottom margin (228/$E4); this value is returned in the Y register. The screen width is calculated by subtracting the left margin (230/$E6) from the right margin (231/$E7); this value is returned in the X register. The accumulator is loaded with the maximum column number for the screen in use (238/$EE)—39/$27 for the 40-column screen or 79/$4F for the 80-column screen.

Note that the height and width values calculated by this routine will be one less that the actual number of rows and columns in the window. For example, with the 40-column screen set for a full-size window (40 columns X 25 rows), this routine will return X and Y register values of 39 and 24, respectively. To return the proper values, the routine should have added 1 to the subtraction results. By comparison, this routine in Commodore 64 ROM, where is it called SCREEN, returns the values 40 and 25 for the standard screen (also 40 columns X 25 rows).

52330 $CC6A PLOT

Reads or sets the current cursor position.
(This routine has a screen editOT jump table entry at 49176/$C018 and a Kernal jump table entry at 6552O/$FFF0.)

Returns the row and column numbers corresponding to the current cursor position or establishes new row and column values, depending on the status of the carry bit upon entry. To set the cursor position, enter the routine with the desired row number (0-24) in the X register, the desired column number (0-39 for the 40-column screen or 0-79 for the 80-column screen) in the Y register, and the carry bit clear. To read the cursor position, enter the routine with the carry bit set; the current row number will be in the X register upon return, and the current column number will be in the Y register (and also in the accumulator).

It's important to remember that coordinate numbering begins with 0, not 1. Thus, setting the cursor position with X and Y containing 5 and 3, respectively, will place the cursor on the sixth row down at the fourth column across. When reading coordinates, remember that the cursor will actually be one position beyond the last character printed. The coordinates you set or read are relative to the home position and left margin of the current window, not the absolute home position and left edge

of the screen display. This makes no difference as long as the window is set for full screen size, but it is significant when you are using a reduced-size window. For example, if the cursor is on the eighth row down from the top of the screen and the tenth column across from the left edge of the screen, the row and column values returned by this routine will be 7 and 9, respectively, if the window is full screen size. However, if the top left corner of the window is moved to the sixth column of the sixth row, the row and column values returned for the same cursor position will be 1 and 3, respectively.

When setting the cursor position, the routine first checks whether the specified position would be beyond either the right or bottom margin of the window. If the position would be outside the window, the routine exits with the carry bit set and without changing the current cursor position. Thus, you can check the carry bit after calling this routine to determine whether the cursor was successfully moved (indicated by a clear carry bit).

This routine has an idiosyncrasy that will trip you up if you're not careful. Keep in mind that the value in the X register contains the vertical (row) position, and the value in the Y register contains the horizontal (column) position. This is opposite from the way you normally think of *x* and *y* coordinates in geometry: *x* is usually horizontal and *y* vertical. Some other routines, such as SCRORG [$CC5B], use X to hold the horizontal value and Y for the vertical value, so don't get confused.

52386 $CCA2 KEYSET

Defines a programmable function key.
(This routine has a screen editor jump table entry at 49185/$CO21 and a Kemal jump table entry at 65381/$FF65.)
Replaces the existing definition string for one of the ten programmable keys, F1-F8, SHIFT-RUN/STOP, and HELP, with a new string. The routine first converts the key number (1-10), in the X register upon entry, to a key index (0-9), which is stored in 220/SDC. The length of the definition string should be in the Y register upon entry, and the address of the zero-page pointer to the definition string should be in the accumulator upon entry. The number of the memory bank where the definition string resides is read from the zero-page location immediately following the two-byte pointer to the string. If the

length of the new definition string is the same as the length of the existing definition string for the specified key, no special handling is required; the new definition string is simply copied over the old one. If the new definition is shorter, all the definitions above the existing one are moved down. If the new definition is longer, all characters must be moved upward. Before anything is actually moved, the routine checks to make sure that adding the extra characters will not exceed the 246 bytes available for string definitions (4106-4351/$100A-$10FF). If the new definition will not fit, the routine exits with the status register carry bit set and without changing the existing definition string.

To add a new definition string, the length of the string is placed in the proper position in the string length table in bank 0 (4096-4105/$1000-$1009). Then characters are loaded from the new string in whatever bank it is located in by using the INDFET routine [$02A2]. They are stored at the proper position in the definition string table (4106-4351/$100A-$10FF). The status register carry bit is cleared before exit to indicate that the new definition string has been successfully added to the table.

52512 8CD20

Calculates the offset to the start of key definition string. Adds the length table entries for all key definition strings with an index lower than the one specified in the X register upon entry. (X should contain a key index, 0-9, not a key number, 1—10.) The total will be in the accumulator upon exit, and carry will be set. Because there are no separator characters between definition strings in the table, the only way to determine the starting position of a particular string is to add the lengths of all the preceding strings. This implies that an incorrect length value in the string length table (4096-4105/$1000-$1009) will result in incorrect strings being returned for all keys with a higher index. Thus, you should use caution when changing string length tables directly.

52524 $CD2C

Changes screen displays (ESC X).
Stores the current character code in 240/$F0, then falls through into the next routine to switch active displays.

52526 $CD2E SWAPPER

Switches active screen displays.
(This routine has a screen editor jump table entry at 49194/SC02A and a Kernal jump table entry at 65375/SFF5R)
Swaps the active and inactive screen variables by exchanging the contents of 224-250/$E0-$FA with the contents of 2624-2650/$0A40-$0A5A. (This duplicates the bug in the initialization routine [$C07B] which copies 27 values, when only 26 are actually valid.) The routine then swaps the active and inactive tab stop and line link bitmaps by exchanging the contents of 852-865/$0354-$0361 with the contents of 2656-2669/$0A60-$0A6D. Finally, bit 7 of the active screen flag (215/$D7) is toggled, switching the active and inactive screen displays.

Note that this doesn't physically turn either video chip on or off. Both video sources remain on at all times. The active display is merely the one to which all printing is currently directed.

52567 $CD57 CRSR80

Sets cursor position on 80-cohimn screen.
(This routine has a jump table entry at 49179/$C01B.)

Checks active screen flag and exits immediately if the 40-column screen is active. The screen memory address for the cursor position is calculated by adding the current cursor column (236/$EC) to the starting address for the screen line (224-225/$E0-$El), This address is then written to the 8563 chip's cursor position registers (R14-R15) using the routine at 52684/$CDCC

This routine is normally called by the BSOUT exit routine [$C30E] after a character has been printed. The routine is necessary because the 80-column cursor is managed in hardware, not software. So, unlike the 40-column display's cursor, it doesn't move automatically when cursor row and column pointers are changed.

52591 $CD6F CRSRON

Turns cursor on.

Begins by checking the active screen flag. If the 40-column display is active, the routine merely clears the cursor enable flag (2599/$0A27) and exits. For the 80-column display, the attribute of the current cursor position is read and stored in 2611/$0A33. The upper four bits (containing the printing style information) are stored in 219/$DB. Then this style information is combined with the current cursor color (the lower four bits of 241/$F1) to determine the new attribute for the cursor position. Finally, the cursor is enabled by writing the current cursor style (stored in 2603/$0A2B) to the 8563 chip's cursor control register (RIO).

52639 $CD9F CRSROFF

Turns cursor off.

Begins by checking the active screen flag. If the 80-column display is active, this routine sets the update location registers (R18-R19) to the address of the attribute for the current cursor position, then restores the cursor position to its original attribute (stored in 2611/S0A33), Finally, the cursor is disabled by writing the value 32/$20 to the 8563 chip's cursor control register (RIO). For the 40-column display, the cursor is disabled by clearing the cursor enable flag (2599/$0A27). Then the blink phase flag (2598/$0A26) is checked to see whether the character under the cursor is in the normal or reversed portion of the blink sequence. If the character is in its normal state, no further action is necessary. However, if the character is reversed, the blink phase flag is reset and the cursor position is restored to its original character and color (stored in 2601/S0A29 and 2602/$0A2A, respectively).

52682 $CDCA WRITE80

Writes a byte value to 80-column chip memory.

Stores the value in the accumulator into a location in the 8563 chip's private 16K of RAM. The target memory address is specified by the current contents of the 8563's update location registers (R18-R19). The contents of the X register will be changed, but the contents of the accumulator and Y register are unaffected. The address in R18-R19 is automatically incremented after a byte is written, so it is not necessary to update those registers before every byte when writing to a series of

sequential addresses. The routine works by loading X with 31/$1F, the number of the 8563's data read/write register. Then it falls through into the following routine to store the accumulator contents in that register. Writing to register 31 causes the value written to be stored in the 80-column RAM location addressed in registers 18-19.

The following example shows how to use this routine to copy the contents of the first 256 bytes of 40-column screen memory (locations 1024-1279/$0400-$04FF) to 80-column screen memory:

```
0B00  LDX  #$12      ; Load registers 18 and 19
0B02  LDA  #$00      ;    with low and high bytes of
OB04  JSR  $CDCC     ;    80-column screen starting
0B07  INX            ;    address ($0000).
0B08  JSR  $CDCC
0B0B  LDY  #$00      ; Initialize loop offset.
0B0D  LDA  $0400,Y   ; Read byte from 40-column screen,
0B10  JSR  $CDCA     ; Store in 80-column RAM.
0B13  INY
0B14  BNE  $0B0D     ; Loop for 256 bytes.
0B16  RTS
```

52684 $CDCC WRITEREG

Writes to an 80-column chip register.

Stores the accumulator contents in the 8563 register specified in the X register (see Chapter 8 for a description of the 8563's registers). The accumulator and X register contents will not be changed, and the Y register is unaffected. No error checking is performed; when you use this routine you are responsible for making sure that X contains a valid register number (0-36/$00-$24). This routine illustrates the proper procedure for writing to an 8563 register: The desired register number is stored in 54784/$D600. Then the routine waits until bit 7 of that location is set to %1, after which the new register value is stored in location 54785/$D601.

52696 $CDD8 READ80

Reads a byte value from 80-column chip memory.

Reads the contents of a location in the 8563 chip's private 16K of RAM. The target memory address is specified by the current contents of the 8563's update location registers (R18-R19). The location's contents will be in the accumulator upon exit

from this routine. The contents of the X register will be changed, but the Y register is unaffected. The address in R18-R19 is automatically incremented after a byte is read, so it is not necessary to update those registers before every byte when reading a series of sequential addresses. The routine works by loading X with 31/$IF, the number of the 8563's data read/write register. Then it falls through into the following routine to load the accumulator with the contents of that register. Reading from register 31 returns the contents of the 80-column RAM location addressed in registers 18-19.

The following example shows how to use this routine to copy the contents of the first 256 bytes of 80-column screen memory to 40-column screen memory locations 1024-1279/ $0400-$04FF:

```
0B00  LDX  #$12      Load registers 18 and 19
0B02  LDA  #$00        with low and high bytes of
0B04  JSR  $CDCC       80-column screen starting
0B07  INX              address ($0000).
0B08  JSR  $CDCC
0B0B  LDY  #$00      Initialize loop offset.
0B0D  JSR  $CDD8     Read a byte from 80-column RAM.
0B1O  STA  $0400,Y   Store in 40-column screen memory.
0B13  INY
0B14  BNE  $0B0D     Loop for 256 bytes.
0B16  RTS
```

52698 $CDDA READREG

Reads from an 80-column chip register.

Reads the current contents of the 8563 register specified in the X register. The register value will be in the accumulator upon exit from this routine. See Chapter 8 for a description of the 8563's registers. The X register contents will not be changed, and the Y register is unaffected. No error checking is performed; when you use this routine you are responsible for making sure that X contains a valid register number (0-36/ $00-$24). This routine illustrates the proper procedure for reading an 8563 register: The desired register number is stored in 54784/SD600. Then the routine waits until bit 7 of that location is set to %1, after which the register value can be read from location 54785/$D601.

52710 $CDE6 SCNPOS

Sets the current address in 80-column screen memory. Loads the 8563 update location registers (R18-R19) with the address of the screen memory location that corresponds to the current cursor position. Upon entry, the Y register should contain the offset to the current column. To calculate the memory address, this offset is added to the address of the first screen memory location for the current screen line (224-225/ $E0-$El). The registers are loaded by using the subroutine at 52684/$CDCC. Once the address is loaded into R18-R19, the next value placed in the read/write register (R31) will be stored in the specified screen memory location.

52729 $CDF9 ATTRPOS

Sets the current address in 80-column attribute memory. Loads the 8563 update location registers (R18-R19) with the address of the attribute memory location that corresponds to the current cursor position. Upon entry, the Y register should contain the offset to the current column. To calculate the memory address, this offset is added to the address of the first attribute memory location for the current screen line (226-227/$E2-$E3), The registers are loaded by using the routine at 52684/$CDCC. Once the address is loaded into R18-R19, the next value placed in the read/write register (R31) will be stored in the attribute memory location.

52748 SCEOC INIT80

Initializes character definitions for 80-column screen. (This routine has a screen editor jump table entry at 49191/SC027 and a Kernal jump table entry at 65378/$FF62.)

Copies the contents of the character generator ROM (at 53248-57343/$D000-$DFFF in bank 14) to the character definition area at 8l92-16383/$2000-$3FFF in the 80-column video chip's private block of RAM (which is not part of the 8502 microprocessor's address space). This is necessary because the 8563 chip has no character ROM of its own. Because the 8563 uses 16-byte character definitions, each 8-byre character definition from ROM is padded with eight zeros when copied to 8563 RAM. See Chapter 8 for information on how the 80-column characters can be redefined.

In the 128's ROM, this routine is called only by the Kernal IOIN1T routine [$E109], which is part of both the reset

and RUN/STOP-RESTORE sequences. However, the call is preceded by a test of the initialization status flag (2564/$0A04). If bit 7 of that flag is set to %1, the step to download 80-column character definitions is skipped. The reset routine [$E000] clears the status flag before calling this routine, so character definitions are initialized when the 128 is reset (including when it is first turned on). Then it sets bit 7 afterward to indicate that this step has been performed. The RUN/STOP-RESTORE sequence [$FA53] does not affect the initialization status flag, so the character definitions will not usually be recopied when IOINIT is called during that sequence. Thus, if you redefine (or, worse, accidentally garble) the character definitions in the 8563's RAM, RUN/STOP-RESTORE won't return them to normal as it does for the 40-column screen. You can restore the normal 80-column character shapes by pressing the RESET button or, less dramatically, by calling this routine (with SYS 52748, for example). You can clear bit 7 of 2564/$0A04 to %0 and **call** IOINIT.

52812 $CE4C COLORTBL

Table of color character translation values.

There is no direct mathematical relationship between the character codes which change printing colors and the actual VIC-II chip color numbers for the selected hues, so this table is used to translate the character codes into values for VIC-II chip color settings. For example, the character code which, when printed, changes character color to dark blue is 31/$ IF. This value is found at an offset of 6 into the table, and 6 is the VIC-II's color number for dark blue.

Table Entry (character code)	Table Offset (color number)	Color
144/$90	0/$00	black
5/$05	l/$01	white
28/$1C	2/$02	red
159/$9F	3/$03	cyan
156/$9C	4/$04	purple
3O/$1E	5/$05	green
31/$1F	6/$06	blue
158/$9E	7/$07	yellow
129/$81	8/$08	orange
149/$95	9/$09	brown
150/$96	10/$OA	light red
151/197	ll/$0B	dark gray

Table Entry (character code)	Table Offset (color number)	Color
152/$98	12/$0C	medium gray
153/$99	13/$0D	light green
154/$9A	14/$0E	light blue
155/$9B	15/$0F	light gray

52828 $CE5C COLOR80

Table of 8563 color code translation values.

The character codes which change printing colors are initially translated into color numbers for the 40-column (VIC-II) chip. The 8563 80-column video chip can produce most of the same colors as the 40-column chip, but its color memory and color registers require different color numbers. This table is used to translate VIC-II color values into 8563 color values. For example, storing 6 in the VIC-II chip register at 53281/SD021 changes the 40-column background to dark blue, but storing 6 in the 8563's background register (the lower nybble of R26) will result in a dark cyan 80-column background. The proper 80-column chip color number can be found by using the VIC-II color number as an offset into this table. For example, the 80-column color value for dark blue, 2, is found at an offset of 6 from 52828/$CE5C,

Offset (VIC-II color number)	8563 Chip (color number)	Color
0/$00	0/$00	black
l/$01	15/$0F	white
2/$02	8/$08	dark red
3/$03	7/$07	light cyan
4/$04	ll/$0B	light purple
5/$05	4/$04	dark green
6/$06	2/$02	dark blue
7/$ 07	13/$0D	light yellow
8/$08	10/$0A	dark purple
9/$09	12/$0C	dark yellow
10/$0A	9/$09	light red
ll/$0B	6/$06	dark cyan
12/$0C	1/S01	dark gray (light black)
13/$0D	5/$05	light green
14/$0E	3/$03	light blue
15/$0F	14/$0E	light gray (dark white)

As an example of how you would use this table to translate color numbers, suppose you had a 40-column color number in the accumulator and needed to know the equivalent 80-column color number. All that's needed is

```
TAX
LDA $CE5C,X
```

52844 $CE6C MASKTBL

Table of bit mask values.

Each of the eight bytes in this table has only one bit set to %1. The mask values are used to decode bitmapped tables such as the ones for tab stops and line links. The binary equivalents of the table bytes, in order, are as follows:

%10000000
%01000000
%00100000
%00010000
%00001000
%00000100
%0 0000010
%00000001

52852 $CE74 VARTBL

Tables of default screen editor variables.

Locations 52852-52877/$CE74-$CE8D hold the default screen editor variable settings for the 40-column screen. Locations 52878-52903/$CE8E-$CEA7 hold the default settings for the 80-column screen. The screen editor initialization routine [$C07B] copies the values for the default screen—determined by the position of the keyboard 40/80 DISPLAY key—into the active screen variable table at 224-249/$E0-$F9, and the values for the other screen into the inactive screen table at 2624-2649/$0A40-$0A59. Thus, the values in these tables determine the variable settings after power-on, reset, or RUN/STOP-RESTORE.

Actually, due to a bug in the initialization routine, one byte too many is copied when the tables are transferred to RAM. Thus, the byte following the active screen default value table will be copied to 250/SFA, and the byte following the inactive screen default table will be copied to 2650/$0A5A.

52904 $CEA8 KEYDEFS

Table of standard function key definitions.

The first ten bytes here hold the lengths of the following ten key definition strings. The screen editor initialization routine [SC07B] copies these lengths and strings to the definition area at 4096/$1000. The text for the standard definitions is as follows:

Key	**Definition**
Fl	GRAPHIC
F2	DLOAD"
F3	DIRECTORY {RETURN}
F4	SCNCLRjRETURN}
F5	DSAVE"
F6	RUN {RETURN}
F7	LIST{RETURN}
F8	MONITOR {RETURN}
SHIFT-RUN/STOP	DL"*{RETURN}RUN{RETURN}
HELP	HELP {RETURN}

If you ever need to restore the standard key definitions, you can recopy the contents of this table into the definition area. In BASIC, the required program line would have this form:

```
BANK 15: FOR 1=0 TO 76: POKE 4096+I,PEEK(52904+I): NEXT I
```

52981-53247 $CEF5-$CFFF Unused

All locations in this unused area of ROM are filled with the value 255/$FF, except for the final two, which are $00 $C3. Ideally, the keyboard decoding tables would have been placed here following the screen editor routines so that the screen editor package could be fully self-contained in this block of ROM. However, there's not sufficient room here for the five 89-byte keyboard tables, so they were placed at the end of Kernal ROM (64128-64572/$FA80-$FC3C).

I/O Chip Registers, Color RAM, and Character ROM

Bits 0 and 4-5 of the MMU configuration register (65280/ $FF00) determine what is visible in the area between addresses 53248-57343/$DO00-$DFFF of the 128's memory space. If bit 0 of the configuration register is %0, the collection of hardware registers and 40-column color RAM known as the I/O block is visible, regardless of the setting of bits 4 and 5 of the register. The I/O block includes the following elements:

53248-53296/$D00O-$D030	VIC (40-column) video chip registers
54272-54300/$D400-$D41C	SID sound chip registers
54528-54539/$D500-$D50B	MMU chip registers
54784-54785/$D600-$D601	VDC {80-cohimn) video chip registers
55296-56319/$D800-$DBFF	Color RAM for VIC chip
56320-56335/$DC00-$DC0F	CIA chip registers (CIA #1)
56576-56591/$DD00-$DD0F	CIA chip registers (CIA #2)
57088-57098/$DF0O-$DFOA	REC expansion controller chip registers (if memory-expansion module connected)

If bit 0 of the configuration register is set to %1, the setting of bits 4 and 5 will determine what is visible in this area. (Actually, these bits control what is seen in the entire area from 49152-65535/$C000-$FFFF.) The four possible selections for 53248-57343/$D000-$DFFF are as follows:

Bits		Block contents
5	**4**	
0	**0**	Character ROM
0	**1**	Internal function ROM
1	**0**	External function ROM
1	**1**	RAM

Character ROM contains the bit patterns to define the shape of the letters, numbers, and symbols for the video displays. When RAM is selected, the RAM block from which the memory is seen is determined by the setting of bits 6 and 7 of the

configuration register. The I/O block is visible in all standard bank configurations except banks 0-3, which are all RAM, and bank 14, where character ROM is seen.

A bit of 128 hardware trivia: Since the MMU configuration register setting in bits 4-5 for character ROM is also the one which selects screen editor ROM at 49152-53247/$C000-$CFFF and Kemal ROM at 57344-65535/$E000-$FFFF, you might expect that character patterns are stored between the screen editor and Kernal routines in the 16K ROM chip designated U35. However, this is not the case. Character patterns are stored in a separate 4K ROM chip, designated U18. So what's between 53248-57343/$D000-$DFFF in the ROM chip containing the screen editor and Kernal? That's where the ROM routines for CP/M mode are stored. The memory-shuffling capabilities of the MMU and PLA allow this area of ROM to appear at addresses 0-4095/$0000-$0FFF when CP/M mode is selected. Because this section of ROM is always invisible to 128 mode, it is not covered further in this book.

VIC (Video Interface Controller) Chip

53248-53296/$D000-$D030

The chip whose registers appear in this area of the I/O block is referred to in Commodore literature as the VIC, even though it is not exactly the same as the chip with that designation in the Commodore 64. However, the chip does provide the same 40-column video display features as its Commodore 64 predecessor. The differences are in the chip's less familiar, but equally vital function of providing all the basic riming signals required by the system. The 128's version of the VIC chip also supports the scanning of the additional 24 keys on the 128's keyboard. For these new features, the 128 version of the VIC has two more registers than its Commodore 64 counterpart (49 instead of 47). There are actually two different versions of the 128 VIC chip, depending on the video system required in the country where the computer is sold. For NTSC (North American) video, the version is officially designated the 8564 chip, while the PAL (European) model is designated the 8566. All the registers described below operate the same on both chips; only the video signal format is different.

Video Fundamentals

The output signals from the VIC chip tell the video device (monitor or television) how to draw the screen display. To understand the operation of the VIC chip, you need to understand a few of the fundamentals of video displays. The display is drawn on the monitor or television picture tube by a "gun" that shoots a beam of electrons at the screen. Where the beam strikes the face of the screen, a spot on the screen's phosphorescent coating glows briefly. The electron gun doesn't just spray electrons at random; the beam is moved in a precisely controlled pattern. Beginning in the upper left corner of the screen, the beam is scanned (moved) horizontally across to the right edge of the screen, drawing a very thin line of dots. It is then blanked while it is moved back to the left edge, but just below the top line. The beam is then scanned horizontally across the screen again, and the process is repeated until the stack of thin lines fills the screen display.

Actually, for a color display there are three separate beams working in conjunction to draw each line—one each for the colors red, green, and blue. Each dot in the thin screen line consists of red, green, and blue points. When the relative intensities of the red, green, and blue points are varied, the dot can take on a variety of hues. The VIC chip can produce 16 different colors. Whenever a memory location or VIC register calls for a color value, the color is specified by a value in the range 0-15. Table 8-1 lists the standard designations for the VIC chip colors.

Table 8-1. Standard VIC Color Values

Value	Color	Value	Color
0/$00	black	8/$08	orange
l/$01	white	9/$09	brown
2/$02	red	10/$OA	light red
3/$03	cyan	ll/$0B	dark gray
4/$04	purple	12/$0C	medium gray
5/$05	green	13/$0D	light green
6/$06	blue	14/$0E	light blue
7/$07	yellow	15/$0F	light gray

The stack of horizontal video lines is called the raster (from the Latin word for rake—the pattern of evenly spaced parallel lines is similar to that produced by pulling a rake through soil). The individual lines are called raster scan lines.

The number of lines required for a full screen depends on the video system in use. The North American (NTSC) version of the VIC chip draws a raster of 264 scan lines, while the European (PAL) version draws 313. Since the screen phosphor glows only briefly when struck by the raster beam, the screen must be constantly redrawn. The rate of redrawing also depends on the video system: 60 times per second for NTSC systems or 50 times per second for PAL systems. Not all of these raster lines are used for the active video display. Most televisions and monitors overscan. That is, some raster lines at the top, bottom, or both are actually drawn off the screen. The VIC compensates by restricting the active portion of the display, the area where characters and graphics can be displayed, to 200 lines in the middle of the raster for both NTSC and PAL systems. (This can be reduced to 192 lines. For details, see the entry for 53265/$D011.) The inactive lines form the top and bottom portions of the border, a solid-color frame around the active screen.

The horizontal dots that make up each scan line are called pixels (short for picture elements). The number of pixels in a scan line depends on the screen mode, and is limited by the speed at which the VIC chip can read data from memory. In the standard two-color modes, where a single bit determines the pixel color, the VIC chip draws 320 active pixels per scan line. In the four-color (multicolor) modes, two bits are required to specify the color of each pixel. Since the VIC must read twice as much data per pixel, only half as many pixels can be drawn in the time allotted for a single scan line. As a result, the multicolor modes have only 160 active pixels per scan line. The display is still the same size; the pixels are twice as wide. {The screen width can also be reduced to 304 standard pixels or 152 multicolor pixels. For details, see the entry for 53270/$D016.) Just as the VIC draws extra lines above and below the active ones, it also draws extra pixels to the left and right of the active ones. The inactive pixels form the sides of the solid-color border.

The VIC chip supports two major classes of display modes—character and bitmapped. These are also referred to as low resolution and high resolution, but that's somewhat misleading, since both provide the same active screen areas—320 pixels X 200 lines for standard modes or 160 pixels X 200 lines for multicolor modes. The difference between the classes

is in the degree of control over individual pixels. The bitmapped (high-resolution) modes allow you to determine the color of each pixel individually, while the character (low-resolution) modes only allow control of groups of pixels. The tradeoff is that the character display modes use much less memory.

Video Banks

Before you read a discussion of the display modes, it's important to understand how the VIC chip sees memory. The VIC uses the same RAM as the 8502 microprocessor, but it views the memory very differently. The VIC chip has only 14 address lines compared to the processor's 16. This means that the VIC can, at any given time, address only 16K (16384 bytes) out of the 64K of RAM in a block. The 16K area seen by the VIC chip is referred to as a video bank, not to be confused with one of the processor's bank configurations—there is no relationship. All of the information for the VIC screen display must be visible within the same video bank. There are four possible video banks per 64K block, or a total of eight possible video banks in the two RAM blocks in the 128. Bits 0-1 of the CIA #2 register at 56576/$DD00 select one of the four banks, and bit 6 of the MMU register at 54534/$D506 selects which 64K-RAM block the video bank will be seen in. Refer to the entries for those locations later in this chapter for more details. The base (starting) addresses for the banks are as follows:

Bank	Base address
0	0/$0000
1	16384/S4000
2	32768/$8000
3	49152/$C000

Character Display Modes

The VIC provides three character display modes: standard, multicolor, and extended background color. The standard character display mode is the default system for the VIC—the one which is active when no other mode is selected. The other two modes are not directly supported by the 128 operating system (there's no GRAPHIC statement to select these modes), so you must enable them by directly setting the appropriate bits in VIC registers. As a result, those modes are a bit more difficult to use effectively.

For a standard (GRAPHIC 0) character display, the 320-pixel X 200-line active screen area is divided into 1000 8-pixel X 8-line character positions, arranged as 25 rows with 40 character positions per row. The contents of the character positions are determined by values stored in a 1000-byte area of screen memory (sometimes referred to as the video matrix). Each location in screen memory corresponds to a single character position on the screen. The value in a screen memory location selects one of 256 standard character-pattern definitions to be drawn in the corresponding character position. The screen memory values are referred to as screen codes, and they are not the same as character codes. See Appendix C for a list of screen codes and corresponding character patterns. The location of screen memory within the current video bank is controlled by bits 4-7 of the VIC register at 53272/$D018. See the entry for that register for details.

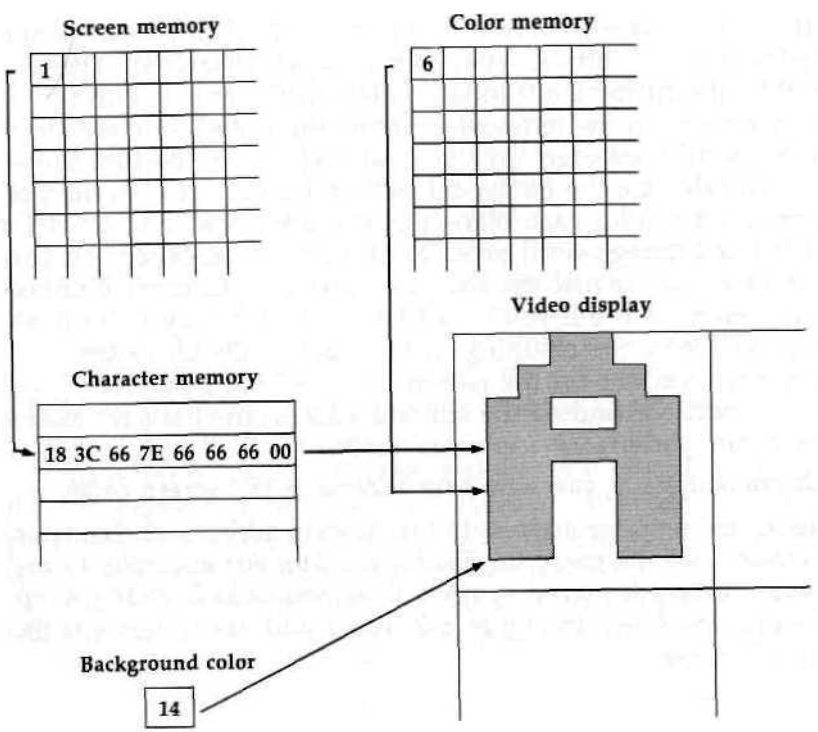
The pattern definitions come from another area of memory known as character memory. As mentioned above, each character position consists of eight scan lines with eight pixels per line—a total of 64 pixels per position. In standard character mode, a pixel can be one of two colors, so only one bit (which can be either %0 or %1) is required per pixel. Thus, a character-pattern definition requires 64 bits, or eight bytes. The pixels represented by %0 bits in the pattern definition are drawn in what is referred to as the background color, which is common to all screen positions. The pixels represented by %1 bits are drawn in what is referred to as the foreground color, which can be independently selected for each character position. The location of character-pattern memory within the current video bank is controlled by bits 1-3 of the VIC register at 53272/SD018. See the entry for that register for more details. Standard character definitions for the 128 come from the character ROM. This ROM is located beginning at address 53248/$D000 in the system's address space, but can be made visible in any video bank. See the section on character ROM later in this chapter for more information.

The background color for %0 bits in all character positions is determined by the value in the VIC register at 53281/$D02I. The foreground color for the %1 bits in each character position is determined by values in another 1000-byte area of memory known as color memory. As in screen memory, each location in color memory corresponds to a char-

acter position on the screen. Unlike the case of screen and character memory, however, the location of VIC color memory is fixed and does not have to be within the current video bank. It always appears to the processor at locations 55296-56319/ $D800-$DBFF in the 1/0 block. Refer to the section on color memory later in this chapter for details.

The procedure for displaying the character A in blue at the upper left corner of the screen would be something like this: The VIC looks to the first location of screen memory to determine which character pattern to display in that position. The screen code for A is 1, so this value is used as an index to the eight-byte pattern definition in character memory. The VIC then looks to the first location of color memory and proceeds to draw pixels in the color specified there (6 for blue) for all %1 bits in the pattern. For all %0 bits, pixels are drawn in the color specified in the background color register. Figure 8-1 illustrates the process.

Figure 8-1. Standard Character Display Mode



Custom Characters

You are not limited just to the character patterns provided in the ROM. It is relatively simple to design your own characters. However, using custom characters is an all-or-nothing affair. Once you switch off the standard ROM-based character set, you must provide definitions for every character you wish to use. You must begin by selecting the area of RAM where you will place the new character set. See the entry for the register at 53272/$D018 for details. If you only want to use a few custom characters while retaining the majority of the standard character set, the next step is to copy the standard character patterns from ROM to the selected RAM area. If you do this, you'll only have to provide custom pattern information for those characters you wish to redefine.

To calculate the proper byte values for a custom character definition, use an 8 X 8 grid as shown in Figure 8-2. Fill in the grid squares for those pixels you wish to have displayed in the foreground color. Unless you are designing patterns that will connect with adjacent patterns (such as the line segments in the standard character set), it is customary to leave at least one row and one column of the pattern blank to provide some horizontal and vertical separation between characters. The ROM patterns for the standard letters and numbers don't fill in any pixels in the leftmost column, and only lowercase characters with descenders (g, j, p, q, and y) use the bottom row.

To calculate the binary bit pattern for each row of the pattern, use a %0 for each blank (background) pixel and a %1 for each filled (foreground) pixel. Next, convert the binary bit pattern to a number (use the machine language monitor's number-conversion feature if you're not handy with binary). The final step is to store the resulting eight values in the character memory locations for the pattern you are changing. The simple formula for finding the starting address in character memory of the pattern for any character is:

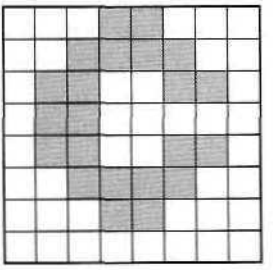
pattern address = character base address + (8 * screen **code)**

The character base address is the starting address of character memory (see the entry for 53272/$D018). For example, to replace the British pound symbol (£, screen code 28/$1C) with the pattern shown in Figure 8-2, you could use statements like the following:

```
600 FOR I = 0 TO 7
610 READ A
620 POKE 8192 + (8 * 28) + I,A
630 NEXT
640 DATA 24,60,102,96,102,60,24,0
```

This example assumes that character memory has previously been relocated to address 8192/$2000.

Figure 8-2. Custom Character Design Grid



```
% D0011000 =  24/$18
% 00111100 =  60/$3C
% 01100110 = 102/$66
% 01100000 =  96/$60
"A011001IO = 102/S66
% 00111100 =  60/S3C
% 00011000 =  24/S1B
% 00000000 =   0/$00
```

Multicolor Character Mode

Multicolor character mode is similar in operation to standard character mode. The difference is that each multicolor character position consists of 4 pixels X 8 lines (instead of 8 X 8). The number of positions remains the same (25 rows X 40 columns) and the positions are the same size, but now each pixel is twice as wide (there are only 160 pixels per raster line). However, each pixel can be one of four colors instead of just one of two colors. Screen memory still holds pointers to pattern definitions in character memory, but the pattern information is interpreted differently. It now takes two bits per pixel to select the color instead of just one, but since there are only half as many pixels per pattern, the number of bits required for each definition remains the same (2 bits per pixel * 4 pixels * 8 lines = 64 bits).

To select multicolor character mode, you must set bit 4 of the VIC register at location 53270/$D016. However, there's a problem here because the screen editor IRQ routine always resets this bit to %0 when setting up the text screen (see the section below on the screen editor IRQ routine). To prevent this, you must turn off the screen-setup portion of the IRQ by storing the value 255/$FF in location 216/SD8. Setting the VIC register bit makes it possible to enable multicolor character

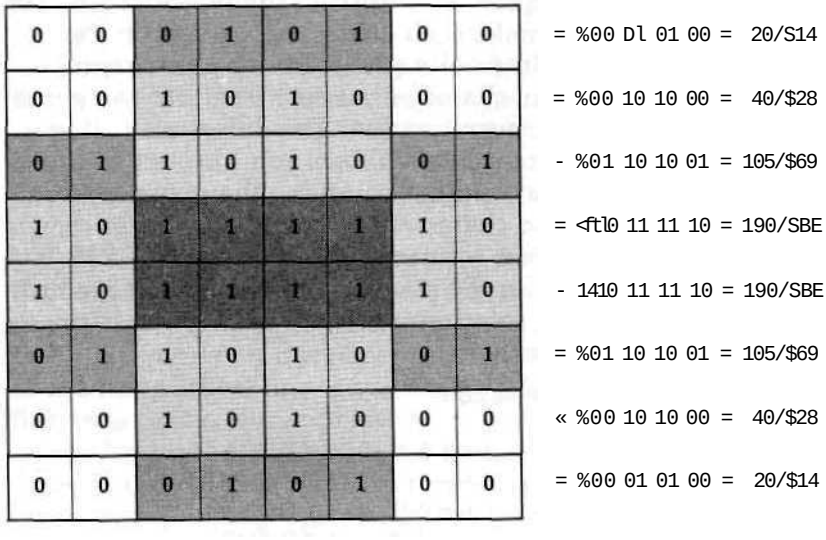
mode for any or all screen positions, but it doesn't actually switch any screen positions to multicolor mode. Multicolor mode must be enabled individually for each character position on the screen. The controlling factor is bit 3 of the color memory location for each character position. When that bit is %0 in a color memory position, the corresponding character position on the screen remains in standard character mode, so it is possible to intermix standard and multicolor character modes on the same screen display. However, only bits 0-2 of the color memory location are now available to hold color values, so the foreground color for standard mode positions in a multicolor display is limited to the first eight values in Table 8-1—black-yellow, values 1-7. To select multicolor character mode for a screen position, you must set bit 3 of the corresponding color memory location to %1. That is, you must store a value of 8 or greater in the location.

When a multicolor character is drawn, all pixels in the pattern represented by %00 bit pairs will be drawn in the background color specified in the VIC register at 53281/$D021. All pixels represented by %01 bit pairs will be drawn in the color specified by the value in the register at 53282/$D022, and all pixels for %10 bit pairs will be drawn in the color specified in the register at 53283/$D023. All of these registers can take any of the 16 standard colors listed in Table 8-1, but since the registers are common to all positions, the color for pixels with %00, %01, and %10 patterns will be the same for all characters. However, the color for pixels with %11 bit patterns can be specified individually for each screen position in the corresponding color memory location. Since bit 3 of the color memory location is used to specify multicolor mode, only bits 0-2 are available to hold color values. As a result, only the first eight colors are available. But since bit 3 must be set to %1, the values you store in color memory to achieve these colors are different from the standard values. For multicolor character mode, the values to store in color memory to select the available colors for %11 bits are as follows:

Desired %11 bit color	Value to store in color memory
black	8/$08
white	9/$09
red	10/$0A
cyan	ll/$0B
purple	12/$0C
green	13/$0D
blue	14/$0E
yellow	15/$0F

Because the standard character sets were not designed to be displayed in multicolor character mode, any text printed to the screen in this mode will be at best barely legible. As a result, multicolor mode is practical only when you are using custom characters designed specifically for this mode. The rules for designing custom characters for multicolor mode are the same as for the standard character mode, except that you design the characters in a 4 X 8 grid, as shown in Figure 8-3.

Figure 8-3. Multicolor Character Design Grid



Each grid position can hold one of four two-bit values representing the four color choices:

%00 Background color 0 (common to all characters)
%01 Background color 1 (common to all characters)
%10 Background color 2 (common to all characters)
%11 Foreground color {independently selectable for all characters, but only eight colors are available)

Once the design is completed, the byte values for the character-pattern definition are calculated just as for standard character mode.

Extended Background Color Mode

The third character mode, extended background color mode, is selected by setting bit 6 of the VIC register at 53265/$D011 to %1. It also uses the same fundamental elements as standard character mode: screen memory, character memory, and color memory. As in standard character mode, the extended background color mode screen is divided into a 25-row X 40-column grid of 8-pixel X 8-line character positions. As in standard character mode, each position has a corresponding screen memory location that holds a value indicating which pattern from character memory is to be drawn in the position. And, as in standard character mode, each character position on the screen takes its foreground color (the color for pixels represented by %1 bits in the character pattern) from the value in a corresponding color memory location. The difference is that extended background color mode allows you to select from among four different background colors for the pixels represented by %0 bits in the character patterns.

The background color for each position is specified by bits 6-7 of the screen code for the position. These bits select which of the four background color registers will specify the background color for the position:

Bits 7 6	Background color source
0 0	background color register 0 (53281/$D021)
0 1	background color register 1 (53282/$D022)
1 0	background color register 2 (53283/SD023)
1 1	background color register 3 (53284/$D024)

Since the highest two bits of each screen memory location are used to specify background color, only bits 0-5 are available to hold screen code data. Thus, there are only 64 different

unique screen code values (0-63), so only the first 64 eight-bit pattern definitions in character memory are used in this mode. For example, screen memory values of 1/$01, 65/$41, 129/$81, and 193/$C1 all produce the same character (screen code 1, the letter *A* in the standard character set), but each provides a different background color for that character.

Bitmapped Display Modes

The VIC provides for two bitmapped modes: standard and multicolor. In these modes, the state of each pixel in the screen display can be controlled independently. The standard bitmapped mode allows you to select one of two colors for each pixel, while the multicolor mode allows you to chose from among four colors. Both modes are supported by the operating system: standard bitmapped mode as GRAPHIC 1 (or, with a text window, as GRAPHIC 2) and multicolor bitmapped mode as GRAPHIC 3 (or, with a text window, as GRAPHIC 4).

Standard bitmapped mode is selected when bit 5 of the VIC register at 53271/$D017 is set to %1 (but see the section below on the screen editor IRQ for information about the shadow for this bit). This mode provides for 320 horizontal pixels per line, each of which can be one of two colors. A single bit is required to specify the color of each pixel, so 320 * 200, or 64,000 bits, are required to "map" the entire display area. At 8 bits per byte, 8000 bytes are required for the bitmap. This is half of the available space in the 16K video bank. The starting address of the bitmap is specified in bit 3 of the register at 53272/$D018.

The VIC's scheme for mapping the screen is simple for the chip (it's a variation of character mode), but it's rather complicated for the programmer. As you would expect, the first eight pixels on the screen, starting in the upper left corner of the first vertical line, are controlled by the eight bits of the first byte of the bitmap. However, the next eight pixels are controlled by the bits of the ninth byte. The bits of the second through eighth bytes in the bitmap control the leftmost eight pixels of the second through eighth vertical lines. This scheme is repeated across the screen. Figure 8-4 illustrates the offsets from the bitmap starting address for the bytes which control the pixels in the upper left corner of the screen.

Figure 8-4. Byte Offsets for Bitmapped Screen

0	B	16	
1	9	17	
2	10	18	
3	11	19	
4	12	20	
5	13	21	
6	14	22	
7	IS	23	
320	328	336	
321	329	337	
322	330	338	
323	331	339	
324	332	340	
325	333	341	
326	334	342	
327	335	343	
640	648	656	
641	649	657	
642	650	658	
643	651	659	
644	652	660	
645	653	661	
646	654	662	
647	655	663	
960	968	976	

This obviously isn't very convenient. Most programmers prefer to use a more familiar ^-coordinate system, as shown in Figure 8-5. In this system, the horizontal *(x)* pixel position will be in the range 0-319 and the vertical *{y)* position will be in the range 0-199. In *x,y* format, the upper left corner of the screen is position 0,0 and the lower right corner is position 319,199. This is also the format used to specify screen positions in BASIC statements such as BOX, CIRCLE, and DRAW, To determine the byte offset (0-7999) within the bitmap and the bit (0-7) within that byte which corresponds to a particular ^-coordinate pair, use the following formulae:

byte offset = 40 * (y AND 248) + (x AND 504) + (y AND 7)
bit = 7 - (x AND 7)

where *x* will have a value in the range 0-319 and *y* will have a value in the range 0-199.

Each bit in the bitmap can be either %0 or %1, so each corresponding pixel on the screen can be one of two colors. By convention, the color specified by a %0 bit is referred to as the background color and the color specified by a %1 bit is referred to as the foreground color. The values for both colors come from the video matrix, the area used as screen memory in the character display modes. (Color memory and the background color registers are not used in standard bitmapped mode.) The low nybble (bits 0-3) of a video matrix location holds the background color value and the high nybble (bits 4-7) holds the foreground color value. Either foreground or background can take any of the 16 colors listed in Table 8-1. However, since there are only 1000 video matrix positions, the foreground and background color cannot be specified individually for each of the 64,000 pixels on the screen. All 64 pixels within an 8-pixel X 8-line area will share foreground and background colors. The common areas are arranged in the same fashion as screen memory: 25 rows X 40 coiumns.^ To determine the video matrix location that holds the color information for a particular *x,y* coordinate, use the following formula:

color byte = 40 * INT(y / 8) + INT(x / 8) + screen base address

Multicolor Bitmapped Mode

Multicolor bitmapped mode is similar to bitmapped mode, but the number of possible colors per common color area is increased to four. To select among four different colors, two bits are required for each pixel. Since twice as many bits are required to specify the color of each horizontal pixel, only half as many pixels can be displayed per line; horizontal resolution is reduced to 160 pixels per line. The display will still be the same size, but each pixel will now be twice as wide. To determine the byte offset (0-7999) within the bitmap and the bit pair (0-3) within that byte which correspond to a particular ^-coordinate pair, use the following formulae:

byte offset = 40 * (y AND 248) + 2 * (x AND 252) + (y AND 7)

bit pair = 3 - (x AND 3)

where *x* will have a value in the range 0-159 and y will have a value in the range 0-199.

Of the four color sources in multicolor mode, one is common to all pixels on the screen and the other three can be selected independently for each common color area. Common color areas correspond in size {4 pixels X 8 lines) and layout (40 X 25) to multicolor character positions. All pixels represented by %00 bit patterns in the bitmap will take the color specified in the VIC background color register at 53281/ $D021. As in standard bitmapped mode, the video matrix (screen memory) area holds color information. In this case, the lower nybble (bits 0-3) of each video matrix location specifies the color for all %10 bit patterns within the corresponding common color area, while the upper nybble (bits 4-7) specifies the color for alt %01 bit patterns in the common color area. The color for any %11 bit patterns in each common color area is determined by the value in the corresponding color memory location (55296-56295/$D800-$DBE7). Any of these color sources can take any of the 16 color values listed in Table 8-1.

Sprites

Sprites, which Commodore officially calls movable object blocks (MOBs), are a special feature of the VIC. As their official name implies, sprites are images that can be easily moved about on the screen. Sprites are completely independent of the background display, and can be made to appear either in front of or behind screen foreground objects. They can move with equal ease over character and bitmapped screens. The manipulation of sprites consumes a substantial portion of the VIC chip's internal hardware. You'll notice in the discussion of VIC registers that 34 of the chip's 49 registers are used for some sort of sprite manipulation. This section uses the standard VIC number designations, 0-7, for the eight sprites. BASIC, on the other hand, uses sprite numbers 1-8. Add 1 to the VIC sprite number to get the corresponding BASIC sprite number, or subtract 1 to convert the BASIC sprite number to a VIC sprite number.

Sprites have the same two basic modes as screen displays: standard and multicolor. Standard sprites are 24 pixels wide X 21 scan lines tall, and can have only one color. Multicolor sprites are 12 pixels wide X 21 scan lines tall, and can have three colors. However, multicolor sprites are the same size as standard sprites because the multicolor pixels are twice as

wide. Sprites can also be doubled in size horizontally or vertically (see the registers at 53271/$D017 and 53277/$D01D).

The rules for defining sprite bit patterns are the same as for custom characters in the corresponding screen modes. Each standard sprite pixel is represented by one bit in a pattern bitmap, while each multicolor sprite pixel requires two bits. Thus, three bytes are required to define each scan line of the pattern, and each sprite pattern definition requires 3 * 21, or 63 bytes. The rules for calculating byte values are the same as for custom character patterns. Figure 8-5 shows a sprite design grid.

Figure 8-5. Sprite Design Grid

For standard sprites, all pixels represented by %0 bits in the definition pattern will be transparent. That is, whatever is on the screen behind the sprite will show through. Pixels represented by %1 bits take the color specified in the color register for that sprite (53287-53294/$D027-$D02E), so each sprite can take a different sprite foreground color. For multicolor

sprites, pixels represented by %00 bit patterns are transparent. Pixels represented by %01 and %11 patterns take the colors specified in the sprite multicolor registers (53285/$D025 and 53286/$D026, respectively). These colors are common to all eight sprites. Multicolor pixels represented by %10 bit patterns take the color specified in the sprite foreground color registers.

The 63 data bytes for the sprite pattern can't be placed just anywhere in memory. The definitions must be located within the current VIC video bank, and must begin at an address which is an exact multiple of 64. A 16K VIC video bank has room for 16384 / 64, or 256 sprite patterns. The pattern for each of the eight sprites is determined by the value in a corresponding sprite pointer. The sprite pointers don't occupy any fixed locations. Rather, they are found at the highest eight locations of the current screen memory (video matrix) area, at offsets of 1016-1023 bytes from the start of the area. The pointer value (0-255) selects one of the 256 sprite pattern areas. The relationship between pointer values and definition pattern area starting addresses is as follows:
pointer value = pattern starting address / 64
or:
pattern starting address = pointer value * 64

The 128 reserves locations 3584-4095/$0E00-$OFFF in block 0 RAM to hold sprite pattern data. This 512-byte area provides room for eight patterns, one for each of the eight sprites. The sprite pointers are initialized to point to patterns in this area as follows:

Sprite	Pointer value	Pattern address
0	56/$38	3584-3647/$0E00-$0E3F
1	57/$39	3648-371l/$0E40-$0E7F
2	58/$3A	3712-3775/$0E80-$0EBF
3	59/$3B	3776-3839/$0ECO-$0EFF
4	60/$3C	384O-3903/$0F0O-$0F3F
5	61/$3D	3904-3967/$0F40-$0F7F
6	62/$3E	3968-403l/$0F80-$0FBF
7	63/$3F	4032-4095/$0FC0-$0FFF

Even after a sprite is assigned a pattern, it will not appear on the screen until it is enabled and moved into the visible area of the screen display. Sprites are enabled by setting the appropriate bits in the register at 53269/$D0l5. The position of each sprite on the screen is specified by values in the regis-

ters at 53248-53264/$D000-$D010. Refer to the discussion of those registers for details.

When two sprites positions' overlap, one will appear in front of the other. The one that appears in front is said to have higher priority. The priority of the sprites in relation to each other is fixed. Sprite 0 has the highest priority, and will appear in front of any other sprites it may overlap. Sprite 1 has the next highest priority; it can appear in front of any sprite except sprite 0. The priority decreases with increasing sprite number, down to sprite 7, which appears behind any other sprite it may overlap. The priority of sprites in relationship to screen foreground objects is programmable; sprites can appear to pass either in front of or behind screen foreground pixels. See the discussion of the register at 53275/$D01B for details.

When two sprites overlap, or when a sprite overlaps screen foreground pixels, a *collision* is said to occur. The VIC records these collisions automatically, and can generate interrupts as a result. See the discussion of the registers at 53278-53279/$D01E-$D01F.

Screen IRQ Routines

The 128 introduces a feature that may be unfamiliar to those with previous Commodore experience: shadow registers. A shadow register is a RAM memory location that is copied into a hardware register at regular intervals. Shadow registers are a feature of the system's software, not its hardware. The system IRQ interrupt sequence, the collection of routines executed every 1/60 second (1/50 second in PAL systems), includes two separate sections which affect the VIC chip. The screen editor IRQ routine [$C194] controls the screen mode and raster interrupt, and the BASIC IRQ routine [$A84D] controls sprite movement, detects sprite collisions, and reads the light pen. Because these routines maintain shadows of some VIC registers, the registers cannot be changed directly while the normal interrupt sequence is active. If you try to store a new value in a register that has a shadow, the interrupt will replace your value with the shadow register contents at the next system IRQ interrupt—within 1/60 second. The discussion of the VIC registers below notes which registers have shadows and explains how to go about changing such registers. Refer to the appropriate ROM routine entry for more information on the interrupt routines.

VIC Registers

Table 8-2 is a summary of the VIC chip's registers. A detailed description of each register follows.

Table 8-2. VIC Chip Registers

Address	Function
53248/ID000	Sprite 0 horizontal position register
53249/$D001	Sprite 0 vertical position register
53250/$D002	Sprite 1 horizontal position register
53251/$D003	Sprite 1 vertical position register
53252/$D004	Sprite 2 horizontal position register
53253/$D005	Sprite 2 vertical position register
53254/$D006	Sprite 3 horizontal position register
53255/$D007	Sprite 3 vertical position register
53256/$D008	Sprite 4 horizontal position register
53257/$D009	Sprite 4 vertical position register
53258/$D00A	Sprite 5 horizontal position register
53259/$D00B	Sprite 5 vertical position register
53260/$D00C	Sprite 6 horizontal position register
53261/$D00D	Sprite 6 vertical position register
53262/$D00E	Sprite 7 horizontal position register
53263/$D00F	Sprite 7 vertical position register
53264/SD010	Sprites 0-7 horizontal position (most significant bits)
53265/$D011	Control/vertical fine scrolling register
53266/$D012	Raster scan-line register
53267/$D013	Light pen horizontal position
53268/SD014	Light pen vertical position
53269/$D015	Sprite enable register
53270/$D016	Control/horizontal fine scrolling register
53271/$D017	Sprite vertical expansion register
53272/$D018	Memory control register
53273/$D019	Interrupt flag register
53 2 74/$ DO 1A	Interrupt mask register
53275/$D01B	Sprite-to-foreground priority register
53276/$D01C	Sprite multicolor mode register
53277/$D0ID	Sprite horizontal expansion register
53278/$D01E	Sprite-sprite collision register
53279/$D01F	Sprite-foreground collision register
53280/$D020	Border color register
53281/$D021	Background color (source 0) register
53282/$D022	Background color (source 1) register
53283/$D023	Background color (source 2) register
53284/$D024	Background color (source 3) register
53285/SD025	Sprite multicolor (source 0) register

Address	Function
53286/$D026	Sprite multicolor (source 1) register
53287/$D027	Sprite 0 color register
53288/$D028	Sprite 1 color register
53289/$D029	Sprite 2 color register
53290/ID02A	Sprite 3 color register
53291/$D02B	Sprite 4 color register
53292/$D02C	Sprite 5 color register
33293/$D02D	Sprite 6 color register
53294/$D02E	Sprite 7 color register
53295/$D02F	Extended keyboard scan register
53296/$D030	Processor clock rate control register

53248	$D000	SPOX
53249	$DOO1	SPOY
53250	$D002	SP1X
53251	$D003	SP1Y
53252	$D004	SP2X
53253	$DOO5	SP2Y
53254	$D006	SP3X
53255	$D007	SP3Y
53256	$D008	SP4X
53257	$D009	SP4Y
53258	$D00A	SP5X
53259	$D00B	SP5Y
53260	$D00C	SP6X
53261	$D00D	SP6Y
53262	$D00E	SP7X
53263	$D00F	SP7Y
53264	$D01O	MSIGX

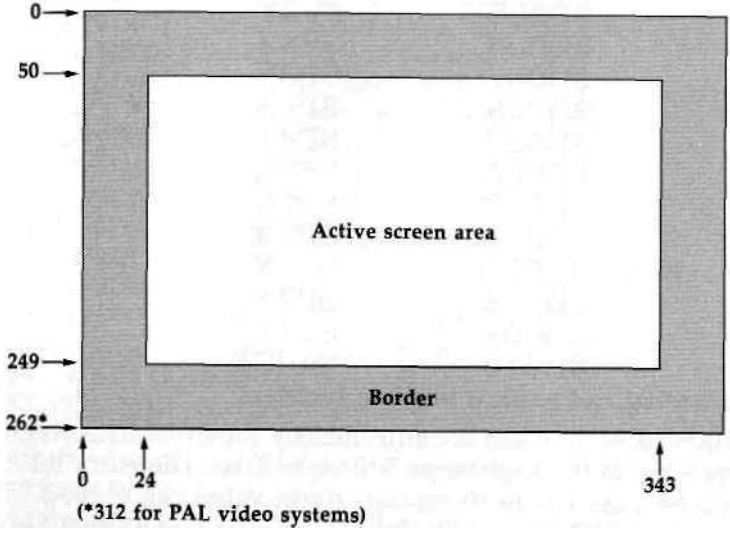
Sprite horizontal and vertical position registers

The position of each sprite is controlled by a pair of these registers plus a bit in the register at 53264/$D010. The extra bit is required because the horizontal position value can exceed 255. The extra bit is effectively the ninth (most significant) bit of the horizontal position register. The relationship of registers to sprites is as follows:

Sprite	Horizontal register	Bit in $D010	Vertical register
0	53248/$DO00	0	53249/$D001
1	53250/$D002	1	53251/$D003
2	53252/$D004	2	53253/$D005
3	53254/$D006	3	53255/$D007
4	53256/$D008	4	53257/$D009
5	53258/$DO0A	5	53259/$D00B
6	53260/$DOOC	6	53261/$D00D
7	53262/SDOOE	7	53263/SD00F

The values in these registers specify a sprite's position on the screen. (Actually, the values set the position of the upper left corner of the 24 X 21-pixel sprite pattern.) The coordinate system used is slightly different from the one used for bit-mapped graphics, as illustrated in Figure 8-6. Note that the sprite coordinate system remains the same regardless of the current screen mode (character or bitmapped) or sprite mode (standard or multicolor).

Figure 8-6. Sprite Position Values



The horizontal position range includes both the active screen area and the inactive left and right borders, in units that equal standard screen pixel widths. The vertical position range also includes the top and bottom borders in addition to the active screen area. Note that the horizontal position can be

greater than 255, which is the largest value that can be stored in a horizontal position register. To move a sprite to horizontal position 256, you must set the horizontal position register to 0/$00 and set the most significant horizontal bit for that sprite (in the register at 53264/$D010) to %1. The most significant bit must remain set for all positions greater than 255. Because of the extra programming required to move the sprite to this right portion of the screen, the vertical column at horizontal position 256 is frequently referred to as the *seam.*

The values in these registers cannot be changed directly while the normal BASIC IRQ routine [$A84D] is in use. That routine copies the contents of the shadow sprite position locations (4566-4582/$llD6-$llE6) into these registers during each system IRQ interrupt. (See the discussion of the BASIC IRQ routine in Chapter 5 for more details.) There are two ways to deal with this. The simplest solution is to store the desired sprite position values in the proper shadow location (see the entry for the shadow locations in Chapter 3). If you wish to use the true registers, you can prevent the execution of the BASIC portion of the IRQ interrupt. This will disable most BASIC sprite and sound commands, but that shouldn't be a problem for machine language programmers. To prevent execution of the BASIC IRQ routine, you can either set bit 0 of the initialization status flag at 2564/$0A04 to %0 (which tells the Kernal that BASIC is not initialized), or you can store any nonzero value in the BASIC IRQ status flag at 4861/S12FD.

53265 SD011 SCROLY

Vertical smooth scrolling and control register

Bits 0-2: These bits control the VlC's vertical fine scrolling feature. The value here specifies how many scan lines downward the display should be shifted. The available three bits allow the screen to be scrolled up to seven scan lines. The IOINIT routine [$E109] initializes these bits to %011 = 3, so the default display will be scrolled down three scan lines from its highest possible position.

The display scrolls without wrapping; blank scan lines are moved in from the top and scan lines on the bottom move off the visible screen. Pixels in scan lines scrolled off the bottom of the display are not erased. They become visible again if the display is scrolled back upward by reducing the value in these bits. The 24-row feature is useful with scrolling because it cre-

ates hidden scan lines at the top and bottom of the screen that can be scrolled onto the visible area.

Bit 3: This bit determines the vertical height of the active portion of the screen display. Setting this bit to %1 selects a 25-row (200 scan-line) active screen. The bit is initialized to this setting during the IOINIT routine [$E109], and neither the operating system nor BASIC changes this value. Setting this bit to %0 reduces the height of the active portion of the display to 24 rows (192 scan lines) by blanking the top four and bottom four scan lines of the display. The pixels in the blanked columns aren't erased; they'll still be intact when the screen is switched back to 25 rows. The 24-row feature is useful in conjunction with the vertical scrolling feature described in bits 0-2.

Bit 4: This bit enables or disables the VIC screen display. While the bit is %1, the VIC provides its normal screen output. The bit is initialized to this setting during the IOINIT routine [$E109]. When the bit is set to %0, the VIC suspends the active portion of the display and provides a solid screen in the border color specified in the register at 53280/$D020. The screen is not erased, just blanked. Any text or graphics will still be intact if the screen is again enabled. With the display blanked, the VIC doesn't have to steal timing cycles from the processor as it normally does. As a result, the 128 actually runs about 7 percent faster with the VIC display off.

There are several uses for this blanking feature. For example, you could blank the display while drawing a complex bitmapped graphics screen, then dramatically unveil the completed picture by reenabling the display. Since the VIC cannot provide a proper display at the 2-MHz clock rate, the VIC display is normally blanked when the system is switched to fast mode. One step in the BASIC statement FAST involves setting this bit to %0. The SLOW statement returns it to %1. This bit is also set to %0 to blank the screen during tape operations, and restored to %1 upon completion of the operations.

Bit 5: This bit selects whether the VDC will generate a character mode display or a bitmapped mode display. (See the introduction to this section for a discussion of the display modes.) Setting this bit to %0 selects character mode, while setting it to %1 selects bitmapped mode.

This bit cannot be directly modified while the normal system interrupt sequence is active because a step in the screen

editor IRQ routine [$C194] sets this bit according to the screen mode flag (location 216/$D8). To change the setting of this bit, you can either store the appropriate value in the flag location, or you can disable the screen-setup portion of the IRQ routine and change this bit directly. To use the flag location, set bit 5 of location 216/$D8 to the desired setting (%0 or %1) for the register bit. To turn off the screen-setup portion of the interrupt routine, store the value 255/SFF in location 216/$D8.

Bit 6: This bit controls extended background color mode, which offers a choice of four different background colors for each character position. For details about this mode, refer to the introduction for this section. Extended background color mode is enabled when this bit is set to %1. The mode works only in conjunction with character mode. You should not select extended background color mode while bit 5 is set to %1 to select bitmapped mode.

Bit 7: This bit is effectively the ninth bit of the raster register at 53266/$D011. Refer to the discussion of that register for details. The extra bit is necessary because a full screen contains more than 256 scan lines.

53266 $D012 RASTER

Raster compare register

This register has two different functions, depending on whether it is being read from or written to. As explained in the introduction for this section, the video screen display consists of a stack of thin horizontal lines of dots called a raster. When the register is read, the value returned is the number of the raster scan line currently being drawn. The range of scan-line numbers depends on the video system in use. For NTSC (North American) systems, values can be in the range 0-262, while PAL (European) systems have a maximum count of over 300. In either case, the active portion of the screen consists of scan lines 50-249.

The maximum scan-line number in either system is larger than can be held in a single eight-bit register, so bit 7 of the register at 53265/$D011 is used to hold the ninth bit of the value. When bit 7 of 53265/$D011 is %1, you should add 256/ $100 to the value in this register to get the true scan-line number. For scan line 262, for example, bit 7 of 53265/$D011 will be %1 and the value in this register will be 6/$06. Be-

cause the value in this register changes so rapidly (over 15,000 times per second), it can't be read usefully from BASIC. By the time you PEEKed the value into a variable, the raster scan line would have moved far beyond the line recorded in the variable value.

When you write to this register or to bit 7 of 53265/$D011, the value written is stored in an internal nine-bit raster compare latch register. Whenever the current scan-line count equals the value in this register, bit 0 of the interrupt register at 53273/$D019 will be set to %1. If bit 0 of the interrupt mask register at 53274/$D01A was previously set to %1, this will also trigger an external interrupt request to the processor. The value you store in this register represents the scan line at which you wish the interrupt to occur. For scan-line values less than 256, you must also set bit *7* of 53265/$D011 to %0. For scan-line values of 256 or greater, you must set bit 7 of 53265/$D011 to %1 and store the line number minus 256 in this register. To calculate the top scan line corresponding to any row of character positions, use the following formula:
scan line = (row * 8) + 50
where the row value is in the range 0-24.

Raster interrupts can be used to program a variety of special video effects, including split screens like those in the GRAPHIC 2 and GRAPHIC 4 modes. The 128 also uses a raster interrupt off the visible screen (at scan line 255) to drive the system IRQ sequence. Because of this, you cannot write a new value directly to this register while the the normal system interrupt sequence is in use. The screen-setup portion of the screen editor IRQ routine [$C194] will write the value 255/$FF to this register on every pass, except when setting up the bit-mapped portion of a split bitmapped/text screen. In that case, the value in location 2612/$0A34 will be copied into the register. In either case, bit 7 of the register at 53265/$D011 will be set to %0. To use a raster interrupt for your own purposes, you must write a new interrupt routine. Refer to Appendix A for more information.

53267	$D013	LPENX
53268	SD014	LPENY

Light pen horizontal and vertical positions

Whenever the VIC's LP input line is brought to a low (0 volts) state, the raster beam's horizontal dot position and vertical

scan line are latched into these registers. The register at 53267/SD013 will hold the horizontal position value, and the one at 53268/$D014 will hold the vertical position value. To signal that a new value has been latched, bits 7 and 3 of the interrupt register at 53273/SD019 will be set to %1. These registers are read-only; writing to them has no effect. The registers are not cleared when read; the latched values will be retained until the LP line is again brought low.

The range of scan-line values in the register at 53268/$D014 is the same as the range of sprite vertical positions shown in Figure 8-6. For example, the top scan line of the active screen area is 50, In that figure, you'll note that the range of horizontal positions extends to 343, which is greater than can be represented in a single eight-bit register. To compensate, the range of horizontal values in the register at 53267/$D013 is the equivalent of one-half the range of horizontal values shown in Figure 8-6. For example, the horizontal-position value for the center of the screen is 184, so the corresponding light pen position will be about 92.

The LP line is connected to pin 6 of control port 1 (control port 2 does not support a light pen). A light pen has at its tip an electronic device known as a phototransistor, which is connected so as to cause a low pulse whenever the video beam moves past the pen. These registers can be tricked into reading false values. Pin 6 of control port 1 is also used for light pen input for the VDC chip, so a light pen signal generated on the 80-column screen will latch meaningless values in these registers. In lieu of a light pen, several other events can cause a pulse on the LP line. That control port pin is also used for the joystick fire button, so pressing the button of a joystick plugged into port 1 will also latch values in these registers. Because of this joystick button function, the port line is also connected to the line from row 4 of the keyboard matrix. This has two consequences. First, pressing any of the following keys with no light pen connected will latch meaningless values: Fl, Z, C, B, M, period, right SHIFT, space, the 2 and ENTER keys on the numeric keypad, and the ^ key in the cursor group. More significantly, while a light pen is connected, all of these keys will be "dead," and cannot be typed.

53269 $D015 SPENA
Sprite enable register

This register controls which sprites are enabled. Only enabled sprites can be visible, and only enabled sprites can be involved in collisions. Setting a bit in this register to %1 enables the corresponding sprite. However, an enabled sprite won't be visible unless it is also positioned within the visible screen area, has a pattern definition that includes some nonzero bits, and is set to a color different from the screen background color. Setting a bit in this register to %0 turns off the corresponding sprite, but does not change the setting of any other parameters. When reenabled, the sprite will still have the same position, color, and pattern (unless those were changed while the sprite was turned off). The sprites are controlled as follows:

Bit	Bit value	Sprite controlled
0	1/$01	0
1	2/$02	1
2	4/$04	2
3	8/$08	3
4	16/$10	4
5	32/$20	5
6	64/$40	6
7	128/$80	7

This register is initialized to 0/$00 (all sprites disabled) by the IOINIT routine [$E109], part of the reset and RUN/STOP-RESTORE sequences. Sprites are normally disabled during tape and serial bus operations to prevent timing problems. In this case, the contents of the register are stored in location 2616/$0A38 for the duration of the operation, then restored to the register when the operation is completed. You can prevent this and keep sprites enabled by setting bit 7 of the custom mode flag (2618/$0A3A) to %1.

53270 $D016 SCROLX
Horizontal smooth scrolling and control register

Bits 0-2: These bits control the VIC's horizontal smooth scrolling feature. The value here specifies the number of pixels the display is to be shifted to the right. The available three bits allow the screen to be shifted as many as seven pixels. The display scrolls without wrapping; blank pixels are moved in from the left and pixels on the right move off the visible

screen. Pixels scrolled off the right are not erased, just hidden. They will become visible again if the display is scrolled back to the left by reducing the value in these bits. The 38-column feature is useful with scrolling because it creates a hidden left column that can hold pixels to be scrolled onto the visible area.

Bit 3: This bit determines the horizontal width of the screen. Setting this bit to %1 selects a 40-column (320-pixel) screen. The bit is initialized to this setting during the IOINIT routine [$E109], and neither the operating system nor BASIC ever change this value. Changing this bit to %0 reduces the width of the active portion of the display to 38 columns (304 pixels) by blanking the leftmost and rightmost columns. (Actually, 7 pixels on the left and 9 on the right are blanked.) The contents of the blanked columns aren't erased; they'll still be intact when the screen is switched back to 40 columns. The 38-column feature is useful in conjunction with the horizontal scrolling feature in bits 0-2. Scrolling the 38-column screen the maximum 7 pixels to the right will make the contents of the previously hidden leftmost column visible.

Bit 4: This bit controls multicolor mode for both the character and bitmapped screens. While this bit is %0, foreground pixels will be limited to one color (although the color can be different for every character position). Setting this bit to %1 enables multicolor mode, which allows a choice of three different colors for each foreground pixel. However, selecting multicolor mode also cuts horizontal resolution for the screen in half. Selecting multicolor mode for the screen has no effect on any sprites that might be displayed on that screen. Sprite multicolor mode is controlled by the register at 53276/$D01C.

This bit cannot be directly modified while the normal system interrupt sequence is active because a step in the screen editor IRQ routine [$C194] sets this bit according to the screen mode flag (location 216/$D8). To change the setting of this bit for bitmapped mode, you can either store the appropriate value in the flag location, or you can disable the screen-setup portion of the IRQ routine and change this bit directly. To use the flag location, set bit 7 of location 216/$D8 to the desired setting (%0 or %1) for the register bit. To turn off the screen-setup portion of the interrupt routine, store the value 255/$FF in location 216/SD8. To select multicolor character mode, you must use the option to disable the screen setup, since the text mode-setup subroutine always sets this bit to %0.

Bit 5: This bit is referred to in Commodore literature as the reset bit, but its intended usage is unclear. The value here has no apparent affect on any VIC operations.

Bits 6-7: These bits are unused. Writing to them has no effect, and they always return %1 when read.

53271 $D017 YXPAND

Sprite vertical expansion register

Each bit in this register controls the vertical expansion feature for one of the eight sprites. The relationship of sprites to register bits is the same as for the sprite enable register (53269/$D015). Setting a bit here to %1 will double the vertical height of the corresponding sprite. Each sprite can be expanded independently, with regard for the background screen on which the sprite is displayed. The resolution of the sprite is not increased—it will still be 21 pixels tall—but the height of the pixels will be doubled. Vertical expansion can be selected in conjunction with horizontal expansion to double the size of a sprite. (Horizontal expansion is controlled by the register at 53277/SD01D.) The default value in this register, established by the IOINIT routine [$E109], is 0/$00, so no sprites are initially expanded.

53272 $D018 VMCSB

Screen and character base address register

The value in this register determines the location within the current 16K VIC video bank of the two movable components of video memory: the screen memory/video matrix and character memory/bitmap areas. (There is no provision for moving color memory; that area always appears at 55296-56319/$D800-$DBFF.) The contents of this register cannot be changed directly while the normal system IRQ interrupt sequence is in use. The screen editor IRQ routine [$C194] copies the contents of a shadow location into this register during each interrupt. The shadow location depends on the screen mode in use. For text mode, or for the text portion of a split screen, the contents of location 2604/$0A2C will be copied here. For bitmapped modes, the value in location 2605/S0A2D will be copied into the register. You have two choices for changing the value in this register. You can either store the desired value in the appropriate shadow location, or you can turn off the screen-

setup portion of the screen editor IRQ routine to gain direct access to the register. To disable the screen-setup step, store the value 255/$FF in location 216/$D8.

Bit 0: This bit is unused. Writing to it has no effect, and it always returns %1 when read. Thus, the value read from this register will always be odd.

Bits 1-3: The value in these bits determines the location of character memory (for character mode) or of the bitmap (for bit-mapped modes). For character mode, a complete 256-character set requires 2048 (2K) bytes, and the character set must start on an even 2K memory address boundary. Possible selections are as follows:

Bits			Offset for
3	2	1	character set
0	0	0	0/$00
0	0	1	2048/$0800
0	1	0	4096/$1000
0	1	1	6144/$1800
1	0	0	8192/$2000
1	0	1	10240/$2800
1	1	0	12288/$3000
1	1	1	14336/S3800

These bits do not determine the absolute address of the character set, but rather the offset from the starting address of the current video bank. For example, video bank 2 begins at address 32768/$8000, so a bit setting of %100 in that case would place the character set at 32768 + 8192 = 40960/$A000. However, the default video bank (bank 0) begins at location 0/$0000, so in that case the offset is equal to the actual address.

The 128 normally provides a pair of character sets and allows you to switch between them by pressing the SHIFT - Commodore key combination. If you are setting up a single custom character set, you should disable this character set-switching feature by setting bit 7 of location 247/$F7 to %1. If you wish to retain the character set-switching feature, you must provide two character sets, one at an "even" position (one for which bit 1 of this register is %0), and the other at the next higher ("odd") position.

The CINT screen editor initialization routine [$C07B] sets bits 1-3 to %010 in location 2604/$0A2C, the shadow for this register in text mode. This places the default character memory at an offset of 4096/$1000 from the starting address of the

video bank (and implies that the alternate character set will be located at an offset of 6144/$1800), This may seem strange, since there is certainly no character pattern data stored in RAM at addresses 4096-8191/$1000-$lFFF, and since the 128's memory map shows that the character ROM is actually located at addresses 53248-57343/$D000-$DFFF. However, the 128 has the capability to make the VIC chip see character ROM at addresses 4096-6143/$1000-$17FF (for the uppercase/ graphics set) and addresses 6144-8191/$1800-$lFFF (for the lowercase/uppercase set) in any video bank. (Note that this is a change from the Commodore 64, which could only see character ROM in video banks 0 and 2.) Only the VIC will see the character ROM at those addresses. To the processor, those locations will still be RAM.

This feature is controlled by bit 2 of the 8502's built-in I/O port, at location l/$01. When bit 2 of the register at l/$01 is %0, the VIC chip will see character ROM in the character memory areas beginning at addresses 4096/$1000 and 6144/$1800. No other character memory slots are affected. When the I/O port bit is set to %1, the VIC will instead see the true contents of RAM at those addresses rather than images of the character ROM. Like other screen control locations, the I/O port bit has a shadow location. Bit 2 of location 217/SD9 is copied into bit 2 of location l/$01 during each pass through the text screen-setup portion of the screen editor IRQ routine. To switch out character ROM, you must set bit 2 of the shadow location to %1 (store the value 4/$04 in location 217/$D9). Alternatively, you can disable the screen-setup portion of the screen editor IRQ routine by storing the value 255/$FF in location 216/$D8; then you change bit 2 of location l/$01 directly.

Finding free space for a custom character set can be a challenge. If you are using machine language exclusively, any of the character set slots above address 4864/$1300 can be used, but none of the address slots are completely free in the standard configuration with BASIC. If you disable character ROM, there is free memory at 6144-7167/$ 18OO-$1BFF for half a character set (128 character patterns). If your program doesn't use a bitmapped screen, you can allocate a bitmap area and then use the reserved space for custom character sets. For example, if your BASIC program includes the statements GRAPHIC 1:GRAPHIC 0, the area from 7168-16383/ $1COO-$3FFF will

be protected from BASIC, and any character memory slot in that area can be used to hold the new character set.

Since the bitmap for a high-resolution display requires 8000 bytes of memory and must begin at an even 8K memory address boundary, there are only two possible positions for the bitmap within the 16K video bank. When you are specifying the location of the bitmap, only the setting of bit 3 is significant. When that bit is %0, the bitmap will begin at an offset of 0/$0000 from the start of the video bank. When bit 3 is %1, the bitmap begins at an offset of 8192/$2000 from the start of the video bank. The CINT screen editor initialization routine [$C07B] sets bits 1-3 to %100 in location 2605/$0A2D, the shadow for this register in bitmapped mode, so the default location of the bitmap is 8192/$2000 bytes beyond the start of the video bank (address 8192/$2000 for the default video bank).

Bits 4-7: These bits determine the location of the video matrix area, which is used as screen memory in character mode and to hold color information in bitmapped mode. The video matrix requires 1000 bytes of memory and must start on an even IK address boundary. Possible selections are as follows:

Bits				Offset for
7	6	5	4	video matrix
0	0	0	0	0/$0000
0	0	0	1	1024/$0400
0	0	10		2048/$0800
0	0	11		3072/$OC00
0	10	0		4096/$1000
0	10	1		5120/$1400
0	1	1	0	6144/$1800
0	1	1	1	7168/$1COO
10	0	0		8192/$2000
	0	0	1	9216/$2400
	0	1	0	10240/$2800
	0	1	1	11264/$2C00
	1	0	0	12288/$3000
	1	0	1	13312/$3400
	1	1	0	14336/S3800
	1	1	1	15360/$3CO0

These bits do not determine the absolute address of the video matrix, but rather the offset from the starting address of the current video bank. For example, video bank 3 begins at address 49152/$C000, so a bit setting of %0010 in that case would place the character set at 49152 + 2048 = 51200/

$C800. However, the default video bank (bank 0) begins at location 0/$0000, so in that case the offset is equal to the actual address.

The C1NT screen editor initialization routine [$C07B] sets bits 4-7 to %0001 in location 2604/$0A2C, the shadow for this register in text mode. This places the default character memory at an offset of 1024/$0400 from the starting address of the video bank. You can't change this register directly while the normal system interrupt sequence is active. To move the video matrix for the text screen, you can either change the value in location 2604/$0A2C, or you can turn off the screen-setup portion of the interrupt sequence by storing the value 255/$FF in location 216/$D8. After that, you can change the register directly. Even if you change the value in this register, all printed characters will continue to go to the former screen memory range until you change the value in location 2619/ $0A3B to reflect the new starting page for screen memory.

Because the bitmap for a high-resolution display requires half of the memory available in a 16K video bank, only half of the possible addresses are really useful. When setting up a bit-mapped display, you should select a video matrix area in the portion of the video bank not occupied by the bitmap. The CINT screen editor initialization routine [$C07B] sets bits 4-7 to %0111 in location 2605/$0A2D, the shadow for this register in bitmapped mode, so the default location of the video matrix in that mode is 7168/$1COO bytes from the start of the video bank (address 7168/$1COO for the default video bank). Thus, the bitmapped mode video matrix will not disturb character mode screen memory.

To move the video matrix for the bitmapped screen, you can either change the value in location 2605/$0A2D, or you can turn off the screen-setup portion of the interrupt sequence by storing the value 255/$FF in location 216/SD8. After that, you can change the register directly.

53273 $D019 VICIRQ
Interrupt register

This register is read-only; writing to this location has no effect. Bits 0-3 indicate the status of the four interrupt sources for the VIC chip. These bits will always reflect the status of their corresponding events, regardless of whether or not register 53274/$D01A has been set to trigger an external interrupt re-

quest for the event. Once a bit here is set to %1, it will retain that value until the register is read, at which time all functional bits are reset to %0.

Bit 0: This bit will be set to %1 whenever the raster scan-line count equals the value in the raster compare register 53266/$D012 (plus bit 7 of 53265/$D011).

Bit 1: This bit will be set to %1 whenever a nontransparent portion of a sprite overlaps any screen foreground pixels. (Remember, however, that for multicolor character or multicolor bitmapped screens, no collision can be detected between sprites and foreground pixels represented by %01 bit patterns.) The register at 53279/$D01F records which sprites are involved in collisions with foreground data.

Bit 2: This bit will be set to %1 whenever two or more sprites overlap. Collisions can only be detected between the foreground portion of the sprites, those pixels represented by nonzero bit patterns. No collision is detected when the overlapping portions of the sprites are transparent (when the pixels are represented by %0 or %00 bit patterns). The register at location 53278/$D01E records which sprites are involved in collisions with other sprites.

Bit 3: This bit will be set to %1 whenever a new value is latched into the light pen registers at 53267-53268/$D013-$D014.

Bits 4-6: The bits are not used, and always return %1 when read.

Bit 7: Whenever any internal interrupt source sets one of the other bits in this register to %1, this bit will also be set to %1 to indicate that an internal interrupt has been recorded. Thus, you need to test only this bit to determine whether an internal interrupt has occurred. Note that this register is automatically cleared after it is read, so you'll need to save the register value before testing this bit if you want to be able to subsequently read any of the other bits.

53274 **$D01A** **IRQMSK**

Interrupt enable register

Certain events such as sprite-sprite collisions always generate internal VIC interrupts, recorded in the register at 53273/

$D019. The VIC chip has the capability to generate an external interrupt request to the processor as a result of any of these conditions. In the 128, the VIC's interrupt request output line is connected to the processor's IRQ line, so internal VIC events can trigger processor IRQ interrupts. See Appendix A for more information on IRQ interrupts.

Bit 0: Setting this bit to %1 enables an external interrupt request when the raster count match is recorded in the flag at bit 0 of the register at 53273/$D019. Because raster interrupts are the normal source of the system jiffy interrupt that drives keyboard scanning and other important housekeeping features, this bit is initialized to %1 by the Kernal IOINIT routine [$E109]. Changing this bit to %0 will completely disable the system interrupt.

Bit 1: Setting this bit to %1 enables an external interrupt request when a sprite-foreground collision is recorded in the flag at bit 1 of the register at 53273/$D019, as when a sprite passes over a character on the text screen.

Bit 2: Setting this bit to %1 enables an external interrupt request when a sprite-sprite collision is recorded in the flag at bit 2 of the register at 53273/$D019.

Bit 3: Setting this bit to %1 enables an external interrupt request when a new value is recorded in the flag at bit 3 of the register at 53273/$D019. It is risky to use this interrupt source, because a number of events other than the light pen can trigger the latching of values into these registers. See the entry for the light pen registers for details.

Bits 4-7: These bits are unused; writing to them has no effect, and they always return %1 when read. Thus, the value seen when this register is read will always be at least 240/$F0. To mask off these unused bits and see the valid interrupt settings, use AND 15 in BASIC or AND #$0F in machine language.

53275 8D01B SPBGPR

Sprite-to-foreground priority register

Each bit in this register controls the sprite-to-foreground priority for one of the eight sprites. The relationship of sprites to bits is the same as for the sprite enable register (53269/$D015). While a bit here is %0, the corresponding sprite will have higher priority than the screen foreground, and will thus appear to pass in front of anything displayed in the fore-

ground color on the text or bitmapped screens. When a bit here is set to %1, the corresponding sprite will have lower priority than the screen foreground, and will thus appear to pass behind anything displayed in the foreground color. For multicolor character or multicolor bitmapped screen modes, the low-priority sprite will appear to pass behind screen pixels represented by %10 and %11 bit patterns, but will still pass in front of pixels represented by %01 bit patterns. (The sprite always appears in front of pixels with the %00 background bit pattern.) The default value for this register, established by the IOINIT routine [$E109], is 0/$00, so all sprites initially have higher priority than the screen foreground.

53276 **8D01C** **SPMC**

Sprite multicolor mode register

Each bit in this register controls the multicolor feature for one of the eight sprites. The relationship of sprites to register bits is the same as for the sprite enable register (53269/$D015). Setting a bit here to %1 selects multicolor mode for the corresponding sprite. Multicolor mode can be selected independently for each sprite, and standard and multicolor sprites can be mixed freely on the same screen. Multicolor sprites are not restricted to multicolor screen modes; they can be used freely in standard screen modes as well. The default value for this register, established during the IOINIT routine [$E109], is 0/$00, so no sprites are initially in multicolor mode.

Selecting multicolor mode for a sprite reduces the number of horizontal pixels for the sprite from 24 to 12. However, the sprite remains the same size; the multicolor sprite pixels are twice as wide. Two bits of the pattern definition are required for each pixel. All pixels represented by %00 are transparent; whatever is behind the sprite will show through. Pixels with %01 and %11 bit patterns take their color values from the registers at 53285/$D025 and 53286/$D026, respectively, which are common to all multicolor sprites. Pixels with %10 bit patterns take the color specified in the color register for the particular sprite (53287-53294/$D027-$D02E).

53277 $D01D XXPAND

Sprite horizontal expansion register

Each bit in this register controls the horizontal expansion feature for one of the eight sprites. The relationship of sprites to

register bits is the same as for the sprite enable register (53269/$D015). Setting a bit here to %1 will double the horizontal width of the corresponding sprite. Each sprite can be expanded independently, without regard for the background screen on which the sprite is displayed. The resolution of the sprite is not increased (it will still be 24 pixels wide) but the width of the pixels will be doubled. Horizontal expansion can be selected in conjunction with vertical expansion to double the size of a sprite. (Vertical expansion is controlled by the register at 53271/$D017.) The default value in this register, established by the IOINIT routine [$E109], is 0/$00, so no sprites are initially expanded.

53278 8D01E SPSPCL

Sprite-to-sprite collision register

This register records collisions between two or more sprites. This register is read-only; storing values here has no effect. A collision occurs whenever any nontransparent portions of the sprites overlap. Transparent pixels (pixels with bit patterns of %0 or %00) are not involved in collisions. When sprites collide, the bits in this register corresponding to those sprites are set to %1 (the correspondence between sprites and bits is the same as for the sprite enable register). Since a minimum of two sprites are involved in any sprite-sprite collision, at least two bits will be set. The register is automatically cleared to 0/$00 after each time it is read, so you'll need to store the value you read if you wish to perform multiple tests. Bit 2 of the interrupt register at 53273/$D019 will also be set to %1 whenever any sprite-sprite collision occurs.

This register indicates only that sprites have collided. It does not necessarily tell you which sprites are involved in a particular collision. If only two bits are set, then those sprites obviously must have hit each other. However, if you find that three or more bits are set, you must read the horizontal and vertical position registers for the involved sprites and determine which sprites are in contact with each other. Remember that the sprite position registers return the position of the upper left corner of the sprite, which may not be the point of the sprite that is colliding with the foreground.

One thing to beware of when reading this register is that after a sprite overlaps a foreground object, the register will continue to record collisions until the sprite is moved away

from the object. Also, sprite-sprite collisions can occur even when the sprites are located completely outside the visible screen area.

53279 $D01F SPFGCL

Sprite-foreground collision register

This register records collisions between sprites and the screen foreground. This register is read-only; storing values here has no effect. A collision occurs whenever any pixel in a sprite represented by a nonzero bit pattern overlaps any foreground pixel on the screen, as when a sprite passes over a character on the text screen. Background or transparent pixels are not involved in collisions. For the purposes of collision detection, multicolor screen pixels represented by %01 bit patterns are not considered foreground, and no collision will be detected when a sprite pixel overlaps a multicolor screen %01 pixel. When a sprite collides with a foreground pixel, the bit in this register corresponding to that sprite is set to %1 (the correspondence between sprites and bits is the same as for the sprite enable register). The register is automatically cleared to 0/$00 after each time it is read, so you'll need to store the value you read if you wish to perform multiple tests. Bit 1 of the interrupt register at 53273/$D019 will also be set to %1 whenever any sprite-foreground collision occurs.

This register indicates only that a sprite has collided with some portion of the screen foreground. It does not tell you exactly what the sprite is overlapping. To determine that, you must determine which sprite or sprites are involved in collisions, then read the horizontal and vertical position registers for the involved sprites and determine what screen-foreground object is located in that vicinity. Remember that the sprite position registers return the position of the upper left corner of the sprite, which may not be the point of the sprite that is colliding with the foreground.

One thing to beware of when reading this register is that after a sprite overlaps a foreground object, the register will continue to record collisions until the sprite is moved away from the object.

53280 $D020 EXTCOL

Border color register

The value in this register determines the color of the screen border, the area of the screen surrounding the active portion

of the display. When the screen is blanked by setting bit 4 of the register at 53265/$D011 to %0, the entire display area will be filled with the color specified here. Since the VIC can produce only 16 different colors, only four bits are required to hold all possible color values. Thus, only bits 0-3 of this location hold meaningful values. Bits 4-7 are unused; writing to those bits has no effect, and the bits always return %1 when read. Thus, the value in this register will always be at least 240/$F0. To mask off these bits and read the true color value you should use AND 15 in BASIC or AND #$0F in machine language. See Table 8-1 for a list of standard VIC color values. The default setting for this register, established by the IOINIT routine [$E1O9], is 13/$0D (light green). From BASIC, the value here can be changed using the statement COLOR 4,rc (where *n* is the desired BASIC color number; BASIC color numbers are equal to VIC color values plus one).

53281 **$DO21** **BGCOLO**

Background color register 0

In standard character mode, the value here determines the color of all pixels in a screen position which are represented by %0 bits in the character pattern. This background color is common to all screen positions. Clearing the screen fills all positions with the screen code for the space character, which has a pattern that is all %0 bits. Thus, a cleared screen will be filled with the color specified here. The color of pixels with %1 bits in the character pattern (the foreground color) is determined by the value in the color memory location for the position.

This register is also used in both of the other character modes. In multicolor character mode, the value here determines the color of all pixels in all screen positions represented by %00 bit pairs in the character pattern. In extended background color mode, the value here determines the background color in any screen positions for which bits 6-7 of the screen code in the corresponding screen memory location are %00. This register is unused in standard bitmapped mode, but for multicolor bitmapped mode the value here determines the color for all pixels represented in the bitmap by %00 bit pairs.

Since the VIC can produce only 16 different colors, only four bits are required to hold all possible color values. Thus, only bits 0-3 of this location hold meaningful values. Bits 4-7 are unused; writing to those bits has no effect, and the bits al-

ways return %1 when read. Thus, the value in this register will always be at least 240/$F0. To mask off these bits and read the true color value you should use AND 15 in BASIC or AND #$0F in machine language. See Table 8-1 for a list of standard VIC color values. This register is initialized to 11/$0B (dark gray) by the IOINIT routine [$E109], part of both the reset and RUN/STOP-RESTORE sequences. From BASIC, the value here can be changed using the statement COLOR *0,n* (where *n* is the desired BASIC color number; BASIC color numbers are equal to VIC color values plus one.)

53282	$D022	BGCOL1
53283	$D023	BGCOL2
53284	$D024	BGCOL3

Background color registers 1-3

These three registers are used only when multicolor character mode or extended background color mode is enabled. For multicolor character mode, the value in 53282/SD022 determines the color of any pixels in a screen position represented by %01 bit pairs in the character pattern, and the value in 53283/SD023 determines the color of all pixels with %10 pairs. Since there is only one set of registers for all screen positions, the colors of pixels with %01 and %10 bit pairs will be common to all characters on the screen. The color of pixels with %00 bit pairs is determined by the value in 53281/SD021, and the color of pixels with %11 bit pairs is selected individually for each character position by the value in the corresponding color memory location (55296-56319/ $D800-$DBFF). The register at 53284/$D024 is unused in multicolor character mode,

For extended background color mode, these registers determine the background color for the character position for the character, depending on the setting of bits 6-7 of the screen code in the screen memory location for the position. The selections are as follows:

Bits 7 6	Background color source register
0 0	53281/SD021
0 1	53282/$D022
1 0	53283/$D023
1 1	53284/$D024

Since the VIC can produce only 16 different colors, only four bits are required to hold all possible color values. Thus, only bits 0-3 of these locations hold meaningful values. Bits 4-7 in any of these locations are unused. Writing to those bits has no effect, and the bits always return %1 when read. Thus, the value in any of these locations will always be at least 240/$F0. To mask off these bits and read the true color value you should use AND 15 in BASIC or AND #$0F in machine language. See Table 8-1 for a list of standard VIC color values. These registers are initialized during the IOINIT routine [$E019], part of the reset and RUN/STOP-RE STORE sequences. The default values are 1 (white) for 53282/$D022, 2 (red) for 53283/$D023, and 3 (cyan) for 53284/$D024. There is no BASIC statement specifically for changing the settings of these registers.

53285 $D025 SPMCO
53286 $D026 SPMC1
Sprite multicolor registers

The values in these registers determine two of the colors for multicolor sprites. (A sprite can be switched to multicolor mode by setting the appropriate bit in the register at 53276/$D01C.) The value in the register at 53285/$D025 determines the color for all pixels in the sprite represented by %01 bit pairs in the pattern. The value in 53286/$D026 determines the color for all pixels represented by %11 bit pairs. Since there are only these two registers for all eight sprites, the colors of pixels with %01 and %11 bit pairs will be common to all sprites. The color of pixels represented by %10 bit pairs can be selected individually for each sprite using the registers at 53287-53294/$D027-$D02E. Pixels with %00 bit pairs in the pattern will be transparent.

Since the VIC can produce only 16 different colors, only four bits are required to hold all possible color values. Thus, only bits 0-3 of these locations hold meaningful values. Bits 4-7 in any of these locations are unused. Writing to those bits has no effect, and the bits always return %1 when read. Thus, the value in any of these locations will always be at least 240/$F0. To mask off these bits and read the true color value you should use AND 15 in BASIC or AND #$0F in machine language. See Table 8-1 for a list of standard VIC color values. The default settings of these registers, established by the

IOINIT routine [$E109], are 1 (white) for 53285/$D025 and 2 (red) for 53286/$D026. From BASIC, these settings can be changed using the SPRCOLOR statement.

53287	SD027	SPOCOL
53288	SD028	SP1COL
53289	SD029	SP2COL
53290	$D02A	SP3COL
53291	8D02B	SP4COL
53292	$D02C	SP5COL
53293	$D02D	SP6COL
53294	$D02E	SP7COL

Sprite color registers

Each of these registers holds color information for one of the eight sprites. For standard sprites, the value here determines the color of all %1 bits in the sprite pattern (%0 bit positions will be transparent). For multicolor sprites, the value here determines the color of all %10 bit groups in the pattern.

Since the VIC can produce only 16 different colors, only four bits are required to hold all possible color values. Thus, only bits 0-3 of these locations hold meaningful values. Bits 4-7 in any of these locations are unused. Writing to those bits has no effect, and the bits always return %1 when read. Thus, the value in any of these locations will always be at least 240/$F0. To mask off these bits and read the true color value you should use AND 15 in BASIC or AND #$0F in machine language. See Table 8-1 for a list of standard VIC color values. The following table shows the default colors for each sprite, established by the IOINIT routine [$E109], part of both the reset and RUN/STOP-RESTORE sequences.

Register	**Sprite**	**Default color**
53287/$D027	**0**	0 (black)
53288/$D028	**1**	**1 (white)**
53289/$D029	**2**	**2 (red)**
53290/$D02A	3	3 (cyan)
53291/$D02B	**4**	**4 (pmple)**
53292/$D02C	**5**	5 (green)
53293/$D02D	**6**	6 (blue)
53294/$D02E	**7**	7 (yellow)

53295 $D02F XSCAN
Extended keyboard scan-line control register

Bits 0-2: Each of these bits controls the state of one of the three output lines from the VIC chip, K0-K2. Setting a register bit to %0 causes the corresponding output line to go to a low (0 volts) state, while setting a register bit to %1 causes the output line to go to a high (+ 5 volts) state. Reading these bits returns the current state of the corresponding output lines. In the 128, the three output lines are used to scan the three keyboard columns containing the 24 keys in the numeric keypad and top row of control keys. See Figure 7-1 and the discussion of the keyboard scanning routine [$C55D] in Chapter 7 for more information.

Unless you are writing a custom keyboard scanning routine, there's rarely a need to tinker with this register. The output lines are connected only to the keyboard, and are not available externally.

Bits 3-7: These bits are unused. Writing to them has no effect, and they always return %1 when read. Thus, the value in this location will always be at least 248/$F8. To mask off these irrelevant bits, use AND 7 in BASIC or AND #$07 in machine language.

53296 $D030 CLKRATE
Processor clock rate control register

Bit 0: This bit controls the processor clock speed. (Remember, the VIC chip is the source of most of the system's timing signals.) When the bit is set to %0, the processor operates at its normal 1-MHz rate. To be precise, the clock frequency is 1.02273 MHz for NTSC (North American) systems and 0.98525 MHz for PAL (European) systems. Setting this bit to %1 doubles the clock rate, providing what is commonly referred to as 2-megahertz (MHz) mode. This is also known as fast mode, the old standard speed being disparagingly referred to as slow mode. During the reset and RUN/STOP-RESTORE sequences, the IOINIT routine [$E109] sets this bit to %0 for slow mode.

Fast mode does have a few limitations. While the 8502 microprocessor and the VDC 80-column video chip have no problems operating at the higher clock rate, most of the other I/O chips cannot keep up at this speed. The VIC chip itself cannot maintain its video display at this speed—the 40-column

screen becomes a colorful pattern of rapidly flashing squares. It is common practice to set bit 4 of the VIC register at 53265/$D011 to %0 to blank the 40-column screen display while operating in 2-MHz mode. For example, the BASIC routine for the FAST statement [$77B3] includes this step. The VDC provides an alternative to the VIC for fast mode, but other I/O chips have no substitutes. In these cases, the system employs an elaborate technique known as clock stretching, where the clock period is extended to create an effective 1-MHz rate for the portion of the clock cycle when the I/O chip is being accessed.

Because some serial bus and tape communications routines depend on software loops for timing functions, the system is usually switched to the slower clock frequency when serial bus or tape operations are being performed. The contents of this register are stored in location 2615/$0A37 during the operation, and restored to the register when the operation is completed. You can prevent this by setting bit 7 of the custom mode flag (location 2618/$0A3A) to %1. In this case, the clock rate will not be changed during tape and serial operations.

Bit 1: This bit is described in Commodore literature as a test bit. The IOINIT routine [$E019] sets the bit to %0, and no other 128 ROM routine changes that setting. Some programmers have discovered that setting this bit to %1 will blank the 40-column screen display, and have even used this as an alternative to clearing bit 4 of location 53265/$D011 when switching the processor to fast mode. While this does appear to work without side effects, such undocumented "features" are best avoided.

Bits 2-7: These bits are unused. Writing to them has no effect, and they always return %1 when read. Thus, the value in this register will always be at least 252/$FC. To mask off these bits, use AND 3 in BASIC or AND #$03 in machine language.

53297-53311 $D031-$D03F Unused

These unused register addresses always return the value 255/$FF when read. Writing to these locations has no effect.

53312-54271 $D040-$D3FF

VIC chip register images

Due to incomplete address decoding, images of the VIC chip registers repeat every 64 bytes through the remainder of this page of memory. That is, storing a value in any location in this range which has an address that is an exact multiple of 64 greater than one of the base register locations has the same effect as storing that same value at the base register location. For example, storing a value at 53312/$D040 or 54208/$D3C0 has the same effect as storing a value in 53248/$DOOO. However, it's better programming practice to use the officially designated register addresses.

SID (Sound Interface Device) Chip Registers

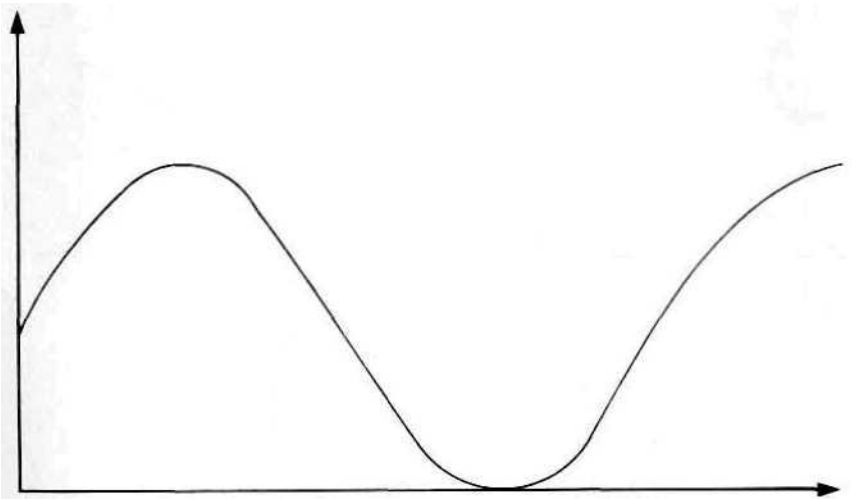
S4272-54300/$D400-$D41C

The SID (Sound Interface Device) chip, officially designated the 6581, was a remarkable breakthrough when it was first introduced in the Commodore 64. It remains the most sophisticated piece of standard audio hardware in any currently available home computer. The SID incorporates most of the features of a complete sound synthesizer in a single chip, and it has been a key to the Commodore 64's success as a music machine.

Some sound fundamentals: Most sounds we hear are transmitted in the form of pressure waves through the air. The sounds humans usually consider pleasant have regularly repeating patterns. To duplicate (synthesize) these sounds, an electronic device such as the SID chip generates patterns of electrical signals that are passed through an amplifier to a speaker, which translates the signals into corresponding sound waves. The most rudimentary characteristic of a sound wave is its frequency, the measure of how many times per second the bask sound pattern repeats. In music, frequency is expressed as pitch. The higher the frequency of a sound wave, the higher its pitch. Frequency is measured in units of cycles per second, called hertz (Hz). The generally accepted range of frequencies audible to humans is 20 Hz to 20,000 Hz. Any computer that provides for sound output will allow you to control the frequency of the sound. For those computers like the IBM and Apple that have only rudimentary sound capabilities, frequency is the only component you can control.

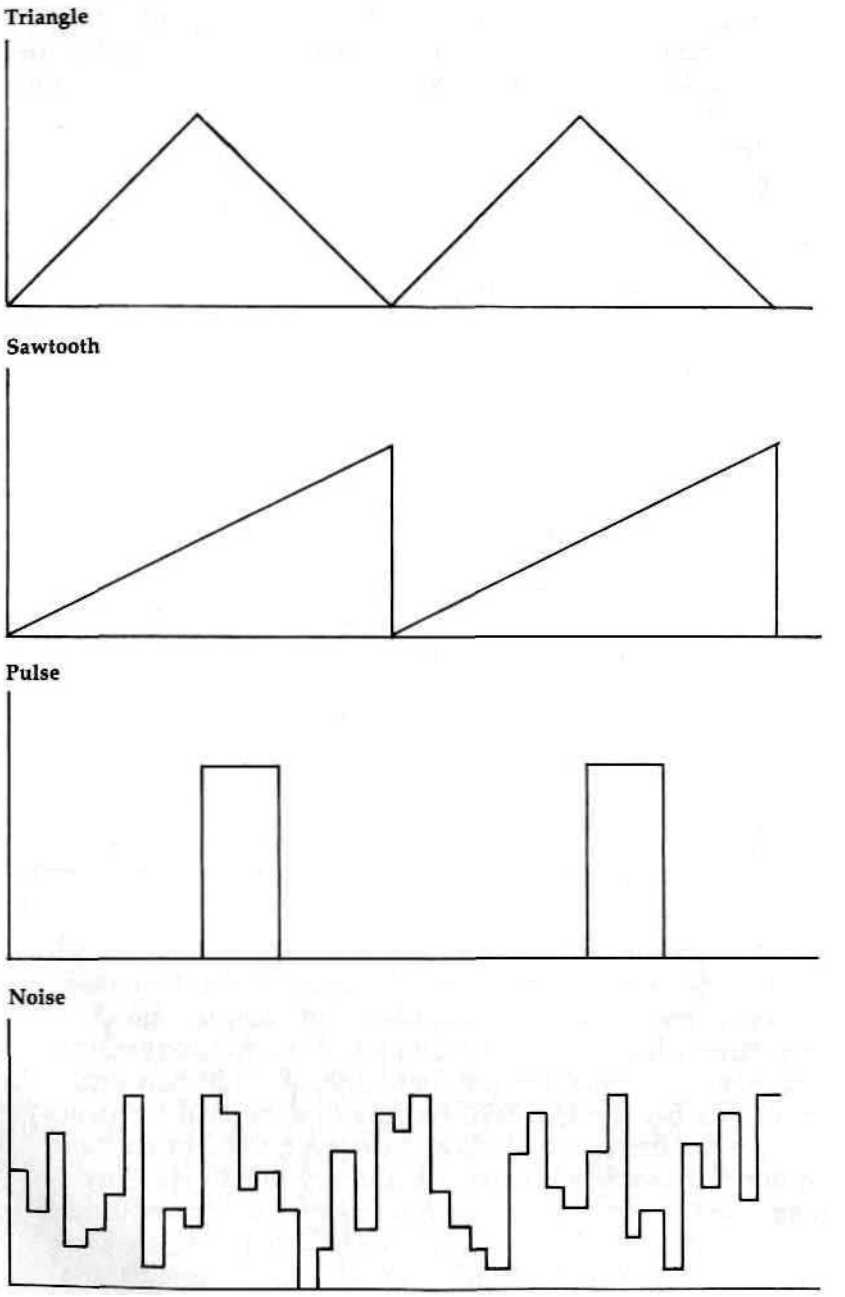
In the common musical scale, every note is assigned a particular frequency (see Appendix D). But you don't have to know much about music to know that a C note played on a guitar sounds different from one played on a piano or flute or organ. Obviously, there is more to sound than just frequency. The next important characteristic is waveform. The sound wave for a "pure" tone will have a sinusoidal waveform, as shown in Figure 8-7. However, a sine waveform is relatively difficult to synthesize digitally, so it is fortunate that pure tones are relatively rare.

Figure 8-7. Sinusoidal Waveform



Most sounds, including the sounds of almost all musical instruments, contain *harmonics* in addition to the fundamental frequency. Harmonics are components of complex sound waveforms which are exact multiples of the fundamental frequency. For example, the first harmonic of a 120-hertz wave will have a frequency of 120 Hz (the fundamental frequency). The second harmonic will have a frequency of 240 Hz, and the third harmonic will have a frequency of 360 Hz. Any wave shape can be expressed in terms of a fundamental sine wave and additional harmonics. The SID can generate three different wave shapes: the triangle, which corresponds to a

Figure 8-8. SID Waveforms



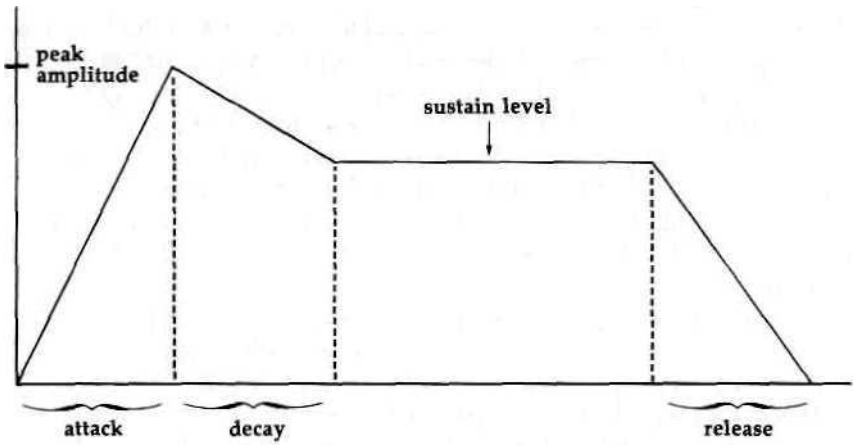
fundamental frequency plus odd harmonics in diminishing proportions; the sawtooth waveform, which corresponds to a fundamental frequency plus both odd and even harmonics in diminishing proportions; and the pulse waveform, which has a varying mix of harmonics depending on the width of the pulse. Figure 8-8 illustrates these waveforms. The triangle waveform corresponds to mellow sounds like those of the xylophone. The sawtooth waveform corresponds to the slightly harsher sound of instruments like the guitar or accordion. Because the harmonic content of the pulse waveform is variable, it can be used for a variety of sounds ranging from piano to trumpet.

Note that Figure 8-8 includes a fourth waveform not previously mentioned. In addition to the "orderly" waveforms, the SID can also produce a waveform that varies constantly with no discernible frequency. Such a pattern (or lack of pattern) is characteristic of the class of sounds we call noise. Depending on how fast the wave changes levels, it can range from a low buzz to a high hiss like radio station static.

The final component of sound that can be controlled by the SID is the *envelope.* If you visualize the sound waveforms as shown in Figure 8-8, the volume of the sound corresponds to the height of the waveform, technically called the amplitude of the wave. While some instruments like the organ can start a note playing at a constant amplitude (volume) and turn it off almost immediately, most instruments take a certain amount of time to bring a note to full amplitude, and in some instruments a note will linger for a brief period after it has been played. This rise and fall of amplitude is called the envelope of the waveform, and is usually described in terms of attack, decay, sustain, and release (ADSR), as illustrated in Figure 8-9, Each class of instrument has a characteristic ADSR envelope.

The attack is the time required for the note to rise from silence to maximum volume after it is begun—for example, after a string is picked or bowed or struck. Decay is the time required for the note to drop from maximum volume to its sustain level, where it remains until it begins to die away to silence again (the release phase). Not every sound will exhibit every phase of the ADSR envelope. For example, the envelope for instruments like the guitar or piano which have plucked or struck strings will have almost no attack time; the envelope for

Figure 8-9. ADSR Envelope



instruments like the flute, where the player must start a column of air vibrating, have a significant attack time. The discussion of the ENVELOPE statement in the *System Guide* that came with your 128 shows the relative ADSR parameters of a variety of instrument types. If you are confused about the relationship between frequency, waveform, and envelope, Figure 8-10 should help clear up some of the confusion.

Table 8-3 lists the available registers of the SID chip. A detailed description of each follows. The SID's 29 registers fall into two distinct classes. The first 25 are write-only; reading from them returns zeros or meaningless values. The final 4 are read-only; writing to them has no effect. All 128 SID registers appear in the same locations and have the same functions as the Commodore 64's SID chip. Thus, any Commodore 64 sound routine should also work in Commodore 128 mode. In addition to being loaded directly, some of these registers can also be loaded indirectly from shadow registers as part of the BASIC IRQ service routine.

Figure 8-10. Sound Characteristics: Frequency, Waveform, and Envelope

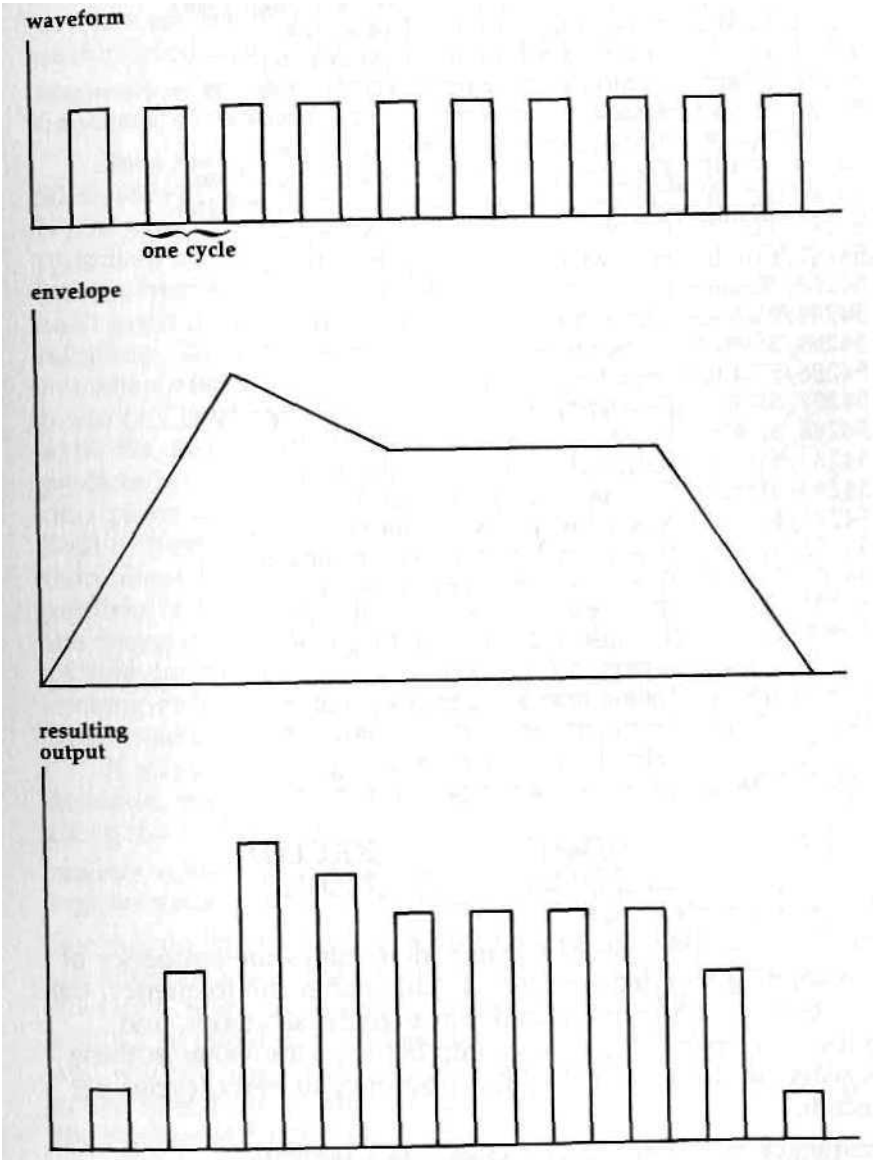


Table 8-3. SID Chip Registers

54272/$D400	Frequency register for voice 1 {low byte)
54273/$D401	Frequency register for voice 1 (high byte)
54274/$D402	Pulsewidth for voice 1 (low byte)
54275/$D403	Pulsewidth for voice 1 {high byte)
54276/$D404	Control register for voice 1
54277/$D405	Attack/decay register for voice 1
54278/$D406	Sustain/release register for voice 1
54279/SD407	Frequency register for voice 2 (low byte)
54280/$D408	Frequency register for voice 2 (high byte)
54281/$D409	Pulsewidth for voice 2 {low byte)
54282/$D40A	Pulsewidth for voice 2 {high byte)
54283/$D40B	Control register for voice 2
54284/$D40C	Attack/decay register for voice 2
54285/$D40D	Sustain/release register for voice 2
54286/$D40E	Frequency register for voice 3 (low byte)
54287/$D40F	Frequency register for voice 3 (high byte)
54288/$D410	Pulsewidth for voice 3 {low byte)
54289/$D411	Pulsewidth for voice 3 (high byte)
54290/$D412	Control register for voice 3
54291/$D413	Attack/decay register for voice 3
54292/$D414	Sustain/release register for voice 3
54293/SD415	Filter cutoff frequency {low byte)
54294/$D416	Filter cutoff frequency (high byte)
54295/$D417	Resonance/filter control register
54296/$D418	Volume/filter mode register
54297/$D419	Potentiometer (paddle) x position
54298/$D41A	Potentiometer (paddle) y position
54299/$D41B	Voice 3 oscillator output
54300/$D4lC	Voice 3 envelope generator output

54272 $D400 FRELO1
54273 $D401 FREHI1
Frequency control registers

The value in this pair of registers determines the frequency of the sound generated for voice 1. The higher the frequency, the higher the pitch of the sound. For triangle, sawtooth, and pulse waveforms, the relationship between the value in these registers and the voice 1 sound frequency in hertz (cycles per second) is:

frequency = register value * clock rate / 16777216

The first register (54272/$D400) holds the low byte of the value and the second (54273/$D401) holds the high byte. The

clock rate depends on which video system the 128 uses. For NTSC (North American) systems, the clock value is 1022730 Hz, while for PAL (European) systems it is 985250 Hz. Thus, you can use the following expressions to calculate the frequency produced by any given register setting:

frequency = register value * 0.06096 (for NTSC systems)
frequency = register value * 0.05873 (for PAL systems)

Since the register pair can hold values from 0-65535/ $0000-$FFFF, the range of possible output frequencies for an NTSC system is 0-3995 Hz. A register value of zero corresponds to no frequency, hence no sound. The upper limit of human hearing is about 20,000 Hz, but the fact that the SID can't generate frequencies above 4000 Hz isn't really a serious handicap. The frequencies between 4000 and 20,000 Hz are not often used in music. For example, the highest note on a piano (a C four octaves above middle C) has a frequency of 4186 Hz, just slightly beyond the SID's range. The SID can produce notes corresponding to those for any of the 88 keys on a piano keyboard except the very rightmost one. The lower limit of human hearing is about 20 Hz, so register values less than about 320/$0140 should result in inaudible output regardless of the volume setting. Actually, this is true only for the triangle waveform. The sudden transitions in output amplitude during sawtooth and pulse waveforms will produce a clicking or buzzing output at these supposedly inaudible low frequencies.

If you know the frequency (in hertz) that you wish to generate, you can calculate the corresponding register setting using the following formulae:

register value = desired frequency * 16.40 (for NTSC systems)
register value = desired frequency * 17.03 (for PAL systems)

See Appendix D for a list of the standard frequencies for musical notes and the register values required to produce those frequencies. It is permissible to change the frequency of a sound while it is playing. The change will take effect immediately, and will not affect the envelope. Changing the frequency of an active sound can produce interesting effects. See the entry for the register at 54299/$D41B for an example.

The noise waveform doesn't have a regularly repeating pattern like the triangle, sawtooth, and pulse waveforms. The output jumps erratically from one amplitude level to another,

so noise doesn't really have a frequency in the same sense that the other waveforms do. For noise, the value in these registers determines how rapidly the amplitude level changes. The number of changes per second is approximately equal to the register value. For example, a register value of 1000/$03E8 will cause the noise output level to change about 1000 times per second, or approximately one change every 1/1000 second. The faster the output level changes, the higher the perceived pitch of the generated noise.

Remember that these are write-only registers; they will always return 0/$00 when read, regardless of the values you have stored in them.

54274 $D402 PWLO1
54275 SD403 **PWHI1**
Pulsewidth control registers

This register pair controls the waveform shape when pulse output is selected for voice 1. The value here has no effect on any other waveform. The pulse waveform is binary; that is, it has two states: maximum amplitude and off {no amplitude). Unlike the ideal sine waveform, which has a smooth transition between maximum and minimum amplitudes, the pulse waveform switches almost instantly. The duration of each cycle of the waveform is controlled by the registers at 54272-54273/ $D400-$D401. What the registers here control is how much of each cycle the waveform spends in the zero-amplitude state.

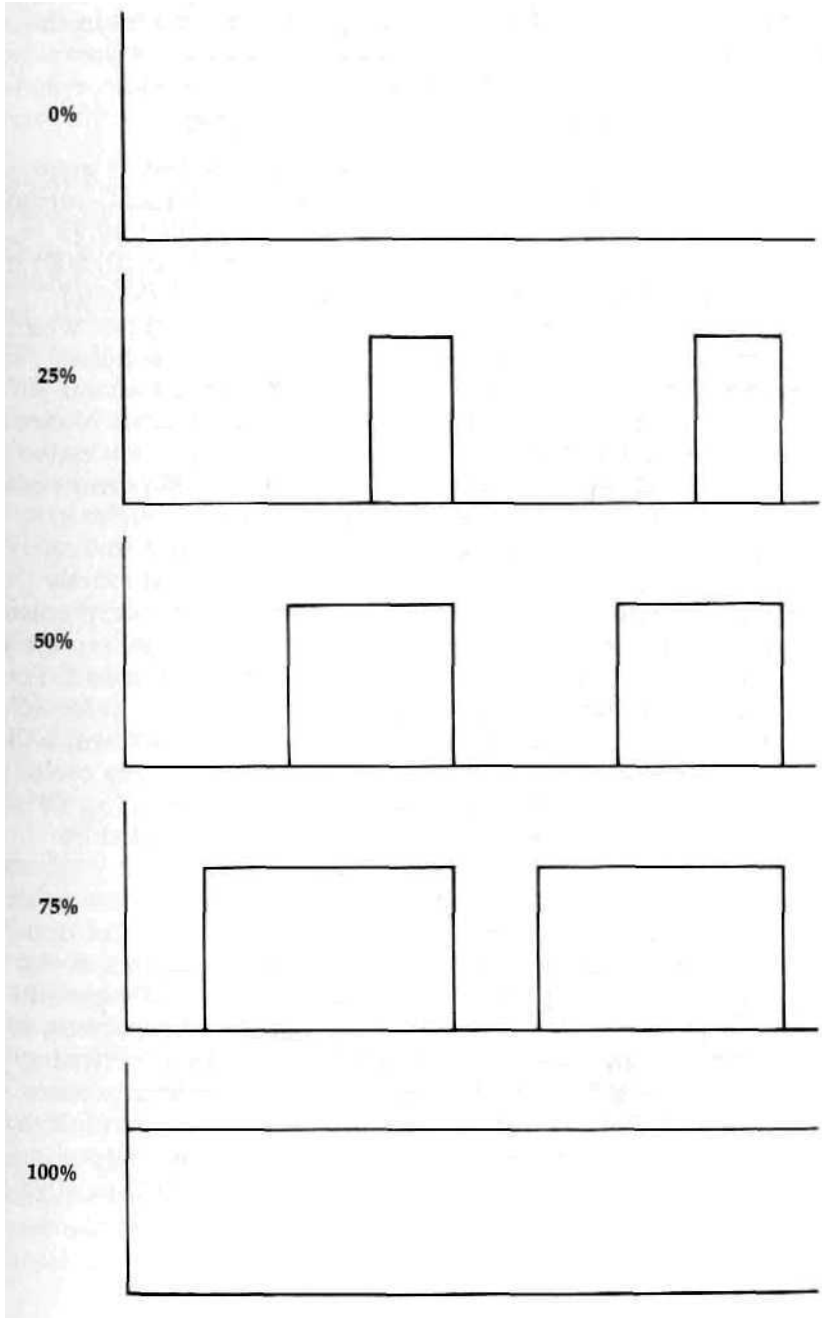
Pulse waveforms are often described in terms of their duty cycles, the percentage of the total waveform cycle spent in the maximum amplitude state. A pulse waveform with a 0-percent duty cycle is always off, while a pulse waveform with a 100-percent duty cycle is always at maximum amplitude. A waveform with a 50-percent duty cycle is at maximum amplitude for half of the cycle and off for the remaining half, resulting in a square wave. Figure 8-11 illustrates various duty cycles.

This register pair controls the duty cycle (expressed as a percentage of the total duration of one cycle of the waveform) according to the following formula:

duty cycle = 100 — (register value / 40.95)

Only 12 of the 16 bits in the register pair are used. The lower 8 bits come from the register at 54274/$D402 and the higher 4 bits come from bits 0-3 of the register at 54275/$D403. Bits

Figure 8-11. Pulse Waveform Duty Cycles



4-7 of 54275/$D403 are unused; any value written to these bits has no effect. The available 12 bits allow register values of 0-4095/$000-$EFF, Thus, you can specify duty cycles in the full range 0-100 percent. If you know the duty cycle you want, you can calculate the proper register value with:
register value = (100 — desired duty cycle) * 40.95

In the expression above, the duty cycle is expressed as a percentage. For example, the required register value for a 50-percent duty cycle—a square wave—would be (100 — 50) * 40.95 = 2048/$0800. For this, you would store the low byte (0/$00) in 54274/$D402 and the high byte (8/$08) in 54275/$D403.

There's one phemomenon you need to be aware of when selecting duty cycles. The relative percentage of the pulse waveform cycle spent in each state, rather than the actual state, determines how the resulting output will sound. Notice in Figure 8-11 that both the 25-percent and 75-percent duty cycles have waveforms that are in one state for 25 percent of the cycle and the other state for 75 percent of the cycle. In either case, the ratio of time spent in each state is 3 to 1. A pulse wave with a 75-percent duty cycle will sound exactly the same as one with a 25-percent duty cycle. For every pulse waveform duty cycle less than 50 percent, there is a duty cycle greater than 50 percent that will produce the same sound. For example, a waveform with a 10-percent duty cycle (on for 10 percent of the cycle and off for the remaining 90 percent) will sound the same as a waveform with a 90-percent duty cycle (on for 90 percent of the cycle and off for the remaining 10 percent), since for either waveform the cycle is divided between the two states in a 9-to-l ratio.

The closer the register value is to 2048/$0800—the value for a square wave (50-percent duty cycle, 1-to-l ratio of time in each state)—the richer the resulting output will sound. As the register value approaches 4095 or 0, for 0- or 100-percent duty cycles, respectively, the ratio approaches its maximum of 1 to 4095 and the resulting sound output becomes increasingly thin. Very low or very high duty cycles result in nearly inaudible output. Duty cycles of exactly 0 or 100 percent result in constant output levels. Since some variation in the output is required to produce sound, register values of 0/$000 or 4095/$FFF produce no audible output.

It is permissible to change the value in these registers, and hence the width of a pulse waveform, while the voice is generating output. This will affect only the waveform, not the envelope. Remember that these are write-only registers; they will always return 0/$00 when read, regardless of the values you have stored in them.

54276 8D404 VCREG1

Voice 1 control register

Each bit of this register controls some aspect of the output sound for the voice. Remember that this is a write-only register. All of the following bits return %0 when read.

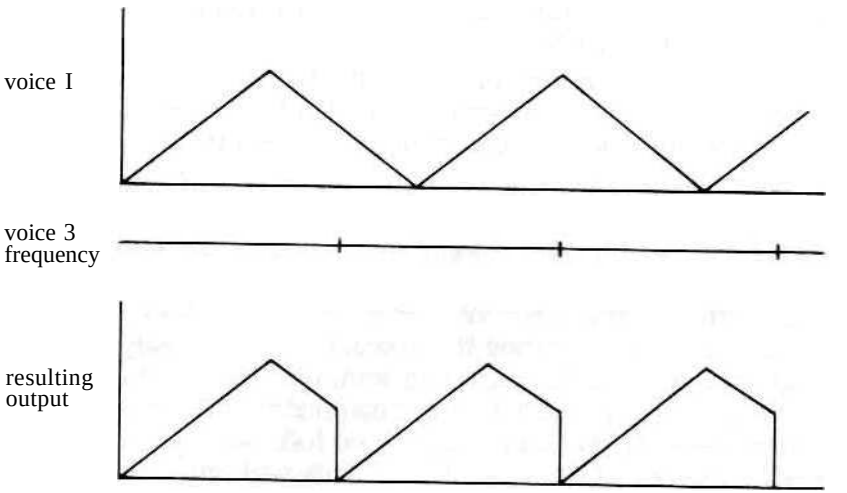
Bit 0: This bit, called the *gate* flag, is used to initiate output for this voice and to trigger the release portion of the defined sound envelope for the voice. (See the introduction of this section for more information on sound envelopes.) Writing a %1 here starts the attack portion of the envelope. This is sometimes referred to as gating the voice. For audible output, the frequency registers for this voice and the overall volume register for the chip must contain nonzero values. The attack portion of the envelope will normally be followed by the decay phase, after which the sound amplitude will remain at the specified sustain level until a %0 is written to this register. After the %0 is written, the release phase begins and the sound output level will fade away to silence at the specified release rate. (Release does not occur unless a %0 is specifically written to this bit.)

Note that writing a %0 here triggers the release phase regardless of the current state of the envelope. For example, if you write a %1 here and then immediately write a %0 before the attack phase is completed, the attack will be aborted and release will begin from the current amplitude level. Likewise, writing a %1 here initiates the attack phase, regardless of the current state of the envelope. For example, if a %1 is written here while the envelope for the voice is in the decay phase, the decay phase will be aborted and another attack phase will begin from the current amplitude level.

Bit 1: This bit, called the *sync* flag, controls a special effect known as synchronization, which changes the frequency of and adds extra harmonics to the voice 1 output. When this bit is set to %1, the waveform of voice 1 will be synchronized

with the waveform of voice 3. That is, whenever the waveform of voice 3 starts a new cycle, voice 1 will also start a new cycle, regardless of its point in its current waveform. Figure 8-12 illustrates the effect on the voice 1 waveform.

Figure 8-12. Voice 1 Synchronized with Voice 3



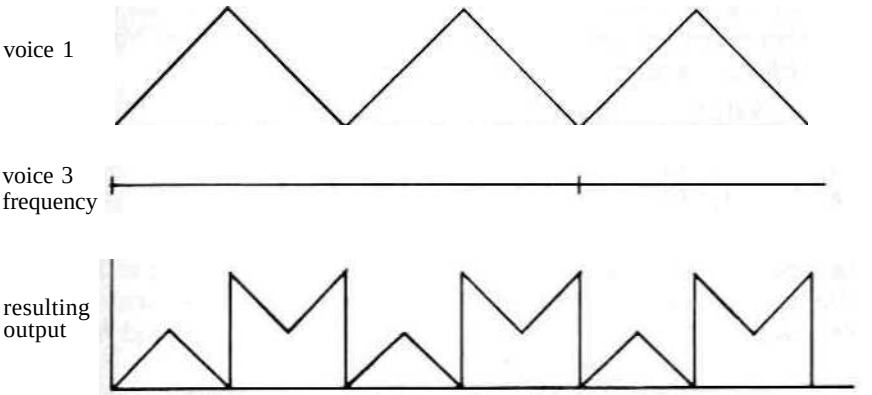
Only the frequency specified for voice 3 is significant for synchronization. In fact, it doesn't matter whether voice 3 is turned on or not, so long as a frequency value is stored in the frequency control registers for the voice. (Synchronization has no effect if the frequency for the synchronizing voice is zero.) Ideally, the frequency for voice 3 should be less than that specified for voice 1.

Synchronization also works for voices 2 and 3. Setting this bit in the control register for voice 2 (54283/$D40B) allows voice 2 to be synchronized with voice 1, and setting this bit in the control register for voice 3 (54290/$D412) allows voice 3 to be synchronized with voice 2. It is possible to have two or all three voices synchronized simultaneously.

Bit 2: This bit, called the *ring mod* flag, controls a special effect known as ring modulation. When this bit is set to %1 and a triangle waveform is selected for the voice, the triangle waveform will be ring modulated at the frequency of voice 3. (Ring modulation works only when the triangle waveform is selected.)

Ring modulation is rather difficult to explain simply. When you ring modulate voice 1 at the frequency of voice 3, the resulting waveform is the equivalent to the sum of two waveforms of different frequencies. However, the resulting frequencies are different from both the frequency of voice 1 and that of voice 3. Figure 8-13 is an example of one observed ring-modulated output.

Figure 8-13. Voice 1 Ring Modulated by Voice 3



The ring modulated waveform will have a complex mix of harmonics. This is useful for simulating the sounds of bells, gongs, and similar instruments whose waveforms don't closely resemble the triangle, sawtooth, or pulse.

Only the frequency specified for voice 3 is significant for ring modulation, (Voice 1 must be set for a triangle waveform, but the waveform of voice 3 is irrelevant.) It doesn't even matter whether voice 3 is turned on or not, as long as a frequency value is stored in the frequency control registers for the voice. (Ring modulation has no audible effect if the frequency for the modulating voice is zero.)

Ring modulation also works for voices 2 and 3. Setting this bit in the control register for voice 2 (54283/$D40B) allows voice 2 to be ring modulated by voice 1. (Voice 2 must be set for a triangle waveform.) Setting this bit in the control register for voice 3 (54290/$D412) allows voice 3 to be ring modulated by voice 2, (Voice 3 must be set for a triangle waveform.) It is possible for two or all three voices to be simultaneously ring modulated.

Bit 3: This bit, called the *test* flag, can be used to reset the internal oscillator for the voice. When this bit is set to %1, the internal-oscillator register is reset to zero, halting sound output for the voice. The oscillator remains stopped until a %0 is written to this bit. You can use this bit to precisely control when the voice oscillator turns off or on. You must also use this bit to unfreeze the noise output if you accidentally turn on noise while another waveform is active.

Bits 4-7: These bits control the behavior of the internal oscillator for the voice, and hence the resulting waveform of the sound output for the voice. (See the introduction to this section for more information on waveforms.) Each bit controls one of the standard waveforms, as follows:

Bit	Value	Waveform
4	16/$10	triangle
5	32/$20	sawtooth
6	64/$40	pulse
7	128/$80	noise

Setting one of these bits to %1 tells the oscillator to produce the corresponding waveform. Setting the bit to %0 turns off that waveform. One of the waveforms must be selected for the voice to produce any output.

Since the waveforms can be independently selected, you might be tempted to simultaneously enable more than one waveform for the voice. This won't hurt the SID chip, but you'll probably be disappointed with the results. When you select more than one waveform, the resulting output is not the simple combination of the selected waveforms. Commodore literature continues to claim that the result will be a logical ANDing of the selected waveforms, but the SID's designer has stated that this is not the case. In any event, mixed waveforms tend to produce a rather erratic sound, so the technique isn't really useful. Furthermore, you'll cause a problem if one of the waveforms in the combination is noise. When the noise waveform is selected while any other waveform is also selected, the noise generator for the voice will cease to function. To restart it you'll have to write a %1 and then a %0 to the TEST flag (bit 3 of this register) or reset the computer.

54277 $D405 ATDCY1

Attack and decay control register

This register controls the behavior of the attack and decay phases of the envelope for the voice.

Bits 0-3: These bits control the decay rate, the amount of time required for the voice to drop from the peak amplitude attained during the attack phase to the specified sustain level. For decay to have any audible effect, the sustain level must be less than %1111/$F. There is no simple formula relating the bit value to the corresponding time. The following table shows the relationship:

Bits	Value	Time required for decay phase (in seconds)
32 10		
0 0 0 0	0/$00	0.006
0 0 0 **1**	**l/$01**	0.024
0 0 10	2/$02	0.048
0 0 11	3/$03	0.072
0 10 0	4/$04	0.114
0 10 1	5/$05	0.168
0 1 1 0	6/$06	0.204
0 1 1 1	7/$07	0.240
10 0 0	8/$08	0.300
10 0 1	9/$09	0.750
10 10	10/$0A	1.50
10 11	11/S0B	2.40
1 1 0 0	12/$0C	3.00
1 1 0 1	13/$0D	9.00
1 1 1 0	14/$0E	15.0
1 1 1 1	15/$0F	24.0

You can change the decay rate while a sound is being played. However, unless the envelope is currently in the attack or decay phase, the change won't have any effect until the next time the envelope is started by writing a %1 to the gate bit for the voice (bit 0 of the control register).

Bits 4-7: These bits control the attack rate, the amount of time required for the sound output of the voice to rise from silence (zero amplitude) to peak amplitude. The attack phase begins when a %1 is written to the gate bit for the voice (bit 0 of the control register). There is no simple formula relating bit values to the corresponding attack rates. The following table shows the relationship. Note that attack rates are three times faster

than corresponding decay or release rates. This is because the attack phase tends to be shorter than decay or release for most naturally occurring sounds.

Bits 7 6 5 4	Value	Time required for attack phase (in seconds)
0 0 0 0	0/$00	0.002
0 0 0 1	16/$10	0.008
0 0 1 0	32/$20	0,016
0 0 1 1	48/$30	0.024
0 1 0 0	64/$40	0.038
0 1 0 1	80/$50	0.056
0 1 1 0	96/560	0.068
0 **1 1 1**	112/$70	0.080
1 0 0 0	128/$80	0.100
1 0 0 **1**	144/$90	0.250
1 0 1 0	160/$A0	0.500
1 0 **1 1**	176/$B0	0.800
1 1 0 0	192/$C0	1,00
1 1 0 1	208/$D0	3.00
1 1 1 0	224/$E0	5.00
1 1 1 1	240/$F0	8.00

Yoti can change the attack rate while a sound is being played. However, unless the envelope is currently in the attack phase, the change won't have any effect until the next time the[1] gate bit for the voice (bit 0 of the control register) is set to %1 to restart the envelope.

54278 SD406 SUREL1

Sustain and release control register

This register controls the behavior of the sustain and release phases of the envelope.

Bits 0-3: The value in these bits determines the amount of time required for the volume level for the voice to drop to zero (silence) during the release phase of the envelope. The release phase doesn't begin until it is triggered by writing a %0 to bit 0 of the control register for the voice. Note that release will have no audible effect if the specified sustain level is zero. There's no simple formula relating the value in these bits to the corresponding release times. The following table lists the relationships:

Bits 3 2 1 0	Value	Time required for release phase (in seconds)
0 0 0 0	0/$00	0.006
0 0 0 **1**	l/$01	0.024
0 0 1 0	2/$02	0.048
0 0 **1 1**	3/$03	0.072
0 1 0 0	4/$04	0.114
0 1 0 1	5/$05	0.168
0 1 1 0	6/$06	0.204
0 1 1 1	7/S07	0.240
1 0 0 0	8/$08	0.300
1 0 0 1	9/$09	0.750
1 0 1 0	10/$0A	1.50
1 0 **1 1**	ll/$0B	2.40
1 1 0 0	12/$0C	3.00
1 1 0 1	13/$0D	9.00
1 1 1 0	14/$0E	15.0
1 1 1 1	15/$0F	24,0

It is possible to change the release rate while a sound is being played. The new rate will supersede the old one, even if the envelope is currently in the release phase.

Bits 4-7: These registers specify the volume level at which the voice output will be maintained during the sustain level of the envelope. Note that this is different from the attack, decay, and release values, which specify periods of time instead of levels. Once the attack and decay phases are completed, the

Bits 7 6 5 4	Value	Percentage of peak output
0 0 0 0	0/$00	0 (no output)
0 0 0 1	16/$10	7
0 0 1 0	32/$20	13
0 0 1 1	48/$30	20
0 1 0 0	64/$40	27
0 1 0 1	80/$50	33
0 1 **1** 0	96/$60	40
0 1 1 1	112/$70	47
1 0 0 0	128/$80	53
1 0 0 1	144/$90	60
1 0 1 0	160/SAO	67
1 0 1 1	176/$B0	73
1 1 0 0	192/$C0	80
1 1 0 **1**	2O8/$D0	87
1 1 **1** 0	224/$E0	93
1 1 1 1	240/SFO	100 (peak output)

voice will remain at the level specified here until the release phase is specifically triggered by writing a %0 to the control register for the voice. The sustain level can be considered a percentage of the peak volume level of the output, as shown in the following table. (The overall peak output level is controlled by bits 0-3 of the register at 54296/$D418.)

If 0 is specified for the sustain level, the voice will die away to silence at the end of the decay period. You can change the value in these bits to reduce the sustain level (and hence the output volume) while a sound is being played. However, if you try to increase the sustain level above its current value while a voice is in the sustain phase, the voice will be turned off.

Voice 2 Control Registers

The following seven registers (54279-54285/$D407-$D40D) provide the same control functions for voice 2 that the registers at 54272-54278/$D400-$D406 provide for voice 1. Refer to the entries for the voice 1 registers for details of how these registers are used.

54279 $D407 FRELO2
54280 $D408 FREHI2
Frequency control registers

54281 $D409 PWLO2
54282 $D40A PWHI2
Pulsewidth control registers

54283 $D40B VCREG2
Waveform control register

54284 $D40C ATDCY2
Attack and decay control register

54285 $D40D SUREL2
Sustain and release control register

Voice 3 Control Registers

The following seven registers (54286-54292/$D40E-$D414) provide the same control functions for voice 3 that the registers at 54272-54278/$D400-$D406 provide for voice 1. Refer to the entries for the voice 1 registers for details of how these registers are used.

54286	SD40E	FRELO3
54287	$D40F	FREHI3

Frequency control register

54288	$D410	PWLO3
54289	$D411	PWHI3

Pulsewidth control register

54290	$D412	VCREG3

Waveform control register

54291	$D413	ATDCY3

Attack and decay control register

54292	$D414	SUREL3

Sustain and release control register

54293	$D415	FCLO
54294	SD416	FCHI

Filter cutoff frequency registers

The value in these registers specifies the cutoff frequency for the filter. See the entry for bits 4-6 of the register at 54296/$D418 for more information on the effect of filtering. Only 11 of the 16 bits in this register pair are used. The lower 3 bits of the value come from bits 0-2 of the register at 54293/SD415 and the upper 8 bits come from the register at 54294/$D416. Bits 3-7 of 54293/$D415 are not used, and writing to those bits has no effect. Remember that these are write-only registers; they will always return 0/$00 when read, regardless of the values you have stored in them.

The available 11 bits allow you to specify values in the range 0-2047/$0000-$07FF. However, there is a great deal of confusion about the exact relationship between the value in these registers and the corresponding cutoff frequency. Supposedly, the value here specifies the cutoff frequency in linear steps between a minimum of about 30 Hz and a maximum determined by two external capacitors connected to the SID chip. Commodore's formal specifications for the SID chip state that the equation for maximum cutoff frequency is:

frequency = 2.6 X 10^{-5} / capacitance

Other official literature, including the 128 *Programmer's Reference Guide*, states that the maximum cutoff frequency is about 12,000 Hz, a calculation based on filtering capacitors of 2200 picofarads (2200 X 10^{-12}). There are a number of problems

here. First of all, the capacitors used in the 128 (and, by the way, also in later versions of the Commodore 64) are instead actually 470 picofarads, not 2200, If the stated equations were correct, this would give a maximum cutoff value of over 55,000 Hz, implying that the majority of register values would produce cutoff frequencies beyond the audible range. Simple experimentation shows that this is not the case. This isn't surprising, since the SID's designer, Bob Yannes, stated in an interview in the March 1985 issue of *IEEE Spectrum* magazine that filtering doesn't work according to the specified equation anyway.

So how do you go about selecting a cutoff frequency? Our experience suggests that, while filtering does work after a fashion for any given 128, a value that produces a particular cutoff frequency on one computer may produce a slightly different cutoff frequency on another system. Although it's not a particularly scientific approach, the best way to discover the proper register value for a given cutoff frequency is simply to try different values until the desired effect is achieved.

54295 $D417 RES/FILT

Filter selection and resonance control register

The bits of this chip select whether the various audio sources in the SID will be passed directly to the chip output or routed through the filter stage. This register also controls a special filtering effect known as resonance. Remember that this register is write-only; all bits will return %0 when read, regardless of the values you write to them.

Bit 0: This bit controls whether or not the output for voice 1 passes through the SID's filter stage. When the bit is %0, the filter is bypassed and voice 1 output is routed directly to the combined SID output. When the bit is %1, voice 1 output is routed through the filter before being passed to the chip output. The filter will modify the voice 1 output according to the filter parameters specified in the registers at 54293-54294/ $D415-$D416 and 54296/$D418.

Bit 1: This bit provides the same filter control function for voice 2 that bit 0 provides for voice 1.

Bit 2: This bit provides the same filter control function for voice 3 that bit 0 provides for voice 1.

Bit 3: This bit controls the handling of any external audio input to the SID. As with bits 0-3, setting this bit to %0 connects the external input directly to the combined output, while setting the bit to %1 routes the external input through the filter before output. These two functions—adding directly to the output or adding to the filtered output—are the only processing the SID can perform on the input signal from an external source. The SID's external input line is connected to pin 5 of the composite (40-column) video port. To avoid damage to the SID, you should not use highly amplified signals such as the final output of a home stereo system for the external input source.

Bits 4-7: These bits control an effect of filtering known as resonance. The four bits provide for 16 evenly spaced steps from no resonance (%0000) to full resonance (%1111). Resonance accentuates frequencies near the cutoff frequency for the filter. The higher the resonance, the more pronounced the effect of the selected filter.

54296 $D418 SIGVOL

Volume and filter mode control register

This register controls the overall volume of the SID output, as well as the type of frequency attenuation provided by the filter. Remember that this register is write-only; reading any of the following bits will return %0, regardless of the values you write to the bits.

Bits 0-3: These bits specify the peak volume for the combined output of all three voices plus any external input. The four bits allow for 16 evenly spaced steps between no output (%0000) and maximum output (%1111). Expressed as a percentage of maximum possible output volume, the effects of the settings are roughly as follows:

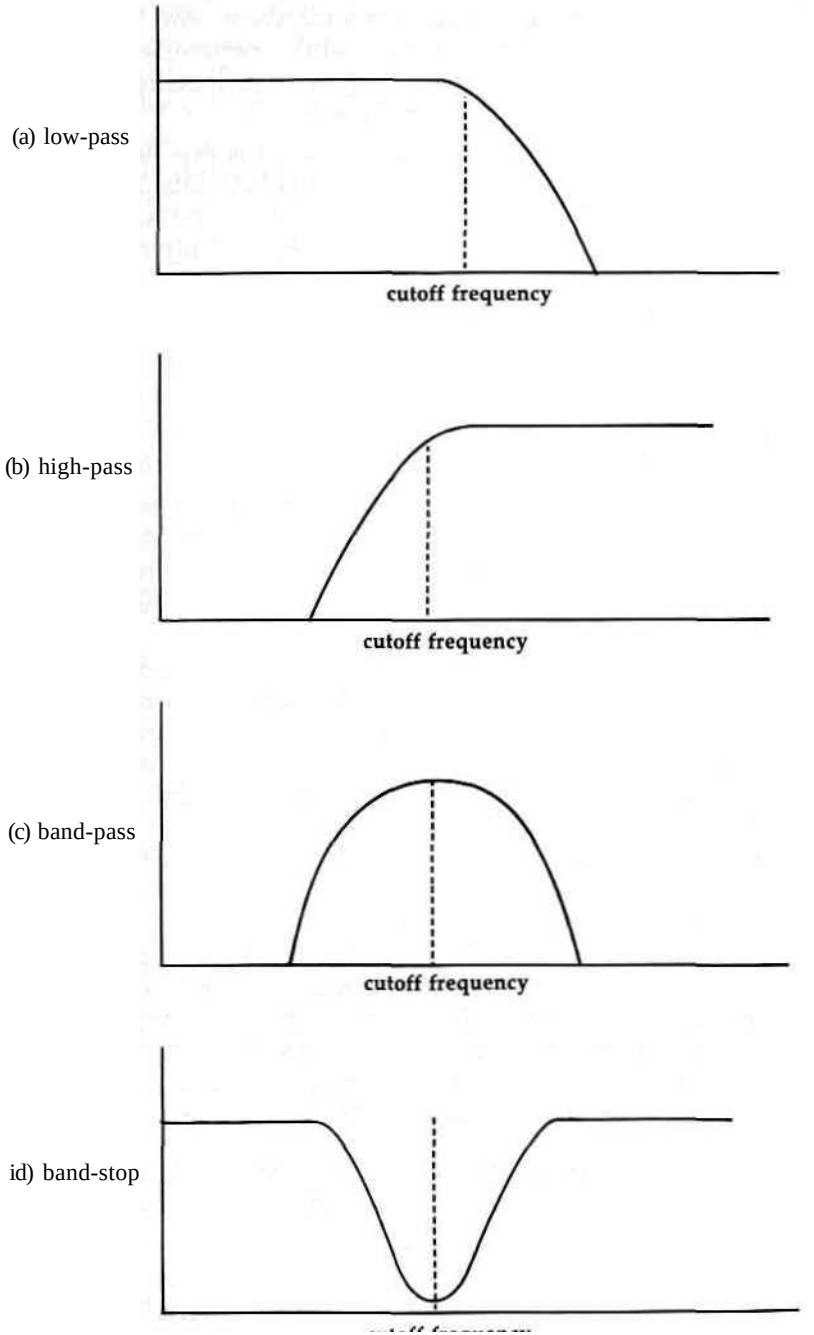
Bits 3 2 10	Value	Percentage of maximum output
0 0 0 0	0/$0	0 (no output)
0 0 0 1	1/$1	7
0 0 1 0	2/$2	13
0 0 1 1	3/$3	20
0 1 0 0	4/$4	27
0 1 0 1	5/$5	33
0 1 1 0	6/$6	40
0 1 1 1	7/$7	47
1 0 0 0	8/$8	53
1 0 0 1	9/$9	60
1 0 1 0	10/$A	67
1 0 1 1	11/$B	73
1 1 0 0	12/$C	80
1 1 0 1	13/$D	87
1 1 1 0	14/$E	93
1 1 1 1	15/$F	100 (maximum output)

These bits must be set to some value greater than zero for the SID to produce any audible output. There's only one volume control for the chip, but the relative output volume level of each voice can be controlled by adjusting the sustain level of the voice's envelope.

Bits 4-6: These bits control the operation of the SID's filter stage. The SID has only one filter for all three voices (plus the external input), so the filtering selections affect any source passed through the filter. Any of the voices can also bypass the filter and connect directly to the output. Bits 0-3 of the register at 54295/$D417 control which voices are routed through the filter. Bits 4-7 of that register control resonance, which can be used to emphasize the effects of filtering.

The SID provides three basic types of filtering, as illustrated in Figure 8-14(a-c). The first type, called a low-pass filter, allows frequencies below a specified cutoff frequency to pass virtually unchanged, but sharply reduces (attenuates) the volume of frequencies higher than the cutoff. The high-pass filter provides the opposite effect, allowing frequencies above the specified cutoff to pass while attenuating frequencies lower than the cutoff. The third selection, the band-pass filter, allows frequencies near the specified cutoff to pass while blocking frequencies that are much above or below the cutoff. The filtering types can also be combined. For example, selecting low-pass and high-pass filters simultaneously provides a fourth

Figure 8-14. Filter Types



type of filter known as the band-stop, or notch, filter (see Figure 8-14d). In this case, frequencies well above and below the cutoff are passed with little change, while frequencies near the cutoff are attenuated. The cutoff frequency is specified in the registers at 54293-54294/$D415-$D416.

Bit 4 controls the low-pass filter, which is enabled when the bit is %1 and disabled when the bit is %0. Bits 5 and 6 provide the control for the high-pass and band-pass filters, respectively, in the same manner. Thus, standard filter selections are as follows:

Bits 6 54	Value	Filter type
0 0 1	16/$10	Low-pass
0 10	32/$20	Band-pass
10 0	64/S40	High-pass
10 1	80/$50	Band-stop

Bit 7: This bit controls whether or not voice 3 can be connected directly to the combined output of the SID chip. While the bit is %0, the voice 3 output will be added to the combined output unless it is routed through the filter. Setting this bit to %1 will prevent the direct connection of voice 3 to the combined output. However, even when voice 3 is blocked from direct connection to the combined output, it's still possible to route voice 3 output through the filter and hence to the combined output. To prevent this and completely disconnect voice 3, make sure that bit 2 of the register at 54295/$D417 is set to %0.

Voice 3 is often used in conjunction with one of the other voices to generate special audio effects, since the oscillator and envelope generator output values for this voice can be read from registers 54299/$D41B and 54300/$D41C, respectively. While voice 3 is used to generate special envelopes or as a random-number generator, it's desirable to insure that the specified frequency and envelope for voice 3 don't cause any disruption of the other voices. This bit provides that feature. Voice 3 is the only voice which can be disconnected.

54297	SD419	POTX
54298	8D41A	POTY

Potentiometer (paddle) reading registers

The SID chip has two special input lines, designated POTX and POTY, which are connected to a pair of internal devices

called analog-to-digital (A/D) converters. The A/D converters generate a one-byte value based on the input voltage. These read-only registers return the values generated by the converters; writing to these locations has no effect. As connected in the 128, the converters measure the voltage across capacitors at the input pins. When a variable resistor (also called a potentiometer) is connected between a constant voltage source and the capacitor, the converters can be used to read the input resistance. The output values will be in the range 0-255, according to the resistor value. Minimum resistance (or a closed circuit) produces a reading of 0/$00 and maximum resistance (or an open circuit, as when nothing is connected to the lines) results in a reading of 255/SFF. The official SID chip specifications state that the relationship between the installed capacitor values and the resistance for maximum output value is:

resistor value = 4.7 X 10^{-4} / capacitor value

The capacitors in the 128 are 1800 picofarads (1800 X 10^{-12}), so any resistance greater than about 4.7 X 10^{-4} / 1800 X 10^{-12} = 261 X 10^{3} ohms (261 Kfl) will result in a register value of 255/$FF. (Actually, our experience indicates that a slightly higher value, 270-280 KQ or so, may be required.)

The SID input lines can be connected to either of the two control ports on the right side of the 128. Bits 6-7 of the CIA #1 port A register at 56320/$DC00 determine which control port is currently connected (for details, see the section on the CIA chip later in this chapter). The default setting connects control port 1, the front one. For either port, the SID register lines are connected to port pins 5 and 9. Location 54297/$D419 will read the level at pin 9 and location 54298/$D41A will read the level at pin 5. From BASIC, the POT function can be used to read these registers. POT(l) reads the register at 54297/$D419 connected to pin 9 of port 1 and POT(2) reads the register at 54298/DlA connected to pin 5 of port 1. POT(3) and POT(4) read the same registers when connected, respectively, to pins 9 and 5 of port 2.

The device most commonly connected to these inputs is the game paddle controller. A paddle is an extremely simple device, consisting of a variable resistor connected between a + 5 volt source (pin 7 of the control port) and the SID A/D converter input line. Turning the paddle knob changes the resistance, and hence the register value. The paddle tends to be more efficient than the joystick for games that require only

horizontal or only vertical movement, such as Pong-type games for which the paddle was originally developed. Since the steps between output levels are so small (only about one ohm), consecutive readings of the same paddle position can vary by one or two register values. This "jitter" can be annoying. The recommended solution is to read the paddle several times, then calculate the average of the readings and use that value.

One thing you should be aware of is that Commodore has changed the values of the capacitors used with the converters since the Commodore 64 was first introduced. The original 64 used 1000 picofarad capacitors, so a resistance of about 470 KX2 was required for a maximum register value. Thus, most paddles currently available for Commodore computers use potentiometers with a top resistance of 470 Kf2 or 500 *KQ*. Such paddles can be used with the 128, but they will swing the full range of register values (0-255) in about the first half of the paddle's full turn. Thus, you probably won't be able to select fine increments of intermediate values. Furthermore, paddles for Atari computers, which are much more widely available than Commodore paddles, use 1 *M£l* (1 million ohm) potentiometers. Again, these can be used with the 128, but in this case you'll see the full register value swing (0-255) in about the first quarter of the paddle's ful] turn. Thus, Atari paddles will give you only very coarse control of the resulting register values.

Most paddle controllers also have fire buttons like joysticks. However, since paddles almost always come in pairs, and since there is only one fire button line per control port, the paddle fire buttons are connected to the lines normally used for joystick direction. Standard Commodore paddles use the lines connected to bits 2 and 3 of each CIA #1 data port (locations 56320-56321/$DCOO-$DC01), the lines for joystick left and right.

These A/D converters can also be used for other interfacing projects. Any device which provides a variable resistance can be connected to the appropriate control port lines and read via these registers. For example, most graphics tablets such as the popular *KoalaPad* effectively function as paddles, with one resistance for the horizontal coordinate of the stylus position and another for the vertical coordinate. You could also rig an interface for an Apple/IBM-style joystick, which

consists of two variable resistors—one on the horizontal axis and one on the vertical axis.

54299 **$D41B** OSC3

Voice 3 oscillator output register

This register reflects the upper eight bits of the internal-oscillator register for voice 3. This is a read-only register; storing values here has no effect. The output signal for each voice is generated by converting the digital bit pattern from the voice's oscillator register into an analog voltage level, so the output signal for the voice is directly proportional to the value in its oscillator. However, voice 3 is the only voice for which the oscillator contents can be read.

For triangle and sawtooth waveforms, the oscillator acts as a repeating counter. For the triangle waveform, the oscillator count starts at zero and increments upward to its maximum count, then decrements downward to zero again. Once the count reaches zero, it immediately begins incrementing again, and the process repeats over and over. The counting rate depends on the value in the frequency control registers for the voice. For all but the lowest frequencies, the count sweeps up and down so quickly that you won't read every intermediate value between 0/$00 and 255/$FF in this register. For the sawtooth waveform, the count also starts at zero and increments upward to maximum count, but in this case the count then returns immediately to zero and begins incrementing again.

Unlike the triangle and sawtooth waveforms, the pulse waveform doesn't sweep smoothly from one output level to the next. Instead, the it jumps back and forth between two discrete levels. When the pulse waveform is selected for voice 3, this register will contain one of two values: 0/$00 when the waveform is at minimum (zero) amplitude and 255/$FF when the waveform is at maximum amplitude. The portion of the cycle spent in the zero-amplitude state is determined by the value in the pulsewidth registers for the voice, and the rate at which the off/on switching repeats is determined by the value in the frequency control registers for the voice.

The noise waveform is different from the others in that it exhibits no regularly repeating pattern. Rather, the oscillator will contain a series of random values. The rate at which the value in the oscillator changes—and hence the rate at which

the value in this register changes—is determined by the value in the frequency control registers for the voice. This feature can be used to provide a random-number generator. If your program needs random numbers in the range 0-255/$00-$FF, simply set the voice 3 frequency registers (54286-54287/ $D40E-$D40F) for a high frequency; then set bits 0 and 7 of the voice 3 control register (54290/$D412) to %1 to start the noise waveform. After that, just read this register whenever you need a random value.

In addition to its use as a random-number generator, this register can be used for a number of special audio effects. The changing output of the oscillator can be used to modify the frequency or pulsewidth of a voice or the filter parameters in realtime. The following is a simple example:

```
1 BOO   LDA #$F0    ;set sustain to maximum level
1B02    STA  $D406
1B0S    LDA #$8F    ;set maximum volume; disconnect voice 3
1B07    STA  $D418
1B0A    LDA #$01    ;set frequency (high byte) for voice 3
1B0C    STA  $D40F
1B0F    LDA #$21    ;start sawtooth waveform on voice 3
1B11    STA  $D412
1B14    LDA #$11    ;start triangle waveform on voice 1
1B16    STA  $D404
1B19    LDA $D41B   ;change voice 1 frequency according to
1B1C    STA  $D401  ;voice 3 oscillator output
1B1F  JMP $1B19
```

Whenever voice 3 is used for special effects such as random-number generation, it should be disconnected from the combined SID output so that it doesn't distort any sounds produced by the other voices. To disconnect voice 3, set bit 7 to %1 in the register at 54296/$D418. Also, make sure that bit 2 of the register at 54295/$D417 is %0. This will insure that voice 3 output is not routed through the filter.

54300 $D41C ENV3

Envelope generator 3 output register

The value in this register reflects the contents of the internal envelope generator for voice 3, This is a read-only register; storing values here has no effect. Each voice has its own envelope generator which controls the peak amplitude of the output for that voice. However, voice 3 is the only one for which the envelope-generator register contents can be read.

The envelope generator regulates the amplitude (volume) of the output for the voice. When voice 3 is silent or turned off, this register will contain 0/$00, indicating no output. When the gate bit for the voice (bit 0 in the register at 54290/$D412) is set to %1, the value here will begin incrementing to 255/$FF, indicating peak output amplitude. This peak amplitude for the voice will be relative to the overall peak volume level specified in bits 0-3 of the register at 54296/$D418.

The rate at which the register increments depends on the attack rate specified in bits 4-7 of the register at 54291/$D413. Once the register reaches 255/$FF, it immediately begins decrementing to the sustain level specified in bits 4-7 of the register at 54292/$D414 (unless the specified sustain level is 15/$F—in that case, the register value remains at 255/$FF following the attack phase).

The rate at which the amplitude drops to the sustain level depends on the decay rate specified in bits 0-3 of the register 54291/$D413. The register value while the voice is in the sustain phase will be equal to the specified sustain-level value repeated in both nybbles. For example, if the sustain level is %1001 = $9, then the value in this register while the voice is in the sustain phase of the envelope will be $99/153. Once the gate bit for the voice is set to %0, the value in this register will decrease from the sustain level to 0/$00. The rate at which the register value decrements is determined by the release rate specified in bits 0-3 of the register at 54292/$D414.

The value in this register can be used to modify other SID parameters, such as the contents of one of the write-only registers, in realtime. For example, you could try continuously storing the value from this register in the filter cutoff-frequency register at 54294/$D416. This would cause the cutoff frequency to rise and fall in conjunction with the voice 3 envelope. If you use the voice 3 envelope generator for special effects, you may want to disconnect voice 3 from the combined SID output so that it doesn't distort any sounds produced by the other voices. To disconnect voice 3, set bit 7 to %1 in the register at 54296/$D418. Also, make sure that bit 2 of the register at 54295/$D417 is %0. This will insure that voice 3 output is not routed through the filter.

54301-54303 $D41D-$D41F Unused

These unused register locations always return the value 0/$00 when read. Writing to these locations has no effect.

54304-54527 $D420-$D4FF

SID register images

Due to incomplete address decoding, images of the SID chip registers appear repeatedly every 32 locations throughout the remainder of this page of memory. That is, storing a value in any location with an address which is an exact multiple of 32 greater than one of the base address locations listed above has the same effect as storing a value in the corresponding base register. For example, storing a value in 54328/$D318 or 54520/$DF18 has the same effect as storing that same value in 54296/$D018. However, it's better programming practice to use the officially designated register addresses.

MMU (Memory Management Unit) Chip Registers

54528~54539/$D500-$D50B and 65280-65284/$FF00-$FF04

The MMU memory management chip, officially designated the 8722, is the cornerstone of the 128 system. In fact, the MMU is what makes most of the 128's special features possible. The MMU was designed by Commodore's engineers specifically to support the 128's multiple operating modes and elaborate memory banking scheme. It is the MMU which determines which microprocessor, 8502 or Z80, has control of the computer. When the 8502 is in control, the MMU determines whether the computer operates in 128 mode or 64 mode. As described in Chapter 1, the 128 hardware includes many times more elements than can simultaneously fit in the 64K address space of the 8502 or Z80 microprocessors. The MMU chip also determines which memory resources are visible to the processsor at any given time.

Most BASIC programming can be done without understanding the inner workings of the MMU, simply by using the available standard bank configurations. However, a thorough knowledge of the MMU is essential for taking full advantage of all the 128's features.

The 17 registers of the MMU are unusual in that they are divided into two separate groups at different memory locations. The first 12 appear in the I/O block at 54528-54539/ $D500-$D50B, while the other 5 appear at 65280-65284/ $FF00-$FF04. This second set of MMU registers is special in that it appears in all memory configurations. There is no way to make the processor see anything other than the MMU registers at those locations while the 128 is in 128 mode. Table 8-4 lists the MMU registers. A detailed description of each register follows.

Table 8-4. **MMU** Chip Registers

Address	**Function**
54528/$D500	Configuration register
54529/$D501	Preconfiguration register A
54530/$D502	Preconfiguration register B
54531/$D503	Preconfiguration register C
54532/$D504	Preconfiguration register D
54533/SD505	Mode configuration register
54534/$D506	RAM configuration register
54535/$D507	Page 0 page pointer
54536/$D508	Page 0 block pointer
54537/$D509	Page 1 page pointer
54538/$D50A	Page 1 block pointer
54539/$D50B	MMU version register
65280/$FF00	Configuration register
65281/$FF01	Load configuration register A
65282/$FF02	Load configuration register B
65283/$FF03	Load configuration register C
65284/$FF04	Load configuration register D

54528 SD500 MMUCR1

Configuration register

The value in the configuration register determines which of the available memory resources will be visible to the microprocessor at any given time. However, this particular register is only rarely used—not at all in 128 ROM except during MMU register initialization—because it has an identical twin at address 65280/SFF00 which is more convenient. There is really only one configuration register; it just appears at two different addresses. The register is accessible here only when the I/O block is visible, whereas it is always accessible at 65280/$FF00, regardless of the memory configuration. As a

result, memory configuration is usually done with the higher configuration register location. Refer to the entry (below) for the other configuration register for details of the function of the register bits.

This register is initialized during the Kernal RESET routine [$E000] to 0/$00, the setting for the bank 15 configuration. However, that step of the routine is redundant because the same value has previously been stored in 65280/$FF00. Reading this register returns the current configuration setting, regardless of whether the setting has been established by writing to this register or to 65280/$FF00.

54529	8D501	PCRA
54530	8D502	PCRB
54531	8D503	PCRC
54532	SD504	PCRD

Preconfiguration registers

These registers provide an indirect method of setting up a memory configuration. Whenever a value—any value—is stored in one of the four]oad configuration registers at 65281-65284/$FF01-$FF04, the value in the corresponding preconfiguration register is transferred to the configuration register. Thus, the preconfiguration registers allow you to set up as many as four different memory configurations, each of which can then be established with a single store operation. The bit functions for preconfiguration register locations are the same as for the configuration register. For details, see the entry below for 65280/$FF00.

The Kernal RESET routine [SEOOO] initializes all the preconfiguration registers to 0/$00, the setting for the bank 15 configuration. The preconfiguration/load configuration register memory management technique is not used by the operating system. However, BASIC ROM routines do make use of this method. The BASIC cold start [$4023] and warm start [$4009] routines both initialize the preconfiguration registers as follows:

Register	Setting	Bank configuration
54529/$D501	63/$3F	0
54530/$D502	127/$7F	1
54531/$D503	l/$01	14
54532/$D504	65/$41	A custom setting with the same visible ROM as bank 14, but with RAM from block 1 instead of block 0

In a number of instances in BASIC ROM you'll find instructions like STA $FF03, where BASIC is using this shortcut method of bank selection.

Because BASIC depends on standard values in the preconfiguration registers, it's generally best to avoid changing the register settings in machine language programs which must work in conjunction with BASIC. To reset the preconfiguration registers to their default values without performing the entire BASIC warm start or cold start sequence, call the BASIC ROM subroutine at 16762/$417A. Of course, if BASIC is not being used, you're free to use the preconfiguration registers to set up any memory configuration you desire.

54533 $D505 MMUMCR

Mode configuration register

The primary function of this register is to select the current operating mode from the three possibilities—128, 64, or CP/M.

Bit 0: This bit determines which microprocessor is in control of the system. Writing a %0 here puts the Z80 in command, while writing a %1 switches to the 8502. Since this bit is reset to %0 when the system is reset or powered on, the Z80 always has control of the system before the 8502. The reset sequence in Z80 ROM is nearly identical to that in 128 mode ROM, including checking for the presence of Commodore 64 cartridges and testing whether the Commodore key is held down for Commodore 64 mode. If the Z80 reset routine does not find a CP/M boot disk in the drive (or a Commodore 64 cartridge or the Commodore key held down), it jumps to a routine it has copied into 65504/$FFE0 in block 0 RAM. That routine ends by setting this bit to %1 to return control to the 8502 for 128 mode.

Switching processors is not for the faint of heart. When you activate the Z80, it will begin executing instructions at whatever address is currently in its program counter registers. The address in those internal processor registers can't be changed from 128 mode, so you're stuck with having the Z80 take up wherever it left off when the system was switched to 128 mode. This address is usually 65518/$FFEE, the location following the one where 128 mode was activated at the end of the Z80's reset routine. In block 0 RAM, that location is initialized with a Z80 instruction (RST1) to perform a warm start of CP/M mode. If you don't have a valid Z80 machine language

instruction there when you activate the Z80—for example, if the system is in a memory configuration such as bank 15 where 128 Kernal ROM is seen at that address—you'll probably experience an immediate system lockup.

If you wish to use any routines in the CP/M BIOS (Basic Input/Output System) ROM, you must make sure that bit 6 of this register also is set to make 128 mode ROM visible. The BIOS code is stored in the same ROM chip used to hold the 128 mode screen editor and Kernal routines, so you'll have problems if that ROM is not visible. Activating the Z80 makes the BIOS ROM visible at addresses 0-4095/$0000-$0FFF (even though the ROM is physically located at 53248-57343/ $DO00-$DFFF). Thus, the Z80 never disturbs the lowest 4K of block 0 RAM, so the 8502's zero page and stack (page 1) are preserved while the Z80 is in control.

If you want to switch to full CP/M mode, there's more involved than simply turning on the Z80. Most of the CP/M operating system must be loaded from disk, so you should instead use the BASIC BOOT statement or the Kernal BOOT_CALL routine [$FF53] with a CP/M boot disk in the drive.

Bits 1-2: Unused. These bits always return %1 when read, and writing to these bits has no effect.

Bit 3: This bit is connected to the MMU pin labeled FSDIR. This line is bidirectional, meaning that it can be both an input and an output. The 128 uses the line only as an output, to control the direction of data flow on the fast serial bus. The lines of the slow serial bus are each controlled by a pair of CIA chip lines, one for input and one for output, but the fast serial bus uses the same CIA chip lines for both input and output. The 128's designers added additional circuitry which insures that the fast serial bus will ignore incoming data during fast serial output. The FSDIR (fast serial direction) line controls that circuitry. Writing a %0 to the bit pulls the line to a low (0 volts) state, which sets the fast serial bus for input. Writing a %1 to the bit allows the line to go to a high (+ 5 volts) state, which sets the fast serial bus for output. The IOINIT routine [$E109] leaves this bit set to %0 so that the system can detect fast serial input. The setting of this bit is controlled in Kernal ROM by the routines SPIN [$E5C3] and SPOUT [$E5D6].

Bits 4-5: These bits are connected, respectively, to the MMU pins labeled GAME and EXROM, which in the 128 are connected to the memory expansion port lines with the same names. These MMU lines are bidirectional, meaning that they can be both inputs and outputs. The 128 uses the lines only as inputs, to read the state of the memory expansion port lines (pins 8 and 9 of the port). Writing to one of the bits sets the corresponding line's output level. Writing a %0 to the bit pulls the line to a low (0 volts) state, while writing a %1 to the bit allows the line to go to a high (+ 5 volts) state. When used for input, an external device connected to the line can pull the line low if its output level is set high, but cannot bring the line high if its output level is set low, so writing a %0 to one of these bits effectively blocks the use of the line as an input.

The Kernal RESET routine [$E000] initializes both of these bits to %1 so that they can be used to read the state of the port lines. In 128 ROM, these bits are read during the reset sequence by the routine which checks for the presence of a Commodore 64 ROM cartridge [$E242], Almost all Commodore 64 cartridges ground one or both of these port lines, so the 128 will assume that a 64 cartridge is present if either of these bits is found to be %0, and will respond by switching to Commodore 64 mode.

While the system is in 128 mode, you can use these bits to control the corresponding expansion port pins as either inputs or outputs. Keep in mind, however, that the pins must be high during reset or the system will enter Commodore 64 mode.

Bit 6: This bit controls which set of ROMs will be visible to the system. Writing a %0 here selects the 128 mode ROM set, while writing a %1 selects the Commodore 64 mode ROM. The 128 mode RESET routine [$EOOOj, of course, initializes this bit to %0. However, simply writing a %1 here won't cause a clean transfer to 64 mode; it's necessary to perform the 64 mode reset sequence after the 64 mode ROM is selected. See the C64_MODE routine [$E24B] for details. This bit has one other effect—selecting 64 mode also makes all the MMU chip registers invisible, so once you make the jump to 64 mode there is no way back to 128 mode short of resetting the computer or turning it off and back on.

Bit 7: This bit is connected to the MMU pin labeled 40/80. This line is bidirectional meaning that it can be both an input and an output. The 128 uses the line only as an input, to read the 40/80 DISPLAY switch on the 128's keyboard. Writing to the bit sets the state of the line output. Writing a %0 to the bit pulls the line to a low (0 volts) state, while writing a %1 to the bit allows the line to go to a high (+ 5 volts) state. The switch connected to the line can pull the line low if its output level is set high, but cannot bring the line high if its output level is set low. Thus, writing a %0 to this bit effectively blocks the reading of the 40/80 DISPLAY switch. The CINT screen editor initialization routine [SC07B], part of both the reset and RUN/STOP-RESTORE sequences, sets this bit to %1 to insure that the switch can be read. The bit returns a %0 when read while the switch is down (80-column position), or a %1 while the switch is up (40-column position). In 128 ROM, this bit is read only during the CINT routine, where its setting is used to determine which display to make active.

54534 8D506 MMURCR

RAM configuration register

An important aspect of the MMU's memory configuration capabilities is its ability to create common areas of RAM, areas where the same RAM is seen regardless of the configuration register selection. Kernal routines copied into the default common area allow the processor to jump from bank to bank, or to manipulate data in other banks. This register controls the common RAM feature, and also specifies which RAM block the VIC video banks are seen in.

Bits 0-3: These bits control the common RAM feature. Bits 0-1 control the size of the common areas, and bits 2-3 control whether the common areas will exist at the top, bottom, or both top and bottom of the RAM blocks. (No RAM will be common if bits 2-3 are set to %00.) When common RAM is specified, the RAM seen as common is always that from block 0. As long as the common RAM feature is enabled, there is no way for the processor to access the RAM in block 1 which is covered by the common area. When a common area at the top of memory is selected, it will be visible only when bits 4-5 of the configuration register are set to make RAM visible at the top of memory.

There are a few exceptions to the common RAM rules. First, locations 0-l/$00-$01, the 8502 on-chip I/O port, and locations 65280-65284/$FF00-$FFO4, the upper MMU registers, always appear in any configuration, regardless of the setting of these bits. That is, there is no way to make the processor see anything other than the hardware registers in these locations (at least not while the computer is operating in 128 mode). Second, the area where the two lowest pages of memory are seen is also affected by the registers at 54535-54538/ $DF07-$DF0A. Even if you change the page pointers at 54535/$D507 or 54537/SD509 to physically move zero page or page 1 somewhere in memory outside the common area, the contents of those pages will still appear to be common as long as a common area at the bottom of memory is selected. If you disable the common area at the bottom of memory, all references to zero page and page 1 will continue to affect only block 0 RAM unless you specifically change the block pointers for those pages in the registers at 54536/SDF08 and 54538/ $DF0A. Finally, you should be aware that the common area setting does not affect the RAM block from which the VIC chip sees its video bank. The VIC's RAM block is determined by bits 6-7 of this register, and all RAM in the VIC bank is seen in the block specified in those bits without regard for the common area specification. The VIC chip can see the memory that is hidden from the processor.

The four possible selections for the size of the common areas are as follows:

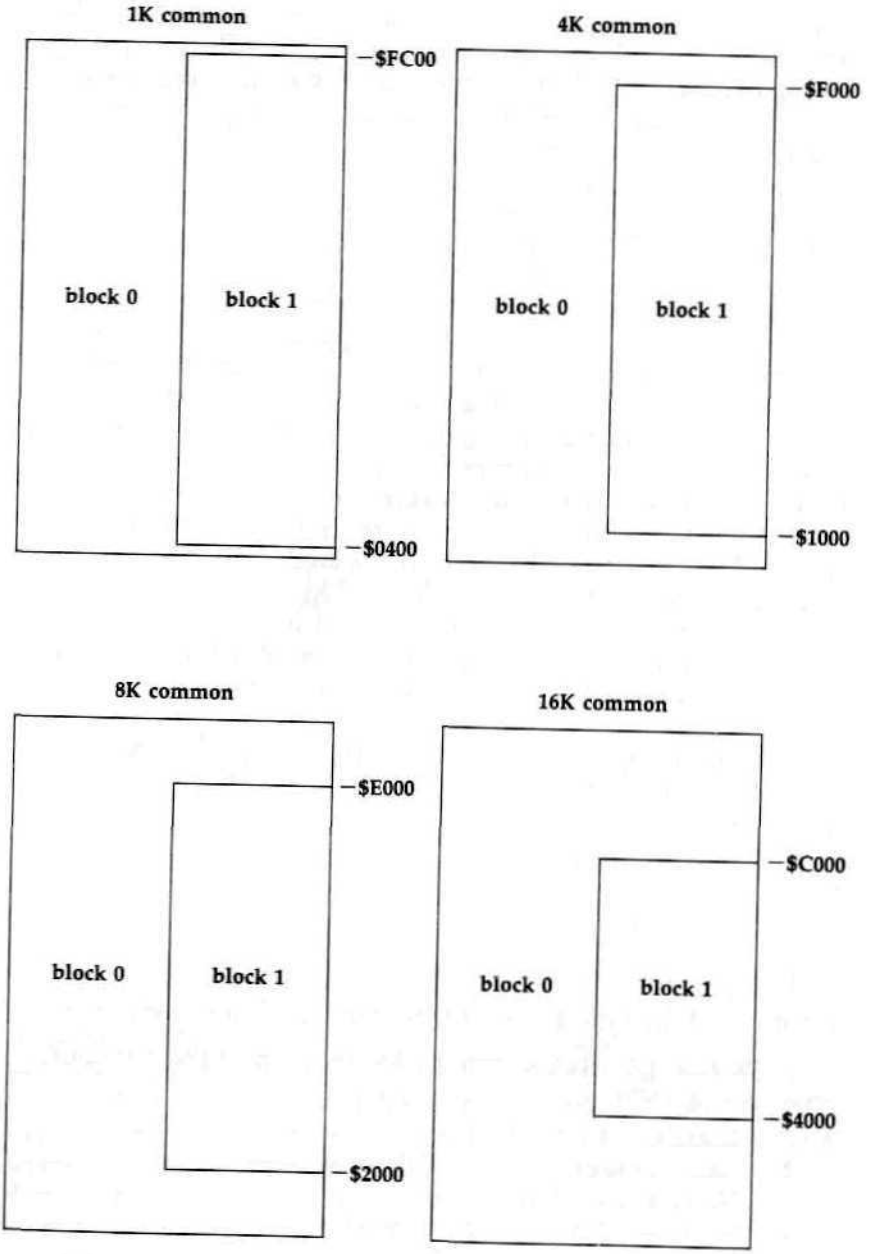
Hits	Size of common areas
1 0	IK
0 0	4K
0 1	8K
1 0	16K
1 1	

Figure 8-15 illustrates the address ranges of these selections.

The four possible selections for location of the common areas are as follows:

Bits	Location of common areas
3 2	
0 0	No common RAM
0 1	Common area at bottom of memory
1 0	Common area at top of memory
1 1	Common areas at both top and bottom of memory

Figure 8-15. Common RAM Areas



The default value in these bits, established during the Kernal RESET routine [SEOOO], is %0100 = $4. This setting selects a IK common area at the bottom of memory, addresses 0-1023/ $0000-$03FF. The proper functioning of both the operating system and BASIC depends on the presence of this common area, so you should change the setting of these bits with great care. BASIC is almost certain to crash if the common area is disabled altogether. The operating system will crash if an interrupt occurs while the system is configured for block 1 RAM with the common area disabled.

You can manipulate these bits to gain temporary access to the IK of RAM from block 1 which is normally hidden by the standard common area, but making any practical use of the hidden area is more than a little complicated. The standard interbank data transfer routines like INDFET and INDSTA can't be used because they depend on the common area. Furthermore, the machine language routine you write to perform the transfer can't use any zero-page locations, nor can it use jumps to subroutines or other instructions which would affect the stack (page 1). Interrupts must also be disabled while the common area is disabled. The following routine swaps the contents of the hidden block 1 area with the contents of the next higher IK area of block 1 (addresses 1024-2047/$0400-$07FF). Note that this routine must be placed in block 1 RAM:

```
1FF60   SEI           ;Disable interrupts
1FF61   LDA  #$7E     ;Make I/O block visible
1FF63   STA  $FF0O
1FF66   LDA  #$00     ;Disable common RAM
1FF68   STA  $D506
1FF6B   LDA  #$01     ;Make pages 0 and 1 visible
1FF6D   STA  $D508    ; block 1 RAM
1FF70   STA  $D50A
1FF73   LDA  #$00
1FF75   STA  $D507
1FF78   LDA  #$01
1FF7A   STA  $D509
1FF7D   LDA  #$00     •Initialize source starting
1FF7F   STA  $FF91    ; address locations
1FF82   STA  $FF98
1FF85   LDA  #$04     initialize target starting
1FF87   STA  $FF95    ; address locations
1FF8A   STA  $FF9C
1FF8D   LDY  #$02     initialize index for first page
1FF8F   LDA  $0000,Y  ;Swap bytes
```

```
1FF92   TAX
1FF93   LDA  $04O0,Y
1FF96   STA  $000O,Y
1FF99   TXA
1FF9A   STA  $O4O0,Y
1FF9D   INY              ;Repeat for 256 bytes per page
1FF9E   BNE  $FF8F
1FFA0   INC  $FF91       increment address high bytes
1FFA3   INC  $FF95
1FFA6   INC  SFF98
1FFA9   INC  $FF9C
1FFAC   LDA  $FF9C       ;Check whether all four pages have
1FFAF   CMP  #$08        ; been swapped
1FFB1   BCC  $FF8F
1FFB3   LDA  #$00        ;I( so, restore pages 0 and 1
1FFB5   STA  $D508       ; to block 0 RAM
1FFB8   STA  $D50A
1FFBB   STA  $D507
1FFBE   LDA  #$01
1FFCO   STA  $D509
1FFC3   LDA  #$04        ;Reenable common RAM
1FFC5   STA  $D5O6
IFFC8   CLI              ;Reenable interrupts
1FFC9   RTS
```

To execute the routine, use J 1FF60 from the monitor or BANK 1:SYS 65376 from BASIC.

Bits 4-5: These two bits are not used in the current version of MMU. The design specifications for the chip indicate that in future versions these bits may be used in a "superbanking" scheme to select one of four separate 256K blocks of RAM in a 1M (one-megabyte, or 1024K) system. It's a tantalizing prospect, but there's no guarantee that a Commodore 1024 will ever be produced. In the current MMU, the bits will retain whatever value is written to them, but changing the bit settings has no effect on the memory configuration. The bits are initialized during the Kemal RESET routine [$E000] to %00.
Bits 6-7: These bits determine which 64K block of RAM the VIC video bank is located in. (See the section on the VIC chip earlier in this chapter for more information on video banks.) The formally defined selections for the bits are as follows:

Bits		RAM block for VIC video bank
7	6	
0	0	0
0	1	1
1	0	2
1	1	3

Remember, however, the 128 actually has only two 64K blocks of RAM. Thus, the setting of bit 7 is meaningless, although that bit will retain whatever value you write to it. Only the setting of bit 6 matters, since the choice is between blocks 0 and 1. The default setting for these bits is %00, since the normal VIC text and bitmapped screens are located in block 0 RAM.

You should note that the operating system and BASIC will not support a VIC video bank in block 1, That is, all the screen editor ROM routines and BASIC bitmapped graphics routines assume that the VIC screen is located in block 0 RAM, and will continue to write data to block 0 even after you've changed these bits to switch the video bank into block 1 RAM. Thus, you must provide your own text and graphics routines for a display from block 1. For text displays, you can use the standard character sets because the character ROM is still visible in a block 1 video bank.

These bits actually determine the RAM block for all DMA (direct memory access) operations, not just for the VIC chip, but the VIC is the only built-in device in the 128 to use DMA. The only other commonly available DMA device for the 128 is the REC (RAM expansion controller) chip in the 1700 and 1750 Memory Expansion Modules. These bits, rather than bits 6-7 of the configuration register, determine which block of 128 system RAM is affected by REC transfer operations.

54535	8D507	MMUPOL
54536	$D508	MMUPOH

Page 0 pointers

One unique feature provided by the MMU is the ability to relocate page 0 anywhere in memory. Zero page is the most heavily used area of memory in any computer built around a 6502-family microprocessor like the 128's 8502, since the processor has many instructions that are designed to work only with this page. Although neither the 128 operating system or BASIC makes use of this page-relocation capability, it has sev-

eral interesting applications. For example, you could have several programs in memory simultaneously and give each its own personal zero page, without having to worry about memory conflicts.

Location 54535/$DF07, the page pointer, selects the page of memory to which zero page is to be relocated. The Kernal RESET routine [$E0O0] stores the value 0/$00 here to locate zero page initially in the true zero page. To move zero page, store the target page number here. {The page number is equivalent to the high byte of the first address in the page.) Actually, the operation is more a swap than a relocation. When you redirect zero page to another page of memory, all references to addresses in the target page are diverted to the true zero page. For example, if you store the value 19/$13 in this register to move zero page to page 19, then all references to addresses in the range 2-255/$02-$FF will be redirected to 4866-5119/$1302-$13FF, and all references to 4864-5119/ $1300-$13FF will be redirected to 0-255/$00-$FF. Notice that addresses 0-1 from zero page are not affected. These are the processor's on-chip I/O port registers—not RAM. Storing values in these locations always affects the registers rather than RAM.

Location 54536/$D508 selects the block of RAM in which zero page will be seen. Only bit 0 of the block pointer is significant. Setting that bit to %0 selects block 0 RAM, while setting it to %1 selects block 1. The bit is set to %0 by the Kernal RESET routine [$E000]. Bits 1-3 will retain whatever value is written to them, but changing these bits has no effect. The RESET routine sets these bits to %000, Bits 4-7 are also unused, but these bits always return %1 when read. As a result, the value in this register will always be at least 240/$F0. Values written to the page pointer take effect immediately, but values written to the block pointer are actually effective only after the next time a value is stored in the page pointer. Thus, if you are changing both registers it is important to remember to store the value in the block pointer (54536/$DF08) before storing the value in the page pointer (54535/$DF07).

However, the block pointer value is ineffective while the RAM configuration register (54534/$D506) is set to provide a common area at the bottom of memory. With the bottom common area enabled, zero page is always seen in block 0 RAM, although it can still be moved around freely within block 0. Since this is the default configuration for the 128, there is

rarely a need to change the value in 54536/$D508. If you select block 1 RAM while the bottom common area is enabled, a strange effect results. Zero page is relocated to the page specified in the page pointer, but that page is not swapped with zero page. That is, both pages will occupy the same area of memory, and the true zero-page area remains untouched.

When choosing areas of memory for a relocated zero page, you must avoid page 1 (or the area where you relocate page 1), since that page is also vital to proper system operation. You should also avoid page 255/$FF, since attempting to place zero page there will cause conflicts with the MMU registers. Because all references to the target page will affect true zero page, you should be careful to avoid using any locations in the target page while zero page is relocated (unless you really want to change true zero page). After relocating zero page, you will probably want to call the Kernal routines RAMTAS [$FF87] and CINT [$FF81] to initialize important zero-page locations. (If you switch to a zero page without proper screen editor variables, you may find yourself looking at a garbled mess on the screen.)

54537 $D509 MMUP1L
54538 SD50A MMUP1H

Page **1** pointers

Just as the MMU can relocate zero page, it also has the ability to make page 1 appear anywhere in memory. Page 1 is vital in any computer built around a 6502-family microprocessor like the 128's 8502, since that page is the processor stack. The stack is the area of memory where the processor stores return addresses while it is executing subroutines or handling interrupts. The stack is also used extensively for temporary data storage. Although neither the 128 operating system or BASIC makes use of this page-relocation capability, it has several interesting applications. For example, you could have several programs in memory simultaneously and give each its own personal stack, without having to worry about memory conflicts.

Location 54537/$DF09, the page pointer, selects the page of memory to which page 1 is to be relocated. The Kernal RESET routine [$E000] stores the value 0/$01 here to locate page 1 initially in the true page 1. To move page 1, store the target page number here. (The page number is equivalent to the high byte of the first address in the page.) Actually, the operation is

more a swap than a relocation. When you redirect page 1 to another page of memory, all references to addresses in the target page are diverted to the true page 1. For example, if you store the value 21/$ 15 in this register to move page 1 to page 21, then all references to addresses in the range 256-511/$0100-$01FF will be redirected to 5376-5631/$1500-$15FF, and all references to 5376-5631/$1500-$15FF will be redirected to 256-511/$0100-$01FF.

Location 54539/$D50A selects the block of RAM in which page 1 will be seen. Only bit 0 of the block pointer is significant. Setting that bit to %0 selects block 0 RAM, while setting it to %1 selects block 1. The Kernal RESET routine [$E000] sets this bit to %0 for block 0. Bits 1-3 will retain whatever value is written to them, but changing these bits has no effect. The bits are set by the RESET routine to %000. Bits 4-7 are also unused, but these bits always return %1 when read. As a result, the value in this register will always be at least 240/$F0. Values written to the page pointer take effect immediately, but values written to the block pointer are actually effective only after the next time a value is stored in the page pointer. Thus, if you are changing both registers it is important to remember to store the value in the block pointer (54538/$DF0A) before storing the value in the page pointer (54537/$DF09).

However, the block pointer value is ineffective while the RAM configuration register (54534/$D506) is set to provide a common area at the bottom of memory. With the bottom common area enabled, page 1 is always seen in block 0 RAM—although it can still be moved around freely within block 0, Since this is the default configuration for the 128, there is rarely a need to change the value in 54538/$D50A. If you select block 1 RAM while the bottom common area is enabled, a strange effect results. Page 1 is relocated to the page specified in the page pointer, but that page is not swapped with page 1. That is, both pages will occupy the same area of memory, and the true page 1 area remains untouched.

This mobile page 1 allows a tricky technique for filling areas of memory. Instead of storing values in a series of locations, you can redirect page 1 to the desired area of memory and push values onto the relocated stack. The advantage of this is that the PHA instruction takes only half as long to execute as the STA (address),Y instruction. Just be sure to pre-

serve the original stack pointer; otherwise, your program will crash when the memory moving subroutine tries to return to its calling routine. The two routines below are alternate methods of filling the high-resolution screen area. The conventional method (on the left) is shorter, but the relocated stack method (on the right) is about 30 percent faster.

```
B00  LDA  #$00
B02  STA  $FB
B04  LDA  #$20
B06  STA  $FC
B08  LDA  #$FF
B0A  LDX  #$20
B0C  LDY  #$00
B0E  STA  ($FB),Y
B10  INY
B11  BNE  $B0E
B13  INC  $FC
B15  DEX
B16  BNE  $C11
B18  RTS
```

```
C00  TSX
C01  STX  $FB
C03  LDX  #$FF
C05  TXS
C06  LDA  #20
C08  STA  $D509
C0B  LDA  #$FF
C0D  LDX  #$20
C0F  LDY  #$00
C11  PHA
C12  INY
C13  BNE  $C11
C15  INC  $D509
C18  DEX
C19  BNE  $C11
C1B  LDX  $FB
C1D  TXS
C1E  LDA  #$01
C20  STA  $D509
C23  RTS
```

54539 $D50B MMUVER

Version register

This read-only register returns a constant value (much like a ROM location) reflecting the amount of RAM in the system and the version of the MMU. If Commodore ever introduces successors to the 128 built with similar system architecture (a Commodore 256 or 1024, for example), this register will allow software to identify the amount of memory available.

Bits 0-3: These four bits indicate the version of the MMU chip installed in the 128. In current 128s the value here is %0000 = 0, indicating version 0 of the MMU, but this may change if new versions of the chip are introduced.

Bits 4-7: These four bits indicate the number of 64K blocks of RAM present in the system. In the 128, the value here is %0010 = 2, indicating 128K of RAM in two 64K blocks.

54540-54783 $D50C-$D5FF Unused

All the unused register addresses in this range return the value 255/SFF when read. Writing to these locations has no effect.

65280 $FF00 MMUCR

Configuration register

This is one of the most important locations in all of 128 memory, since the value here determines what other memory elements will be visible to the processor. The entire design of the 128 is contingent on the MMU's ability to make various selections of the system's memory resources visible at shared locations within the processor's limited address space. The 16 banks supported by the operating system are merely 16 predefined settings of this register—not 16 physical arrangements of memory. See Table 1-1 in Chapter 1 for details of the standard bank configurations.

Configurations other than the standard banks are certainly possible. Since each of the eight bits of this register is assigned a function in the MMU specifications, there are theoretically 256 possible different memory configurations. Actually, there are only 128 functional combinations because bit 7 of the register in not implemented in the current version of the MMU. However, not all of these possible configurations are equally useful. For example, none of the configurations which involve either internal or external function ROM are useful unless you have a function ROM installed. The only configuration regularly employed by the system that doesn't correspond to a standard bank is one used by BASIC, consisting of BASIC ROM, screen editor ROM, character ROM, and Kernal ROM, plus block 1 RAM (essentially the same as bank 14, but with RAM from block 1 instead of block 0). Machine language programmers may find it useful to set up a configuration which switches out BASIC ROM while retaining the I/O block and screen editor and Kernal ROM, Such a configuration leaves 4IK free for ML programs in the range 7168-49151/$1COO-$BFFF. To set up this arrangement, use LDA #$0F:STA $FFOO.

This location has an identical twin at address 54528/$D500. Actually, there is only one configuration register, but it can be accessed at two different addresses. The higher address is used almost exclusively because it is visible in all memory configurations, whereas the register is visible at 54528/$D500 only when the I/O block is selected (and you must have ac-

cess to the configuration register to make the I/O block visible). Both registers will hold identical values, regardless of which register is written to set the value.

There is an alternative to storing values directly in this register. You can store up to four configuration register settings in the preconfiguration registers at 54529-54532/$D501-$D504, then transfer the values to the configuration register by writing to the corresponding load configuration registers at 65281-65284/$FF01-$FF04. See the preconfiguration and load configuration register entries for details.

Bit 0: This bit determines what is seen at addresses in the range 53248-57343/$D000-$DFFF. When the bit is set to %0, the I/O block (containing hardware chip registers and color RAM) is visible. When the bit is set to %1, the contents of the address area is determined by the setting of bits 4-5.

Bit 1: This bit determines what is seen at addresses in the range 16384-32767/$4000-$7FFF. When the bit is set to %0, the lower portion of BASIC ROM appears there. When the bit is set to %1, the address area will contain RAM from the block specified in bits 6-7.

Bits 2-3: These bits determine what is seen at addresses in the range 32768-49151/$8000-$BFFF. The four possible selections are as follows:

Bits		
3	2	Address range contents
0	0	Upper portion of BASIC ROM ($8000-$AFFF), plus monitor ROM ($BOOO-$BFFF)
0	1	Internal function ROM
1	0	External function ROM
1	1	RAM

Internal function ROM refers to ROM in the free ROM socket on the 128 circuit board. External function ROM refers to ROM in a cartridge plugged into the expansion port. If you select either of these sources when no ROM is actually installed, the area will appear to contain unpredictable changing values. When RAM is selected in this area, the block from which the RAM will be seen is determined by the setting of bits 6-7.

Bits 4-5: These bits determine what is seen at addresses in the range 49l52-65535/$C000-$FFFF, with some exceptions. The MMU configuration and load configuration registers always appear at 65280-65284/$FF00-$FF04, regardless of the

settings of these bits. Also, bit 0 of this register can override the specification for the contents of addresses in the range 53248-57343/$D000-$DFFF. As long as bit 0 is set to %0, the I/O block will be seen in that portion of this area, regardless of the setting of these bits. The four possible selections for this area are as follows:

Bits 5 4	Address area contents
0 0	Screen editor ROM ($COOO-$CFFF), character ROM ($D00O-$DFFF), Kemal ROM ($EOOO-$FFFF)
	Internal function ROM
	External function ROM
	RAM

Internal function ROM refers to ROM in the free ROM socket on the 128 circuit board. External function ROM refers to ROM in a cartridge plugged into the expansion port. If internal or external function ROM is selected when no ROM is actually present, the area will appear to contain unpredictable changing values. When RAM is selected, the area will contain RAM from the block specified in bits 6-7, unless the MMU's RAM configuration register at 54534/$D506 specifies a common area at the top of memory. When a common area is enabled, all RAM in the common area will come from block 0, regardless of the block specified in bits *6-7.*

Bits 6-7: The memory configuration established by the 128 always includes RAM in the lowest 16K area (addresses 2-16383/$0002-$3FFF), and RAM may be selected in any of the other three 16K segments in the processor's 64K address space. These bits determine which 64K RAM block the RAM in the selected configuration will be seen from. The formal specifications for these bits are as follows:

Bits 7 6	RAM block selected
0 0	Block 0
0 1	Block 1
1 0	Block 2
1 1	Block 3

Remember, however, that the 128 actually has only two 64K blocks of RAM (blocks 0 and 1). Thus, the setting of bit 7 is meaningless and has no effect in the current version of the

MMU. Bit 7 will retain whatever value you write to it, but only bit 6 is significant.

The block specification in these bits will be overridden for certain ranges of memory if any common RAM areas are specified by the RAM configuration register at 54534/$D506. When common areas are enabled, any visible RAM in the common range will be seen from block 0, regardless of the setting of these bits.

These bits specify the RAM block for the processor only; the block in which the VIC (40-column) chip's video RAM bank is seen can be selected independently. The VIC block is specified in bits 6-7 of the MMU's RAM configuration register at 54534/$D506. The RAM configuration register bits, rather than the configuration register bits, also determine which block will be affected by other DMA (direct memory access) operations, such as data transfers by the REC (RAM expansion controller) chip in the RAM expansion modules.

65281	SFF01	LCRA
65282	$FF02	LCRB
65283	$FF03	LCRC
65284	$FF04	LCRD

Load configuration registers

Each of these registers has a corresponding preconfiguration register at 54529-54532/$D501-$D504. Storing a value in a load configuration register causes the value in the preconfiguration register to be transferred to the configuration register. The value stored in the load configuration register is irrelevant; it is the store operation, rather than the value stored, which causes the transfer. Reading any of the load configuration register locations returns the value in the corresponding preconfiguration register. Values stored in a load configuration register location have no effect on the value returned when the register is read.

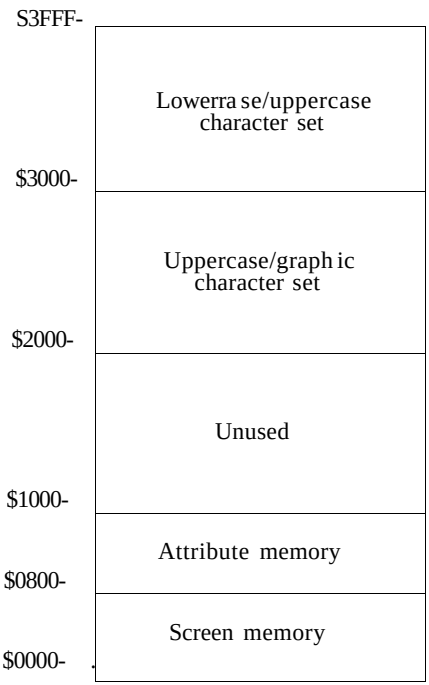
The 128 operating system does not use the preconfiguration or load configuration registers, but BASIC does. See the entry above for the preconfiguration registers for details of the standard configuration settings.

VDC (Video Display Controller) Chip Registers

54784-54785/$D600-$D601

The VDC (video display controller), officially designated the 8563, is a custom chip designed by Commodore's engineers especially to provide the 80-column display for the 128. The VDC provides a digital RGBI signal requiring a special monitor, but other than that the fundamentals of its video display are similar to those described for the VIC chip. The other thing that distinguishes the VDC from the VIC is the VDC's high degree of programmability. Many of the features that are fixed in the silicon of the VIC can be customized on the VDC simply by storing a value in one of its registers. For example, the VDC gives you complete control over the number of rows and columns in the screen display, and even over the number of pixels and scan lines in each character position.

Figure 8-16. VDC Memory Configuration



Although the VDC does have a graphic mode (see the entry for bit 7 of register 25/$19), it is primarily used for text displays. In this mode, it uses the same basic elements as the VIC: screen memory, character memory, and attribute memory (which corresponds to the VIC's color memory). However, all these elements for the VDC appear in a 16K block of memory that is totally separate from the 128's system address space. This separate block is reserved solely for the VDC. Figure 8-16 illustrates the configuration of VDC memory.

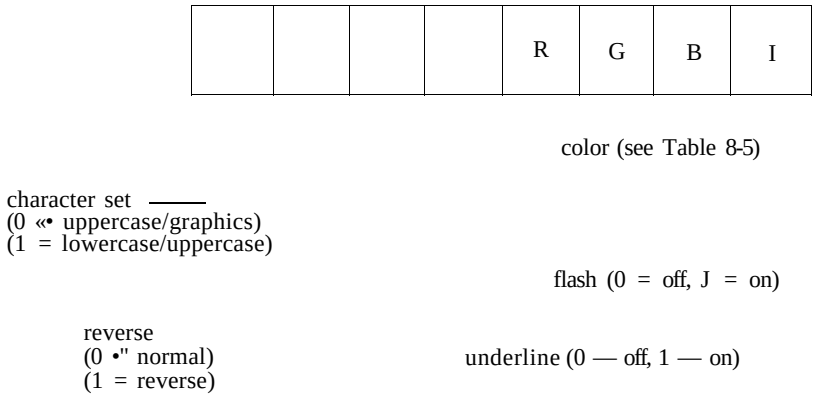
VDC screen memory holds screen code values identical to the screen codes for the VIC display. For each screen position, the screen code serves as an index into character memory to select the pattern to be displayed in the position. All character patterns are in RAM. The VDC has no character ROM of its own, so the contents of the VIC's character ROM is copied into VDC RAM during system initialization. Redefining characters for the VDC is as simple as storing the new pattern in the proper area of character memory. Each character definition is 16 bytes long, but because the default character height is eight scan lines, only the first eight bytes are used to hold character pattern information. The remaining eight bytes are generally padded with zeros. To determine the starting address for the definition for any character, use the appropriate formula from the following:

for uppercase/graphics character set:
address = (screen code * 16) + 8192
for lowercase/uppercase character set:
address = (screen code * 16) + 12288

The remaining memory area is attribute memory, which determines the display characteristics for the character specified in screen memory. It is called attribute memory to distinguish it from simple color memory because each attribute memory location specifies more than just the color. Figure 8-17 shows the use of each bit of attribute memory.

The lower four bits specify the foreground color for the character in the corresponding screen position. Unlike the VIC's color numbers, there is a strictly logical relationship between the value in these bits and the resulting color. The RGBI in Figure 8-17 stands for *red, green, blue,* and *intensity.* For example, a value of %1000 in these bits selects only red, while

Figure 8-17. VDC Character Attributes



%1011 selects red plus blue plus intensity, resulting in light purple. Table 8-5 shows the standard VDC color values. Note that these values are different from the BASIC color numbers.

Table 8-5. Standard VDC Color Values

Value	VDC **color**
0/$00	black
l/$01	dark gray (light black)
2/$02	dark blue
3/$03	light blue
4/$04	dark green
5/$05	light green
6/$06	dark cyan
7/$07	light cyan
8/$08	dark red
9/$09	light red
10/$0A	dark purple
ll/$0B	light purple
12/$0C	dark yellow
13/$0D	light yellow
14/$0E	light gray (dark white)
15/$0F	white

Bit 4 selects the flash attribute, causing the character in the corresponding character position to blink at the rate specified in bit 5 of internal register 24/$18. Bit 5 selects the underline attribute, but the line can be moved to any scan line of the character position. Bit 6 controls the reverse attribute,

.^hich reverses the foreground and background pixels of the character pattern. However, this attribute isn't used by the 128, which instead has reversed character patterns as part of the standard character sets. Finally, bit 7 selects which of the two character sets will be used. The VIC allows only one of the two character sets to be used at any one time, but the VDC allows ju to select the character set independently for each character position.

The register structure of the VDC chip is rather unusual. It has only two registers visible in the normal system address space. You must go through these registers to access any of the internal functions of the VDC. Table 8-6 lists the VDC communications registers visible to the processor. A detailed description of both locations follows.

Table 8-6. VDC External Communications Registers

54784/$D600	VDC address/status register
54785/$D601	VDC data register

54784 $D600 VDCADR

Address/status **register**

The VDC actually has two different registers at this location: a write-only address register and a read-only status register. Any value you store in this location goes to the address register. The value specifies which one of the 37 internal VDC registers can be accessed via the data register at 54785/$D601. Since only six bits are needed to specify all the valid register numbers (0-36), bits 6 and 7 of the address register are unused and have no effect when you are writing to this register.

You can't read this location to determine which internal register is being accessed; reading always returns the status register contents. The status register bits are defined as follows:

Bits 0-2: These locations hold a constant number (like a ROM location) indicating the version number of the VDC chip currently installed. There have been two versions of the VDC to date. In early 128s, the value here will be %000 = 0, while in later models the value will be %001 = 1, The exact difference between the versions is not clear, but they do require slightly different initialization. Specifically, internal register 25, the

horizontal smooth scrolling register, requires different settings for the different versions. The IOINIT routine [$E109] checks these VDC status register bits and performs the initialization accordingly.

Bits 3-4: These bits are unused, and always return %0 when read.

Bit 5: This bit, called the VBLANK flag, is used to indicate when the VDC raster scan is in its vertical blank period. After a full screen has been drawn and the electron beam has reached the lower right corner of the display, the beam must be turned off briefly while it is moved back to its home position in the upper left corner to begin the next frame. Otherwise, moving the beam would result in a diagonal line across the screen. The time while the beam is turned off for this repositioning is called the vertical blanking interval. This bit, normally %0, will be set to %1 during the vertical blanking period.

When you're programming special screen effects, it's handy to know when the blanking period is occurring. The time when one frame has been completely drawn but another one has not yet been started is a good time to change screen parameters without causing excessive flicker. The VIC chip can generate a raster interrupt to signal when its vertical blanking interval is beginning, but the VDC chip can't generate interrupt requests, so this bit provides an alternate method of signaling that vertical blanking is in progress.

Bit 6: This bit, known as the LP flag, is used to indicate when new values have been latched in the internal light pen registers. Whenever a pulse is detected on the VDC chip's LP input line, the row and column positions of the raster beam at that time are latched into internal registers 16—17/$10-$ll and this bit is set to %1. This bit will remain %1 until one or both of the internal light pen registers are read, at which time it will be reset to %0. Thus, a %1 in this bit is a signal that a new light pen position can be read from the internal registers, while a %0 indicates that the register values have not changed since they were last read. See the entry for internal registers 16-17/$10-$ll for details of light pen reading.

Bit 7: This bit, known as the STATUS flag, is used to indicate that the VDC is ready to read from or write to the internal register specified by writing to the external address register. The flag bit will be set to %0 when a value is stored in this Io-

cation, and will return to %1 when the specified register is ready for access. The standard procedure for reading from or writing to a VDC register is as follows;

Reading:

```
        LDX  #register number
        STX  $D600
WAIT    LDX  $D600
        BPL  WAIT
        LDA  $D601
```

Writing:

```
        LDX  #register number
        5TX  $D600
WAIT    LDX  $D600
        BPL  WAIT
        STA  $D601
```

Refer to the entry below for information concerning the restrictions on accessing this location from BASIC.

54785 SD601 VDCDAT

Data register

This register is the gateway to the internal registers of the VDC. After you have set the desired internal register number by writing to the address register at 54784/$D600, the specified register becomes visible at this location. Reading from this location shows the internal register contents, and writing to this location stores the value in the internal register. Once you specify an internal register, that register remains visible here until you change registers by storing a new value in 54784/ $D600. Refer to the internal register descriptions below for more information on the effects of reading from and writing to the various registers.

Commodore's specifications for this chip state that when accessing the VDC, you should avoid machine language instructions that use the indirect addressing mode. That is, you should avoid using instructions like LDA ($CC),Y to read this register because the VDC apparently responds improperly to such instructions. This imposes no particular hardship on machine language programmers, but has highly unfortunate consequences for BASIC programmers. The 128's PEEK, POKE, and WAIT instructions are implemented using the Kernal INDFET, INDSTA, and INDCMP routines, all of which use indirect-Y addressing to read or store values. As a result, *you should not use PEEK, POKE, or WAIT statements to read or change the contents of this location or of location 54784/$D600.*

This would seem to make the VDC inaccessible from BASIC. Fortunately, a pair of screen editor ROM routines provide a simple detour around this roadblock. In Chapter 7 they are designated WRITEREG [$CDCC] and READREG [SCDDA],

and they have the form shown above in the discussion of location 54784/$D600. To store a value in any VDC register from BASIC, use:

SYS DECCCDCC"), *value, register*

or

SYS 52684, *value, register*

where *register* is the number (0-36) of the internal VDC register you wish to access, and *value* is the value (0-255) you wish to place in that register. If you have used any other BANK statements in your program, it would be wise to add a BANK 15 statement before these SYS statements to insure the proper memory configuration.

To read the contents of an internal VDC register from BASIC, use a statement of the form:

SYS DECC'CDDA"),, *register* : RREG A

or

SYS 52698,, *register :* RREG A

Be sure you have two commas between the SYS address and the register number. After this statement has been executed, the variable A will contain the value read from the specified register. Again, if you change bank configurations elsewhere in your program, it would be wise to add a BANK 15 before the SYS statement.

54786-55039 $D602-SD6FF

VDC chip register images

Due to incomplete address decoding, images of the VDC chip registers appear repeatedly through the remainder of this page of memory. That is, storing a value in any even-numbered location in this range has the same effect as storing a value in 54784/$D600, and storing a value in any location in this range having an odd address has the same effect as storing a value in 54785/$D601. It might be easier to remember the address of one of the image locations, such as 55000 in place of 54784. Nevertheless, it's better programming practice to use the officially designated register addresses.

VDC Internal Registers

Table 8-7 lists the various internal VDC registers which can be accessed via the external communications register. A detailed description of each follows.

Table 8-7. VDC Internal Registers

0/$00	Total number of horizontal character positions
l/$01	Number of visible horizontal character positions
2/$02	Horizontal sync position
3/$03	Horizontal and vertical sync width
4/$04	Total number of screen rows
5/$05	Vertical fine adjustment
6/$06	Number of visible screen rows
7/$07	Vertical sync position
8/$08	Interlace mode control register
9/$09	Number of scan lines per character
10/$OA	Cursor mode control
ll/$0B	Ending scan line for cursor
12/$0C	Screen memory starting address (high byte)
13/$0D	Screen memory starting address (low byte)
14/$0E	Cursor position address (high byte)
15/$0F	Cursor position address (low byte)
16/$10	Light pen vertical position
17/$11	Light pen horizontal position
18/$12	Current memory address (high byte)
19/$13	Current memory address (low byte)
20/$14	Attribute memory starting address (high byte)
21/$15	Attribute memory starting address {low byte)
22/$ 16	Character horizontal size control register
23/$17	Character vertical size control register
24/$18	Vertical smooth scrolling and control register
25/$19	Horizontal smooth scrolling and control register
26/$lA	Fore ground/background color register
27/$lB	Address increment per row
28/$lC	Character set address and memory type register
29/$lD	Underline scan-line-position register
30/$lE	Number of bytes for block write or copy
31/$1F	Memory read/write register
32/$20	Block copy source address (high byte)
33/$21	Block copy source address (low byte)
34/$22	Beginning position for horizontal blanking
35/$23	Ending position for horizontal blanking
36/$24	Number of memory refresh cycles per scan line

0 $00

Total number of horizontal character positions

The value in this register determines the total width (in character positions) of each horizontal line of the display. The value stored here should be one less than the desired number of horizontal character positions. This total includes the active portion of the display (where characters can be displayed), the left and right borders, and the horizontal sync width. The total number of horizontal pixels is given by multiplying the value here (plus 1) by the total number of pixels per character position (the value in bits 4-7 of register 22/$16 plus 1).

The default value for this register, established during the IO1NIT routine [$E109], is 126/$7E, This provides 127 horizontal character positions. You'll need to reduce this by half if you enable the pixel double feature (see the entry for bit 4 of register 25/$19). You may need to increase the value here slightly if you use one of the interlaced modes.

1 $01

Number of active horizontal character positions

The value in this register determines how many of the horizontal character positions specified in register 0/$00 can actually be used to display characters. The value stored here should be the desired number of columns for the display. The value here must be less than the value in register 0/$00. The default value for this register is 80/$50, since the default VDC display is an 80-column text screen. The value here also determines the width of the bitmap when the VDC is set for graphic mode. The bitmap width is given by multiplying the number of character positions by the character-position width specified in bits 0-3 of register 22/$16.

The screen editor routines that support printing to the 80-column screen assume that each screen line occupies 80 screen memory locations. If you want the screen printing routines to continue to function properly after you reduce the number of active character positions in this register, you should increase the value in register 27/$lB so that the sum of the value in that register plus the value in this register remains equal to 80. Reducing the value here removes characters from the right of the display area. To center the active display area after reducing the number of character positions, you must reduce the value in register 2/$02. The screen editor routines will not

support a display wider than 80 columns, so you'll have to write your own text handling routines if you want to use a wider display.

2 $02
Horizontal sync position

The value in this register determines the character position at which the vertical sync pulse begins. The value here also determines the horizontal position of the active portion of the screen within the total display. The default value here is 102/$66. Increasing this value moves the active screen area to the left; decreasing it moves the active area to the right.

3 $03
Horizontal and vertical sync width

Bits 0-3: These bits specify the width of the horizontal sync pulse. The value here should be one greater than the desired number of character positions for the pulse. The default value for these bits is 9/$9, for a pulse eight character positions wide.

Bits 4-7: These bits specify the width of the vertical sync pulse. The bit value here should be equal to the desired number of scan lines for the pulse, unless the interlaced sync and video mode is being used (in that case, use a value that is twice the desired number of scan lines). The default value for these bits is 4/$4, for a pulse four scan lines wide.

4 $04
Total number of screen rows

This register specifies the total height (in character positions) of the VDC display. The value stored here should be one less than the desired number of vertical character positions. The total includes the rows for the active display, the top and bottom portions of the border, and the vertical sync width. To determine the height of the raster in scan lines, multiply the value in this register (plus 1) by the number of scan lines per character position (the value in register 9/S09 plus 1) and add any additional scan lines specified in register 5/$05.

The proper number of scan lines for the display is a function of the video system being used; it's different for NTSC (North American) and PAL (European) systems. During the IOINIT routine [$E109], the 128 checks the VIC chip to deter-

mine which system is being used (since the VIC isn't programmable like the VDC, there is a different version of the VIC for each of the two video systems). This register is then initialized accordingly: to 32/$20 for NTSC systems or 39/$27 for PAL systems, selecting 33 or 40 rows, respectively. Since the default character-position height is eight scan lines, the respective total heights are 33 * 8, or 264 lines, for NTSC, and 40 * 8, or 320 scan lines, for PAL. These scan-line totals should remain constant, so if you increase the character height you must decrease the total number of rows, and vice versa.

5 $05

Vertical fine adjustment

Bits 0-4: The total number of scan lines in the VDC's video display should be 264 for an NTSC (North American) system or 320 for a PAL (European) system. The number of scan lines used in the VDC display is given by the total number of vertical positions (specified in register 4/S04) multiplied by the number of scan lines per character position (specified in register 9/$09). If the result doesn't come out exactly equal to the required 264 or 320, the VDC can add a few extra scan lines at the end to achieve the proper result. The value in this register specifies the number of extra scan lines to add. The available five bits allow up to % 11111 = 31/$1F additional scan lines.

The default character height of eight scan lines is an exact multiple of both 264 and 320 (33 * 8 = 264 and 40 * 8 = 320). Thus, no extra scan lines are required, so this register is initialized to 0/$00 by the Kernal IOINIT routine [$E109]. As an example of the use of this register, assume that you increased the character height to nine scan lines. For an NTSC system, 264 / 9 = 29 with a remainder of 3. Thus, for this case you should specify 29 for the total number of vertical character positions and store a 3 in this register to provide the required additional scan lines.

Bits 5-7: These bits are unused; writing to them has no effect, and they always return %1 when read. Thus, the value you read from this register **will** always be at least 224/$E0. To mask off these bits and see only the valid bits of the register, use AND 31 in BASIC or AND #$1F in machine language.

6 $06

Number of visible screen rows

The value in this register determines how many of the vertical character positions specified in register 4/S04 can actually be used to display characters. The value here must be less than the total number specified in register 4/$04. The default value established for this register by the Kernal IOINIT routine [$E109] is 25/$19, which sets up the standard 25-row display. One obstacle to selecting other numbers of rows is that the screen editor ROM routines will, by default, assume a 25-line screen. When decreasing the number of rows, you can make the screen editor use the reduced number by storing a value equal to the new number of rows minus 1 in location 237/ $ED, then resetting the output window to full screen size (by printing two cursor-home characters, for example). The screen editor routines will not support a display with more than 25 rows, so you'll have to provide your own character manipulation routines to use such a screen.

7 $07

Vertical sync position

The value in this register determines the vertical character position at which the vertical sync signal will be generated. This register can be used to adjust the vertical location of the active display area within the screen. The default value for this register, established by the IOINIT routine [$E109], is 29/S1D for NTSC (North American) systems or 32/S20 for PAL (European) systems. Decreasing the value here will move the active display area down the screen, while increasing the value will move the active display area upwards. However, you should not increase the value here above the maximum number of rows specified in register 4/$04.

8 $08

Interlace mode control

Bits 0-1: The value in these bits controls the interlace mode of the screen. The complete standard for NTSC video calls for a frame (raster) of 525 lines to be redrawn 30 times per second, while the PAL standard calls for 625 lines redrawn 25 times per second. The full screen isn't drawn all at once; instead, it's drawn in two passes with half the lines for the frame drawn on each pass. The lines for the second pass of the frame are

drawn between the lines for the first. Like most computer displays, the VDC normally takes a shortcut and draws half the full number of lines at twice the rate. This noninterlaced display provides sufficiently sharp output for most uses. However, the VDC is also capable of producing two interlaced display modes. The modes are selected as follows:

Bits			
I	0	Value	Interlace mode
x	0	0 or 2	noninterlaced
0	1	1	interlaced sync
II		3	interlaced sync and video

The default value is 0/$00, which selects the standard noninterlaced mode. The system never uses any other mode. In the interlaced sync mode, the number of scan lines is doubled. Each horizontal scan line is drawn twice, once on the first pass and very slightly lower on the second pass. The result should be greatly improved vertical resolution, but you'll probably be disappointed. Remember that each tiny dot on the screen glows only very briefly after being struck by the raster video beam. Since this mode must draw twice as many lines, it draws each line only half as often. On most video monitors, the first set of lines will have started to fade before the second set is completely drawn. As a result, the screen will appear to jitter annoyingly in this mode.

The remaining choice is interlaced sync and video mode. In this case, the VDC also draws twice as many lines as noninterlaced mode, but the alternating half-frames are independent, so you can use twice as many horizontal lines per frame (the maximum screen height in scan lines is double that of the noninterlaced mode). This creates the tantalizing prospect of an 80-column X 50-line text screen. The following routine sets up such a display:

```
10  WR=DEC("CDCC")
20  SYS  WR,3,8
30  SYS  WR,64,4
40  SYS  WR,50,6
50  SYS  WR,58,7
60  SYS  WR,128,0
```

However, this setup does have limitations. It suffers from the same jitter problems as the other interlaced mode. Furthermore, the screen editor routines that control printing to the

screen will not support a display longer than 25 lines, so you'll have to write your own text manipulation routines to handle the extra 25. Nevertheless, this mode has interesting possibilities.

Bits 2-7: These bits are unused; writing to them has no effect, and they always return %1 when read. Thus, the value you read from this register will always be at least 252/$FC. To mask off these bits and see only the valid bits of the register, use AND 3 in BASIC or AND #$03 in machine language,

9 $09
Total number of scan lines per character

Bits 0-4: These bits determine the total vertical height (in scan lines) of each character position. The value stored here should be one less than the desired number of scan lines. The total vertical height value includes the scan lines for the active portion of each character position, plus any desired number of blank scan lines for intercharacter vertical spacing. The height of the active portion of the character position is determined by the value in register 23/$17.

The default value for this register, established during the IOINIT routine [SE109], is 7/$07, for a total character-position height of eight scan lines. In this case, there will be no vertical intercharacter spacing because this is less than the active character height. (In the default character set, intercharacter spacing is achieved by leaving the bottom row of the character pattern blank.) The five available bits allow values up to %11111 = 31/$IF, for character-position heights of up to 32 scan lines. However, when changing this value to allow for greater vertical resolution, you must keep in mind that the value here multiplied by the total number of rows specified in register 4/$04 (and plus the number of extra scan lines specified in register 5/$05) determines the number of scan lines in the display. This total should always be 264 lines for NTSC (North American) systems or 320 lines for PAL (European) systems.

The value here also determines how much memory is required for character pattern memory. While the value here is less than or equal to 15 (while the character height is 16 or fewer scan lines), each character pattern is allocated 16 bytes. Since the VDC supports two complete 256-character sets, a total of 512 * 16, or 8192 bytes, are required for character memory' However, if the character height exceeds 16 scan lines (if

the value here is greater than 15), then 32 bytes are allocated for each character pattern. In this case, 512 * 32, or 16,384 bytes, are required for character memory. Note that this is all the memory available to the VDC.

Bits 5-7: These bits are unused; writing to them has no effect, and they always return %1 when read. Thus, the value you read from this register will always be at least 224/$E0. To mask off these bits and see only the valid bits of the register, use AND 31 in BASIC or AND #$1F in machine language.

10 $0A

Cursor mode control

The value in this register cannot be changed directly while the standard screen editor ROM routines are used for printing. The contents of a shadow location at 2603/$OA2B are copied to this register each time the cursor position is updated. Thus, to change the value in this register, you must store the desired value in the shadow location rather than in the register.

Bits 0-4: The value in these bits determines the scan line within each character position for the top of the cursor. Scan lines within character positions are numbered beginning with 0 for the top line of the position. Bits 0-4 of register ll/$0B determine the bottom line, and together these registers determine the height of the cursor. The available five bits allow starting row numbers as large as %11111 = 31/$1F. The default value for these bits is %00000, to start the cursor at character scan line 0, the top line of the character, for the standard full-height block cursor. The operating system also supports an underline cursor, selected by printing ESC U [$CAFE], In this case, the value here is changed to %00111 (7) to start the cursor on the bottom line of the standard character position. The value for the top scan line should be no greater than the maximum number of scan lines specified in register 9/$09, or else the cursor will not be visible.

Bits 5-6: These bits control the type of cursor provided. Unlike the VIC, where the cursor is an effect maintained by software, the VDC has hardware to generate a cursor automatically. The possible modes are as follows:

Bit values	Cursor mode
0/$00	solid {nonblinking) cursor
32/S20	no cursor
64/$40	blinking at 1/16 screen refresh rate
96/$60	blinking at 1/32 screen refresh rate

The default setting for these bits is %11, specifying a cursor blinking at the slower of the two rates. The operating system also supports a nonblinking cursor, selected by printing ESC E [$CB0B]. In this case, the bits are changed to %00. To rum the cursor off when the system is not accepting input (as when a program is running), these bits are reset to %01 [$CD9F].

Bit 7: This bit is unused; writing to it has no effect, and it always returns %1 when read. Thus, the value you read from this register will always be at least 128/$80. To mask off this bit and see only the valid bits of the register, use AND 127 in BASIC or AND #$7F in machine language,

11 $0B

Bottom scan line for cursor

Bits 0-4: These bits determine the scan line within a character position for the bottom of the cursor. Together with bits 0-4 of register 10/$0A, this serves to determine the height of the cursor. The value here should be one greater than the desired bottom scan line (scan-line numbering starts with 0 for the top scan line of the character position). The five available bits allow values up to % 11111 = 31/$1F, so the cursor can go as low as scan line 30, However, the actual displayed cursor height will never be greater than the character-position height specified in register 9/$09. The default value for this register, established by the IOINIT routine [$E109], is 7/$07, so the normal bottom scan line of the cursor is scan line 6 of the character position.

Bits 5-7: These bits are unused; writing to them has no effect, and they always return %1 when read. Thus, the value you read from this register will always be at least 224/$E0, To mask off these bits and see only the valid bits of the register, use AND 31 in BASIC or AND #$1F in machine language.

12 $0C
13 $0D

Starting address for screen memory

For standard text mode, the value in this register pair determines the starting address for screen memory, the area which holds screen codes specifying which character will be displayed in each screen position. The size of the screen memory area is determined by the number of active horizontal positions (specified in register l/$01) multiplied by the number of active rows (specified in register 6/$06) and the address increment per row (specified in register 27/$lB). The order of bytes for the pair is opposite that normally used in the 128 system: The first register (12/$0C) holds the high byte and the second (13/$0D) holds the low byte. Unlike VIC screen memory, which must begin on an even IK address boundary, VDC screen memory can begin at any address in the VDC's address space. See Figure 8-16 for a diagram of the VDC's memory configuration.

For graphic mode, the value in this register pair determines the starting address for the bitmap of the graphic screen. The amount of memory required for the bitmap is found by multiplying the number of horizontal character positions (from register l/$01) by the number of vertical character positions (in register 6/$06) times the total height of each character position (from register 9/$09 plus 1). The bitmap can be started at any address in the VDC address space.

Even if you change the value here, the screen editor ROM routines will continue to assume that screen memory is located in its default position unless you also change the value in the screen memory starting-page pointer at 2606/$0A2E.

14 $0E
15 SOF

Address of current cursor position

For the VIC chip's display, the cursor is an effect laboriously maintained by software. The VDC, by contrast, has hardware to maintain the cursor for its display automatically. The cursor will appear at the character position with the screen memory location specified in this register pair. If the address specified here is outside the area of VDC memory currently being used for screen memory, no cursor will be visible. Other characteristics of the cursor such as its blinking status and position within the character are specified in registers 10/$OA and 11/$OB.

The value in these registers cannot be changed directly while the normal screen editor 80-column printing routines are in use. The printing routines update the cursor position after each character is printed so that these registers always hold the address of the next available character position.

16 $10
17 S11

Light pen vertical and horizontal positions

Whenever the LP input line to the VDC chip is brought to a low (0 volts) state, the row and column values for the current position of the raster beam are latched into these registers. The vertical (row) number will be latched into register 16/$10, and the horizontal (column) number will be latched into register 17/$11. To signal that a value has been latched, the LP flag (bit 6 of the external register at 54784/SD600) will be set to %1. That bit will remain at %1 until either of these registers is read, at which time it will be reset to %0. However, reading these registers does not clear them; the latched values will be retained until the LP line is brought low again.

The VDC's LP line is connected to pin 6 of control port 1 (control port 2 does not support a light pen). A light pen has at its tip an electronic device known as a phototransistor, which is connected so as to cause a low pulse whenever the video beam moves past the pen. Note that the pen will not be triggered if the screen position is black or one of the other dark colors. Only positions which have bright characters can be read. The ideal character to read with a light pen is a white re verse-video space.

When a light pen is used, the range of values in these registers depends on the screen width and height selected by other VDC registers. Unlike the VIC chip, whose light pen registers return scan line and dot position values, these registers return row and column numbers corresponding to the light pen position. This makes the results much easier to interpret, but does not allow precise positioning, so it is unlikely that you'll see any 80-column drawing programs using the light pen as an input device. For the standard 80-column X 25-line screen, the value in register 16/$10 corresponds very closely to the row number: ranging from l/$01 at the top of the screen to 25/$19 at the bottom. Actually, you may find

that if you position the pen slightly below the bottom screen line you can get a reading of 26/S1A.

While the vertical resolution is good, the horizontal resolution is quite poor. The horizontal reading won't correspond to the row number (1-80). Instead, it corresponds approximately to the absolute horizontal character position, which includes the border areas on the left and right edges of the screen. You should find that when the pen is pointed at the leftmost character position, you get a reading of about 27-29 in register 17/$11. This implies that the rightmost character position should give readings of about 106-108. Actually, you may get higher readings—120 or more. In fact, even if you hold the pen perfectly stil] you may see the character position vary up or down by 4 or 5. The moral is that the light pen is much better at reading vertical than horizontal positions. You'll have better luck if you limit yourself to checking whether the pen is within a range of horizontal positions. For example, if you read the horizontal position and store the result in the variable H, then an expression such as H = INT({H — 30) / 8) will return a range of values 0-9 indicating roughly which one of ten eight-column horizontal areas the pen is pointing to.

You should be aware that these registers can be tricked into reading false values. Pin 6 of control port 1 is also used for light pen input for the VIC chip, and a light pen signal generated on the 40-column screen will latch meaningless values in these registers. In lieu of a light pen, several other events can cause a pulse on the LP line. That control port pin is also used for the joystick fire button, so pressing the button of a joystick plugged into port 1 will also latch values in these registers. Because of this joystick button function, the port line is also connected to the line from row 4 of the keyboard matrix. This has two consequences. First, pressing any of the following keys with no light pen connected will latch meaningless values: Fl, Z, C, B, M, period, right SHIFT, space, the 2 and ENTER keys on the numeric keypad, and the ^ key in the cursor group. More significantly, while a light pen is connected, all of these keys will be "dead," and cannot be typed.

1[8] §J2
19 **$13**

Current memory address

This register pair specifies which address in the VDC's private block of RAM will be referenced by the next read or store operation involving register 31/S1F. As with the other VDC address register pairs, the first register (18/$12) holds the high byte of the address and the second (19/$13) holds the low byte. A value stored in register 31/$1F is transferred to the VDC memory location specified in this register pair. Reading register 31/$1F returns the value in the location in VDC memory with the address specified in this register pair. For copy or fill operations, the value in these registers determines the destination address for the operation. These registers are auto-incrementing, meaning that the address value here is automatically increased by 1 after each read or store to register 31/S1F. Thus, when you wish to read or load a continuous series of VDC memory locations you only need to set the memory address in these registers before the first read or store. After that, you can just read from or write to register 31/$1F and the address will be handled automatically.

20 $14
21 $15

Starting address for attribute memory

When attributes are activated, this register pair determines the starting address for attribute memory, the area which holds attribute values for each active character position on the screen. (Attributes can be turned on and off by setting bit 6 of register 25/$19.) The size of the attribute memory area depends on the number of active rows and columns specified in registers 2/$02 and 6/$06, and will be the same as the size of the screen memory area. See the discussion of attributes in the introduction to this section for more information.

These locations are initialized to 2048/S0800, the default starting address for attribute memory. Like the other address pairs in the VDC, the first register (20/$14) holds the high byte and the second (21/$15) holds the low byte. Attribute memory can start at any address within the VDC's address space. Even if you change the value here, the screen editor ROM routines will continue to assume that attribute memory

is in its default position unless you also change the value in the attribute starting-page pointer at 2607/$0A2F.

22 $16

Character horizontal size control

Bits 0-3: The value in these bits determines how many of the total horizontal pixels in the character position will be used to display character pattern data. (The total number is specified in bits 4-7 of this register.) If the number of active pixels is less than the total number of pixels, the extra pixels will be blank for intercharacter spacing. If you specify a value here that is greater than the total number of pixels available for the position, only the specified total number of pixels will be visible. However, the value here should not exceed 8, since a maximum of eight bits are available per byte of character pattern data. Even for values greater than 8, no more than eight pixels will be active per horizontal scan line within the character position. For graphic mode, the value here should be equal to the total number of pixels; otherwise there will be gaps in the display.

The default value for these bits is 8, for eight active horizontal pixels per character-position scan line. This is the same as the total number of pixels per position, so there will be no intercharacter spacing. (For the default character set, the rightmost column of each character pattern is left blank to provide the effect of intercharacter spacing,)

Bits 4-7: The value in these bits determines the width of each character position (in pixels). The value stored here should be one less than the desired total number of pixels. If the total is greater than the number of active pixels specified in bits 0-3 of this register, the extra pixels will be blank for intercharacter spacing. The default value for these bits is 7, for eight total pixels per character position. The total number of horizontal pixels is determined by multiplying the value here (plus 1) by the total number of character positions (from register 0/$00).

23 $17

Character vertical size control

Bits 0-4: The value in these bits determines how many of the total scan lines for each character position (specified in register 9/$09) will be used to display character pattern data. The

available five bits allow you to specify values up to %11111 = 31/$1F. If the value here is less than the total number of scan lines for the character position, the extra lines will be blank for intercharacter spacing. If the value here is greater than the total number of scan lines, only the total number of scan lines will be displayed. For graphic mode, the value here should be at least equal to the total number of scan lines (the value in register 9/$09 plus 1); otherwise there will be gaps in the display.

The default value stored in this register is 8/$08, for eight active scan lines per character position. This is equal to the default total number of scan lines for the position, so there will be no intercharacter spacing. (For the standard character set, intercharacter spacing is achieved by leaving the bottom row of most character definition patterns blank.)

Bits 5-7: These bits are unused; writing to them has no effect, and they always return %1 when read. Thus, the value you read from this register will always be at least 224/$E0. To mask off these bits and see only the valid bits of the register, use AND 31 in BASIC or AND #$1F in machine language.

24 $18

Vertical smooth scrolling and control

Bits 0-4; These bits can be used to smoothly scroll the screen vertically upward. The value here specifies the number of scan lines the display should be shifted upward. Although five bits are available, the value here should not exceed the value in register 9/$09.

Bit 5: This bit controls the blinking rate for all characters on the screen with the flash attribute. A character position has the flash attribute when bit 4 of its corresponding attribute memory is set to %1. The two available blinking rates are once each 16 times the screen is refreshed (selected when this bit is set to %0) or once each 32 times (selected when this bit is set to %l). For NTSC (North American) systems, the screen is redrawn 60 times per second, so the corresponding blinking rates are about four times per second when the bit is %0 and about twice per second when the bit is %1.

The default setting for this bit is %1, for the slower blinking rate. This is established during the IOINIT routine [$E109],

part of both the reset and RUN/STOP-RE STORE sequences. This setting is not changed by any other ROM routine.

Bit 6: This bit controls a special VDC feature known as reverse mode. While this bit is %0, all pixels on the screen represented by %0 bits in character patterns or the graphic screen bitmap take the background color specified in bits 0-3 of register 26/$lA, and all pixels represented by %1 bits take the foreground color specified in a corresponding attribute memory location (or in bits 4-7 of register 26/$lA if attributes are disabled). Setting this bit to %1 reverses the color sources, so that all pixels for %0 bits take the foreground color and all pixels for %1 bits take the background color.

This bit is initialized to %0 for a normal screen display. The screen editor ROM supports escape sequences to change this bit. The ESC R sequence will set the bit to %1, reversing the screen display. The ESC N sequence will clear the bit to %0, returning the display to normal.

Bit 7: This bit determines whether the next block operation initiated by writing to register 3O/$1E will be a copy or a fill. Setting this bit to %0 specifies a fill operation, while setting it to %1 specifies a copy operation. See the entry for register 3O/$1E for more information on VDC block operations. This bit is set to %0 when the register is initialized during the IOINIT routine [$E109].

25 $19

Horizontal smooth scrolling and control

Bits 0-3: These bits can be used to smoothly scroll the screen horizontally. The use of these bits depends on the version of the VDC in your 128. (The version number can be determined by reading bits 0-2 of the external communications register at 54784/$D600.) For version 1 of the VDC, which includes most 128s, this register should be initialized to the maximum character width (in bits 4-7 of register 22/$16). Each decrement of this register shifts the display one pixel to the left. For the older version 0 of the VDC, these bits should be initialized to %0000. In this case, each increment of these bits shifts the display one pixel to the right.

Bit 4: This bit controls the VDC's pixel double feature. While this bit is %0, pixels will be their normal size. Setting this bit to %1 will double the size of all horizontal screen pixels. Since

each pixel is twice as large, there will be room for only half as many on the screen. Thus, you must reduce the values in the horizontal screen width registers (0-2/$00-$02) to half their normal values. The following routine sets up a VDC 40-column display that will be very similar to the VTC's 40-column display:

```
10 WR=DEC("CDCC"):RR=DEC("CDDA")
20 SYS WR,63,0:SYS WR,40,1:SYS WR,55,2
30 SYS RR,,25:RREG A:SYS WR,(A OR 16),25
40 SYS WR,(8*16)+9,22
50 SYS WR,40,27
60 POKE 238,39
70 PRINT"{2 HOMEllCLR}"
```

Bit 5: This bit controls a special VDC feature called semigraphic mode. When semigraphic mode is activated (when this bit is %1), the rightmost active pixel will be repeated through the intercharacter spacing pixels. For this mode to have any visible effect, there must be some intercharacter spacing (the value in bits 0-3 of register 22/$16 must be less than the total number of pixels specified in bits 4-7 of that register). This mode has no effect in graphic mode. One use of this mode is to create a simple "digital" character effect. Try these lines:

```
SYS 52684,118,22
SYS 52698,,25: RREG A: SYS 52684,(A OR 32),25
```

Bit 6: The VDC has two methods of supplying foreground information for its display. When this attribute enable bit is set to %1, each character position will have a corresponding attribute memory location. Refer to the introduction to this section for details on attributes. The starting address of attribute memory is determined by the value in registers 20-21/$14-$15, When this bit is %0, attribute memory is not used. Instead, all character positions take the foreground color specified in bits 4-7 of register 26/$1A. In this case, the character positions cannot have the flash, underline, or reverse attributes, and only the first of the two character sets will be available.

Bit 7: This bit determines whether the VDC will operate as a text or graphics display. Text mode, selected when the bit is set to %0, is the only one supported by the 128 operating system (%0 is the default value for this bit). In that mode, each screen memory position holds a screen code which serves as

an index into character memory to specify the pattern to be displayed in that position.

When this bit is set to %1, graphics mode is selected. In that mode, screen memory is replaced with a bitmap. (There is no cursor on the graphic display.) Each bit in the bitmap controls the state of one pixel in the display. The layout of the bitmap is much simpler than that for'the VIC screen. Each horizontal scan line is controlled by a sequential series of bytes. The size of the bitmap (in bytes) is determined by the number of active horizontal positions times the number of vertical positions times the number of scan lines per vertical position. For the standard screen setup, this means that 80 * 25 * 8, or 16,000 bytes, are required—almost all of the available VDC memory. At eight pixels per byte, there are 128,000 individual pixels on the graphic display.

The graphic display can use attribute memory for color information. In this case, the relationship of attribute locations to bitmap positions is similar to that for the VIC screen. Each attribute memory location controls the color for all pixeis within a character-position area on the screen. However, there isn't enough free memory available for a full bitmap and a full attribute memory area. One solution is to turn off attributes (set bit 6 of register 25/$19 to %0). This limits all screen positions to the same foreground and background colors (as specified in register 26/$lA). The other solution is to reduce the size of the active screen area to free up enough memory for an attribute area. For example, if you reduce the number of active rows to 22, then 80 * 22 * 8, or 14,080 bytes, will be required for the bitmap, and 80 * 22, or 1760 bytes, will be required for attribute memory, so there will be enough room within VDC memory for both bitmap and attributes.

When attribute memory is enabled for a graphic display, the lower four bits of each attribute memory location determine the color of all foreground (%1) pixels in the corresponding character-position area, and the upper four bits determine the color of all background (%0) pixels in the character position.

Program 8-1 is a very simple example of a bitmapped drawing program for the VDC. Use a joystick in port 2 to sketch. Pressing the B key will change the background color and pressing the F key will change the foreground color (attribute memory is turned off, so all character positions use

program 8-1. 80-Column Sketchpad

```
100 GRAPHIC 0:FAST
110 WR=DEC("CDCC"):RR=DEC("CDDA")
120 SYS RR,,25:RREG A:SYS WR,(A AND 63)OR 128,25
130 SYS RR,,24:RREG A:SYS WR,A AND 127,24
140 SYS WR,0,13:SYS WR,0,19:SYS WR,0,31
150 FOR 1=0 TO 63:SYS WR,255,30:NEXT 1
160 X=320:Y=100:BC=0:FC=15
170 GET KS:ON INSTR( "BF{HOME HCLR j " , KS ) GOTO 180,1
   80,160,140:GOTO 190
180 BC={BC-(K?="B"))AND 15:FC=(FC-(K$="F"))AND 15:
   SYS WR,FC*16+BC,26
190 D=JOY(2) AND 15:IF D=0 THEN 170
200 Y=Y+(D<3 OR D=8)-(D>3 AND D<7);IF Y<0 THEN Y=l
   99:ELSE IF Y>199 THEN Y=0
210 X=X-(D>1 AND D<5)+{D>5):IF X<0 THEN X=639:ELSE
    IF X>639 THEN X=0
220 AD=(Y*80)+INT(X/8):AH=INT(AD/256):AL=AD-(AH*25
   6)
230 SYS WR, AH ,18:SYS WR,AL,19
240 SYS RR,,31:RREG A
250 SYS WR,AH,18:SYS WR,AL,19
260 SYS WR,A OR 2T(7-(X AND 7)),31
270 GOTO 170
```

the same foreground and background colors). The CLR/HOME key can be used to move the drawing point back to its home position in the center of the screen, and SHIFT - CLR/HOME will clear the display.

26 $1A

Background and foreground colors

Bits 0-3: The value in these bits determines the background color of the display. For text mode, this is the color of all pixels represented by %0 bits in the pattern definition for the character in each screen position. For graphic mode with attribute memory disabled, the value here determines the color °f all pixels represented by %0 bits in the bitmap. The correspondence between register value and background color is as shown in Table 8-5. For graphic mode with attribute memory enabled, the value here determines the color of the screen border only.

The default background color value, 0/$00 (black), is established by the Kernal IOINIT routine [$E109], part of both the reset and RUN/STOP-RESTORE sequences. From BASIC,

the background color can be changed using the statement COLOR 6, *color number*. However, the values for the color number parameter are not the same as the color values shown in Table 8-5. Refer to the description of the COLOR statement in the *System Guide* that came with your 128 for more information.

Bits 4-7: When attributes are disabled (by setting bit 6 of register 25/$19 to %0), the value in these bits specifies the foreground color for the display. For text mode, this is the color for all pixels represented by %1 bits in the pattern definitions for all screen positions. For graphic mode, the value here determines the color of all pixels represented by %1 bits in the bitmap. For either mode, if the screen is switched to reverse mode (by setting bit 6 of register 24/S18), the value here will instead determine the color for all pixels represented by %0 bits in the character pattern or bitmap. The correspondence between bit values and colors is as shown in Table 8-5.

27 $1B

Address increment per row of characters

The value in this register will be added to the value in register 1/S01 to determine the amount by which to increase the screen memory address for each new row of the display. This allows you to set up a virtual screen wider than the actual screen. You can scroll back and forth across the virtual screen by adjusting the screen starting address in registers 12-13/$0C-$0D.

The default value for this register is 0/$00, since no extra columns are used with the 80-column text display. The screen editor routines that support printing to the VDC screen all assume an 80-column screen line. If you reduce the number of active columns in register l/$01, you should increase the value in this register correspondingly so that the total remains 80.

28 $1C

Character pattern address and memory type

Bits 0-3: These bits are unused; writing to them has no effect, and they always return %1 when read. Thus, the value you read from this register will always be at least 15/SOF. To mask off these bits and see only the valid bits of the register, use AND 240 in BASIC or AND #$F0 in machine language.

Bit 4: This bit specifies the type of RAM chip used for VDC video memory. When the bit is %0, the VDC is configured for 4416 chips (16K X 4 bits). When the bit is %1, the VDC is configured for 4164 chips (64K X 1 bit). Since the 16K VDC video memory space in the 128 is provided by two 4416 chips, this bit is initialized to %0 by the Kernal IO1NIT routine [$E109]. It is theoretically possible to replace the existing chips with the 64K variety to quadruple the amount of available VDC RAM. However, the swap involves unsoldering the existing chips from the circuit board and soldering the new ones in their place. This is not a task for the inexperienced, and will most certainly void any warranty on your 128.

Bits 5-7: These bits determine where within VDC memory the character pattern definitions will be located. The amount of memory required for the character set depends on the value in register 9/$09. If the character height is 16 or fewer scan lines, each character set requires 4K (4096 bytes). Character heights of 17-32 scan lines require 8K (8192-b'yte) character sets. The VDC normally supports a pair of character sets, using bit 7 of the attribute memory location to select between them for each character position. Thus, 8K is normally used for character sets when the character height is 16 or fewer scan lines, and 16K is used when the character height is greater than 16 scan lines. In the latter case, bit 5 is not used in the address selection. The possible starting addresses for character patterns are as follows (the asterisks indicate valid selections for 16K-character set pairs):

Bits	Character memory
7 6 5	starting address
0 0 0	0/$0000 *
0 0 1	8192/$2000
0 1 0	16384/$4000 *
0 1 1	24576/$6000
1 0 0	32768/$8000 *
1 0 1	40960/SAOOO
1 1 0	39152/SC0O0 *
1 1 1	57344/$E000

Since the 128 has only 16K of RAM for the VDC, only the first two settings are currently valid. (Note that there is insufficient room in the 128's 16K of VDC video memory for a 16K character set plus screen and attribute memory.) These bits are initialized to %001 by the IOINIT routine [$E109], part of the

reset and RUN/STOP-RESTORE sequences, so the default character set starting address is 8192/$2000. Since this area is RAM, not ROM, it is necessary to copy character patterns into this area of memory if the VDC is to display recognizable characters. This step is performed during the IOINIT routine by calling the screen editor INIT80 routine [$CE0C].

29 $1D

Underline scan-line control

Bits 0-4: The value in these bits determines which scan line within the character position will be filled for any characters with the underline attribute. (A character position has the underline attribute when the corresponding attribute memory position has bit 5 set to %1.) Since the line can appear on any horizontal scan line of the character position, it's not strictly correct to call it an underline. For example, you could move the line to the top line of the position to be an overbar, or to the middle line of the position to serve as an overstrike. Scan line 0 is the top line of the character position. The available five bits allow a maximum scan-line value of %11111 = 31/$1F. However, the underline will not be visible if the value is greater than the maximum character-position height in bits 4-7 of register 22/$16.

Bits 5-7: These bits are unused; writing to them has no effect, and they always return %1 when read. Thus, the value you read from this register will always be at least 224/$E0. To mask off these bits and see only the valid bits of the register, use AND 31 in BASIC or AND #$1F in machine language.

30 $1E

Number of bytes to copy or fill

The VDC has the capability to copy blocks of data up to 255 characters long from one area of VDC memory to another, and to fill areas up to 255 bytes long with a specified value. The value in this register determines the number of bytes to be copied or filled. The copy or fill operation begins immediately after the count value is stored here. The setting of bit 7 of register 24/$18 determines whether the operation will be a copy

or a fill. The operations require different preparatory steps, as outlined below:

For a fill operation:

1. Set bit 7 of register 24/S18 to %0 to indicate a fill operation.
2. Load registers 18-19/$12-$13 with the starting address of the area to be filled (the destination area).
3. Store the value with which the area is to be filled in register ' 31/$1F. This will fill the first location.
4. Store the number of bytes to be filled, minus one because of the store performed in step 3, in register 3O/$1E. This will initiate the block fill operation.

For a copy operation:

1. Set bit 7 of register 24/$18 to %0 to indicate a fill operation.
2. Load registers 18-19/$12-$13 with the starting address of the area to be filled (the destination area).
3. Store the value with which the area is to be filled in register 31/$1F. This will fill the first location.
4. Store the number of bytes to be filled, minus one because of the store performed in step 3, in register 3O/$1E. This will initiate the block copy operation.

Either operation can be performed repeatedly to copy or fill areas larger than 255 bytes. The destination address registers (18-19/$12-$13) and, for copy operations, the source address registers (32-33/$20-$21) increment automatically each time a location is copied or filled, so upon completion of one copy or fill operation they will hold the address of the first byte beyond the area affected by the copy or fill. Thus, it is not necessary to reload the address registers to copy or fill more subsequent memory locations, nor is it necessary to set the operation flag or to load the data register after the first block. (For multiple fill operations, the instruction above to subtract 1 from the desired number of bytes to fill applies only for the first block.)

31 $ 1 F

Memory read/write

This register is the gateway between the VDC's private block of RAM and the rest of the 128 system. When read, this

location reflects the contents of the VDC memory location addressed in registers IS—19/$12—$13. Writing a value to this register will cause the value to be transferred to the location addressed in registers 18—19/$12—13. For both reading and writing, the address in registers 18-19/$12-$13 will automatically be incremented after this register is accessed. Thus, to read or write a sequential series of locations you need only load the starting address of the series into registers 18—19/$12—$13. You can then read or write repeatedly to register 31/$1F; the destination address will automatically increment after each read or write.

32 $20
33 $21

Source address for block copy

The VDC has the capability to copy blocks of data up to 255 bytes long from one area of memory to another (see the entry for register 3O/$1E for details). The value in this register pair determines the source address for copy operations, the address from which data will be copied. Like all other address register pairs in the VDC, the first register (32/$20) holds the high byte of the address and the second (33/$21) holds the low byte—the opposite of the normal 8502 address format. The registers should be loaded with the desired source starting address before the copy operation is initiated. Upon completion of the operation, the registers will hold the address of the next location beyond the last one involved in the operation. Thus, it is possible to copy blocks of more than 255 successive bytes by using repeated copy operations without reloading these registers.

34 $22
35 $23

Horizontal blanking positions

The VDC can adjust its horizontal blanking interval to blank a portion of the screen. These locations control the horizontal width and position of the blanked area. If the blanked area extends onto the active portion of the screen, any text under the blanked area is only covered, not erased. The value in register 34/$22 determines the rightmost blanked column, and the value in register 35/$23 determines the leftmost blanked column. The blanked area extends the entire height of the screen.

The value in register 34/$22 must be less than the value in register 0/$00; otherwise, the entire display will be blanked. The value in 34/$22 here must also be greater than the value in register 35/$23 to prevent an entirely blank display. The default values for these locations are 125/S7D and 100/$64, respectively. This positions the blanking interval entirely outside the active screen area. For purposes of blanking an area of the screen, a value of 6/$06 in these registers corresponds to the leftmost column of the standard screen and a value of 85/$55 corresponds to the rightmost column.

36 $24

Number of memory refresh cycles per scan line

Bits 0-3: The value in these bits determines the number of memory refresh cycles per scan line. The RAM chip used for the VDC's video memory is a type known as dynamic RAM. A dynamic RAM can hold data only briefly without external support. Just as the image on the video screen must be constantly redrawn to keep it from fading away, dynamic RAM must be constantly refreshed to keep it from losing its contents. The VDC handles this refresh function automatically for its video RAM, just as the VIC automatically handles the refreshing of system RAM. However, for the VDC, the number of refresh cycles provided during each scan line is programmable. The IOINIT routine [SE109] initializes these bits to %0101 for five refresh cycles per scan line, and there's no reason to change that setting.

Bits 4-7: These bits are unused; writing to them has no effect, and they always return %1 when read. Thus, the value you read from this register will always be at least 240/SFO. To mask off these bits and see only the valid bits of the register, use AND 15 in BASIC or AND #$0F in machine language.

37-63 $25-83F Unused

Since the external address register at 54784/$D600 allows a six-bit register number, these register addresses can also be specified. However, none of these internal registers are used, and writing to them has no effect. All register numbers in this range return the value 255/$FF when read.

Unused I/O Area

55040-55295/$D700-$D7FF

This area is described in Commodore literature as a reserved I/O expansion slot, but no currently available devices use these locations. All locations in this area of the I/O block return unpredictable changing values, and storing values here has no effect.

VIC Color RAM

5S296-56319/$D800-$DBFF

When the I/O block is selected, this IK area holds color information for the VIC 40-column video chip. Unlike the other portions of the I/O block, this area is RAM memory, not hardware chip registers. Since the VIC chip can generate only 16 different colors, all possible color values can be specified in just four bits (0-15 - %0000-%1111). Thus, the RAM in this area does not consist of the usual eight-bit bytes, but rather of four-bit half-bytes, also called *nybbles.* When you read a location in this area, the upper four bits of the location's contents are meaningless, random values which may be different each time the location is read. Those bits should be masked off to determine the true color value (use **AND** 15 in BASIC or **AND** #$0F in machine language). When you store values in these locations, only the lower four bits of the stored values are significant (POKE 55300,1 and POKE 55300,241 have the same effect).

In the standard character display mode (GRAPHIC 0), each character position in screen memory has a corresponding location in this area which determines the color of the screen dots drawn for any %1 bits in the character pattern specified for that position. Since the screen background color shows in any %0 bits in the character pattern, the color specified in color memory is referred to as the foreground color for the character position. The colors resulting from the possible values in these nybbles are as follows:

Value	Color
0/$00	black
l/$01	white
2/$02	red
3/$03	cyan
4/$04	purple
5/$05	green
6/$06	blue
7/$07	yellow
8/$08	orange
9/$09	brown
10/$0A	light red
11/$OB	dark gray
12/$0C	medium gray
13/$0D	light green
14/$0E	light blue
15/$0F	light gray

The standard screen editor ROM routines for printing text to the 40-column display also set the color memory location for each character displayed. The color to be used is specified in location 241/$F1, the current foreground color. Whenever the output window is cleared—as when you print the {CLR} character, CHR$(147)—all locations in this area which correspond to character positions in the window will be filled with the color specified in the lower four bits of location 241/$F1. Since the output window is reset to full screen size and cleared by the CINT routine [$C07B], part of both the reset and RUN/STOP-RESTORE sequences, either of those events will fill the first 1000 locations in this area, addresses 55296-56295/$D800-$DBE7, with the default foreground color value, 13/SOD (light green). The highest 24 addresses in this area are not affected.

Multicolor Character Mode Usage

For multicolor character mode, each screen position has a corresponding location in this area which serves two functions. Multicolor character mode is selected by setting bit 4 of the VIC register at 53270/$D016 to %1. In multicolor mode, each character is four dots wide by eight lines tall, and the dots can have one of four colors. However, simply enabling multicolor mode does not automatically specify multicolor mode for all

character positions. Multicolor mode must be selected on a position-by-position basis. For each position, bit 3 of the color memory location for that position specifies whether the position will behave like standard character mode or multicolor mode.

Setting bit 3 of a color memory location to %0 will make the position behave as if standard character mode were still active for that position, except that only bits 0-2 of the color memory location are then available to hold foreground color information. Such positions are therefore limited to only 8 choices for foreground color instead of the usual 16. Only the first 8 of the colors listed above are available, since using a color value greater than 7 would put the position into multicolor mode.

Setting bit 3 of a color location to %1 selects multicolor mode for that position. In that case, bits 0-2 of the color memory location specify the color of any %11 bit pairs in the character pattern specified for the position. Since only three bits are available, only eight different colors can be selected. The first eight of the standard colors (black-yellow) are the only ones available, but since bit 3 must be set to %1 for multicolor mode, the value you must store in color memory to get these colors is different from that used for standard character mode. You must add 8 to the normal color value. For example, to make all %11 bit pairs in a character position white, you must store the value 9 (8 + 1, the normal value for white). Commodore's *Programmer's Reference Guide* states that the color values still produce the standard colors, but this is not correct. For a multicolor mode position, storing a 9 in its color memory location makes the %11 bit pairs white, not brown.

Since the operating system does not specifically support multicolor character mode, clearing the output window when this mode is active has the same effect it does in standard character mode—color memory is filled with the value from the current foreground color location (241/SF1).

Bitmapped Mode Usage

This area of memory is not used by the standard bitmapped mode (GRAPHIC 1). That mode gets all of its color information from video matrix locations. For multicolor bitmapped mode (GRAPHIC 3), values stored in this area control the color of those dots defined by %11 bit patterns in the bitmap. For the purposes of controlling color, the 160-dot X 200-line

multicolor bitmapped screen can be thought of as a grid of 4 X 8-pixel blocks that is 40 blocks across and 25 blocks tall. Each block of dots in the grid has a corresponding color memory location. The color of dots within the block which have the same bit pattern is not independently selectable—all dots with %11 bit patterns within the same 4-dot X 8-line block take the same color, the one specified in the color memory location for that block. The color memory locations can hold any of the 16 standard color values.

When the multicolor screen area is cleared with the BASIC statements GRAPHIC 3,1 or SCNCLR 3, all color memory locations are filled with the value from location 133/$85, the multicolor source 2 location. The default value for that location is 2, but that can be changed with the COLOR statement. The GRAPHIC 4,1 or SCNCLR 4 statements should have the same effect, but they do not. The contents of color memory are not affected by either of those statements. Due to a bug in the SCNCLR routine [$6A79], color memory is filled for GRAPHIC modes 2 and 3 rather than 3 and 4. However, the fact that SCNCLR 2 unnecessarily fills color memory has no obvious effect, since the standard bitmapped mode doesn't use color memory.

Color Banks

The 128 actually has two separate IK blocks of RAM available for this address area—a new feature not available in the Commodore 64 (or in the 128's Commodore 64 mode). Normally, the 128 uses one block for character color and the other for multicolor bitmapped mode. This is why GRAPHIC 4 mode (split multicolor bitmapped and text) doesn't cause the conflicts you might otherwise expect. That is, printing on the text screen doesn't disturb colors on the multicolor bitmapped screen, and drawing on the multicolor bitmapped screen doesn't disturb the colors on the text screen, even though both appear to use the same color RAM.

The banking of color RAM is handled by the screen editor portion of the IRQ service routine [$C194j. Bits 0-1 of the 8502's on-chip I/O port at location l/$01 determine which block of color RAM is visible in this address area (see the entry for location l/$01 in Chapter 2 for details). Block 0 is normally used for the multicolor bitmapped screen and block 1 for the text screen. If you disable the screen handling portion

of the interrupt service routine, you can use this feature for other effects. For example, you can maintain two separate color displays for the text screen, or two separate text screens—each with its own color memory—or two separate sets of %11 pixel color patterns for the multicolor bitmapped screen.

Program 8-2 shows one effect you can create with multiple color memory blocks for a single text screen. Line 110 selects the BANK 15 configuration (so that the I/O block, including color RAM, will be visible) and stores the value 255 in the screen mode flag to disable the screen handling portion of the interrupt service routine. Remember that split screen modes like GRAPHIC 2 and GRAPHIC 4 will not work while this portion of the interrupt service routine is disabled. Lines 130-150 set up block 0 of color RAM, filling the screen borders with an alternating pattern of black and white positions. Lines 170-190 perform a similar setup for block 1, but using an alternating pattern of white and black. Lines 210-230 set up screen memory, drawing a border of ball characters around the screen. Lines 250-310 then switch between the two color blocks, producing a marquee effect as the border colors alternate.

Program 8-2. Color Bank Switching Demo

```
100 REM ** COLOR SWITCHING DEMONSTRATION
110 BANK 15:POKE 216,255:REM DISABLE SCREEN INTERR
   UPTS
120 REM ** SET UP BLOCK 0 OF COLOR RAM
130 POKE 1,PEEK(1) AND 254:PRINT"[CLR}"
140 FOR 1=0 TO 39 STEP 2:POKE 55296+1,0:POKE 55297
   +I,1:POKE 56256+1,0:POKE 56257+1,1:NEXT I
150 FOR 1=0 TO lliPOKE 55336+(1*80),1:POKE 55375+{
   I*S0),0:POKE 55376+(I*80),0:POKE 55415+(1*80),1
   :NEXT I
160 REM ** SET UP BLOCK 1 OF COLOR RAM
170 POKE 1,PEEK(1) OR 1:PRINT"ICLRJ"
180 FOR 1=0 TO 39 STEP 2:POKE 55296+1,1:POKE 55297
   +I,0:POKE 56256+1,1:POKE 56257+1,0:NEXT I
190 FOR 1=0 TO 11:POKE 55336+(1*80),0:POKE 55375+(
   I*80),1:POKE 55376+(I*80),l:POKE 55415+(1*80),0
   :NEXT I
200 REM ** SET UP SCREEN MEMORY
210 PRINT"{HOME}[3 DOWNJtRIGHT}tRVSJ PRESS ANY KEY
    TO HALT DEMONSTRATION^ SPACES J"
```

```
220 FOR I=0 T0 39!POKE 1024+1,81 sPOKE 1984+1,81 -.NE
   XT I
230 FOR 1=0 TO 22:POKE 1064+(1*40),81:POKE 1103+(I
   *40),81:NEXT I
240 REM ** ALTERNATE COLOR RAM BLOCKS
250 POKE 1,PEEK(1) AND 253
260 FOR 1=1 TO 65JNEXT I
270 GET A$:IF A$<>"" THEN 310
280 POKE 1,PEEK(1} OR 2
290 FOR 1=1 TO 65:NEXT I
300 GET A$:IF A$="" THEN 250
310 END
```

CIA (Complex Interface Adapter) Chip Registers

56320-56335/$DC00-$DC0F and 56576-56591/$DD00-$DD0F

The CIA (complex interface adapter) chips perform the majority of the 128's input and output functions. Between them, the CIAs are responsible for handling communications with the keyboard, joysticks, the serial bus (where disk drives and printers are connected), the RS-232 port (where modems are connected), and the user port. In fact—with the exception of video output provided by the VIC and VDC chips and the audio output provided by the SID chip—the list of I/O functions performed by devices other than the CIAs is quite short: the VIC and VDC chips handle light pen input for their respective displays, the SID chip reads paddle controllers (although a CIA reads paddle buttons and selects which pair of paddles is to be read), the processor's on-chip I/O port is used to control some aspects of tape data storage and to read the CAPS LOCK key, and an MMU register line is used to read the 40/80 DISPLAY key.

All CIA registers are set to zero when the system RESET line is pulled low, as when the reset button is pushed. Most of the CIA registers are initialized during the IOINIT routine [$E1O9]. Table 8-8 lists the functions of the various CIA registers. A detailed description of the use of each follows.

Table 8-8. CIA Chip Registers

CIA #1	CIA #2	
56320/$DC00	56576/$DD00	Port A data I/O register
56321/$DC01	56577/$DD01	Port B data I/O register
56322/$DC02	56578/$DD02	Port A data direction register
56323/$DC03	56579/$DD03	Port B data direction register
56324/$DC04	56580/$DD04	Timer A latch/counter (low byte)
56325/$DC05	56581/$DD05	Timer A latch/counter (high byte)
56326/$DC06	56582/$DD06	Timer B latch/counter (low byte)
56327/$DC07	56583/$DD07	Timer B latch/counter (high byte)
56328/$DC08	56584/$DD08	Time-of-day dock (1/10 seconds)
56329/$DC09	56585/$DD09	Time-of-day clock (seconds)
56330/$DC0A	56586/$DD0A	Time-of-day dock (minutes)
56331/$DC0B	56587/$DD0B	Time-of-day dock (hours)
56332/$DC0C	56588/$DD0C	Serial data register
56333/$DC0D	56589/$DD0D	Interrupt control register
56334/$DC0E	56590/$DD0E	Control register A
56335/$DC0F	56591/$DD0F	Control register B

CIA #1 Registers

This CIA is used to read the keyboard, joysticks, and other devices connected to the control ports, such as the 128 mouse. It also selects which pair of paddles will be read. The timers and FLAG input line are used in reading from and writing to tape. The chip's serial data communications hardware is used for fast serial bus I/O.

56320	SDC00	D1PRA
56321	$DC01	D1PRB

I/O port data registers

The CIA chip has two eight-line data ports, designated A and B. Each bit in the data registers is connected to one of the port lines (PA0-PA7 for port A and PB0-PB7 for port B). The lines can be either inputs or outputs, depending on the setting of the data direction registers (56322-56323/$DC02-$DC03). For port B, bits 6-7 (lines PB6-PB7) can also have special timer output functions. See the discussion of the control registers at 56334-56335/$DC0E-$DC0F for more information.

When a port line is set for input, its corresponding data register bit will reflect the state of the line. The bit will hold %0 when the line is pulled low (0 volts), or %1 when the line is high (+ 5 volts). An unconnected input line, or one connected to a device that isn't actively pulling the line low, will

"float" in a high state, and the corresponding data register bit *yn* hold a %1. Writing to a data register bit for an input line has no effect on the state of the line, or on the value returned when the bit is read. However, the value written to the bit will be retained internally, and will determine the state of the line if the line is changed to an output.

When a port line is set for output, its corresponding data register bit will control the state of the line. Setting the bit to %0 will pull the line to a low (0 volts) state, and setting the bit to %1 will allow the line to go to a high state (+5 volts). Note that string the bit to %1 doesn't guarantee that the output line will be set to a high state. The data CIA can force an output line low when an external device connected to the line is trying to hold the line high, but it cannot bring an output line high when an external device is holding the line low. Reading the data register bit for an output line returns the current state of the line (%0 if the line is low, or %1 if the line is high). Thus, a line set for output can still be used for a limited form of input. If the data register bit for the output line is set to %1, then reading that bit will return %1 while the output line is high and %0 when an external device pulls the line low. This explains how a joystick can be read from port A despite the fact that the lines of that port are normally configured as outputs.

Port B has a feature not available with port A. A special handshaking output line (from the CIA) designated PC, normally high, will go low for one system clock cycle each time data is read from or written to port B. This can be used to signal an external device that data has been written or accepted at the port. The PC line from CIA #1 is not connected to anything in the 128, but the PC line from CIA #2 is available at pin 8 of the user port.

For CIA #1, all port A lines are normally configured as outputs and all port B lines are normally configured as inputs. This is to set up the lines for their primary function—reading the keyboard. Refer to Figure 7-1 and the discussion of the keyboard-scanning routine [$C55D] in Chapter 7 for more details on how these lines are used for that purpose.

The second major function of these ports is to read the status of digital controllers connected to the two control ports °n the side of the 128. Each control port is connected to five lines of one of the CIA #1 ports. Port B is connected to control

port 1 {the front one) and port A to control port 2 (the back one), which is the opposite of what you might expect. The connections are as follows:

CIA port bit	Controller port pin	Joystick function
0	1	up
1	2	down
2	3	left
3	4	right
4	6	button

The control port pins are usually described in terms of their joystick functions, since the joystick is the device most commonly connected to the ports. A joystick is a very simple device consisting of five switches, one for each of the four primary directions and one for the fire button. The switches are normally open, meaning that no connection is made when the joystick is not being pressed. Remember that unconnected CIA port lines float to a high state (+ 5 volts), so the corresponding CIA port bits will be %1. Pressing the stick in one of the eight possible directions (four primary directions and four diagonals) closes one or two of the switches, grounding the CIA port lines and forcing the corresponding bit or bits to %0. Pressing the fire button grounds that line, so it is possible to have as many as three lines simultaneously grounded.

For reading the joysticks, BASIC provides the JOY function [$8203], which returns a value from 1-7, depending on the direction the stick is pressed (the value is 0 if it is not being pressed), with 128 added if the fire button is pressed. To read the joystick from machine language, you simply read the corresponding CIA port data register and check for %0 bits. Even though the port A lines are normally set up as outputs, you can still read the joystick without switching the lines to inputs. The keyboard scan routine leaves all port A bits except bit 7 set to %1, so it is still possible for external devices like the joystick to pull the lines low (bit 7 of that port will normally always be %0). The possible port readings are as follows:

	ortl SDC01)	Control port 2 CIA port A. ($DC00)		Joystick direction
$11111111	255/SFF	%oiinm	127/$7P	not pressed
fciiiiino	254/SFE	%01111110	126/S7E	up
%11111101	253/$FD	%01111101	125/$7D	down
ftllUlOll	251/SFB	%01111011	123/$7B	left
£11111010	250/$FA	%01111010	122/S7A	up + left
£11111001	249/$F9	%01111001	121/579	down + left
%iinom	247/$F7	%omom	119/$77	right
#11110110	246/$F6	%01110110	118/$76	up + right
%11110101	245/$F5	%onioioi	117/$75	up + left

If the fire button is pressed, bit 4 of the data register will also be set to %0, and the values listed above will be reduced by 16/$10.

If you'd prefer to read %1 bits instead, try the following code:

Control port 1	Control port 2
LDA $DC01	LDA $DC00
AND#$1F	AND #$1F
EOR #$1F	EOR #$1F

Using this method, the accumulator will hold one of the following values after the port is read:

Value	Joystick direction
0/$00	not pressed
l/$01	up
2/$02	down
4/$04	left
5/$05	up + left
6/$06	down + left
8/$08	right
9/$09	up + right
10/$00	down + right

Pressing the fire button will add 16/$10 to any of the values listed above.

You should be aware, however, that using control port 1 (CIA port B) for joystick input can have an undesirable side effect. Since the input lines of that port are also used for reading the keyboard, the keyscan routine [$C55D] has no way to tell whether the port lines are being grounded by keypresses or

joystick presses. As a result, moving a joystick effectively generates a keypress, and certain keypresses produce the same effect as moving the joystick:

Joystick conflicts		Keyboard conflicts	
Direction	**Effective keypress**	**Keypress**	**Effective direction**
up	**ALT**	**1**	up
down	SHIFT-INST/DEL	-	down
left	RETURN	CONTROL	left
right	cursor left	2	right
fire	SHIFT-F7	space	fire

Many programs avoid this problem by using only port A (control port 2), which is the simplest solution. Because the port A lines are outputs, the joystick is never mistaken for the keyboard in control port 2. You can prevent the keyboard conflicts by disabling interrupts and forcing all keyboard column lines high before reading the port (add SEI:LDA #$FF:5TA $DC00:STA $D02F before the LDA $DC01 instruction, and CLI after it). The joystick conflicts are more difficult to prevent—there is no way to disable the joystick. If your program doesn't require keyboard input, a rather inelegant solution is to simply zero the count of keys in the buffer and pending function key characters (locations 208/$D0 and 2O9/$D1) before exiting from the program.

Any device which behaves like a joystick can be read in the same manner. This includes trackballs and the new mouse controllers (which are more or less upside-down trackballs). Devices such as paddles or graphics tablets are analog, not digital, devices, and are read by the SID chip. (See the entry for the SID registers at 54297-54298/$D419-$D41A.) However, any buttons on these devices are read as if they were joystick lines. For example, the two buttons on a standard pair of Commodore paddles are read exactly like the joystick left- and right-direction lines for the corresponding control ports. Since the CIA ports are bidirectional, you could also use the control port lines in interfacing projects. Together they provide ten lines which can be either inputs or outputs.

The final function of these CIA ports is that bits 6-7 (lines PA6-PA7) of port A control which control port will be connected to the SID lines for reading paddles. Since paddles come in pairs and the 128 has two control ports, you can connect up to four paddles. However, the SID has only two paddle inputs. As a result, you can read paddles from only one port at a time. Valid selections are as follows:

Bits	
7	6
0	1
1	0

Control port selected

The other combinations result in either both or neither ports being selected. The port A lines are always outputs, so there is no problem with using them for this purpose. The default value in port A (except during the IRQ, when it is being used to scan the keyboard) is 127/$7F. This has the bits set to %01, selecting control port 1. Unless you really need four paddles, it's best to use this port. To write any value other than 127/S7F into port A, you must disable interrupts, since the keyboard-scanning routine always leaves the port set to that value.

Another bit of 128 hardware trivia: Our experience indicates that a CIA #1 failure is one of the most common hardware problems the 128 owner is likely to experience. Integrated circuit chips like the CIA are very sensitive to electric discharges such as the static electric spark you see when you touch a doorknob after shuffling across a carpet. Since the control port lines lead directly to the pins of CIA #1, touching a control port pin when your body carries a static charge is like a lightning strike to the chip. Unfortunately, since the control ports lie so close to the reset and power switches, it's very easy to touch them unintentionally, especially control port 2. Of the five 128s we have had at COMPUTE! Publications to date, three have experienced "blown" CIAs. Bit 1 of port A seems to be particularly susceptible. This is in keeping with our experience with Commodore 64s over the past several years, where we have lost approximately a dozen CIA chips out of about 30 computers. If one joystick direction or a group of keys suddenly becomes impossible to read, a blown CIA is the likely source. To prevent this, some users resort to covering the control ports with masking or electrical tape when the port is not in use. Another solution is to leave the joystick plugged in at all times.

56322	8DC02	D1DDRA
56323	SDC03	D1DDRB

Data direction registers

Each of the eight lines in the CIA's two data ports can be individually configured as either an input or an output. These data

direction registers (DDRs) specify the direction of data flow on the port lines. Each register bit corresponds to one port line (PA0-PA7 for port A and PB0-PB7 for port B). Setting a register bit to %0 makes the corresponding port line an input while setting the bit to %1 makes the line an output. The lines are read and controlled by the data registers at 56320-56321/$DCOO-$DC01. For lines PB6-PB7 of port B, the settings of bits 6-7 of 56323/$DC03 can be superseded when those lines are used for their special timer output functions. See the discussion of the control registers at 56334-56335/$DC0E-$DC0F for more information.

For CIA #1, the IOINIT routine [$E109]—part of both the reset and RUN/STOP-RESTORE sequences—initializes the port A DDR (56322/$DC02) to 255/$FF, making all port A lines outputs, and the port B DDR (56323/$DC03) to 0/$00, making all port B lines inputs. These settings are not changed by any other ROM routines. You can change these settings briefly for special I/O functions involving the control ports, but leaving any port A lines set as inputs or port B lines set as outputs will disable normal keyboard functioning. See the discussion of the data registers above for more information on the uses of the port lines.

56324	$DC04	D1T1L
56325	$DC05	D1T1H

Timer A latch/counter registers

Timer A is a programmable counter that can provide a variety of timing functions. It is a countdown timer, meaning that it repeatedly decrements the counter contents until the value is decremented below zero, a condition known as underflow. When Timer A underflow causes an internal CIA interrupt, bit 0 of the CIA interrupt register at 56333/$DC0D will be set to %1. The timer can be set to count down repeatedly or just once. The timer countdown can be driven by either of two sources: the system clock frequency or a signal provided on the CNT line by an external device. The operating conditions of the timer are specified in the CIA control register A (56334/$DC0E).

When read, the registers here return the current value in the 16-bit counter (low byte in 56324/SDC04 and high byte in 56325/$DC05), Data written to these registers does not go directly into the counter unless the timer is currently stopped.

Instead, the values are held in an internal latch register. The latch contents are transferred into the counter whenever the timer underflows. Alternatively, a bit in the control register can force an immediate transfer of the latched value into the counter.

One special function of timer A is to control the rate of data output over the CIA's serial port (SP) line. In this case, timer A generates a clock signal that is provided as an output on the CNT line. Since timer A must underflow twice to produce the full clock cycle on the CNT line required for each bit, you should load timer A with a value which will produce an underflow in half the time desired for each bit. In other words, the duration of each bit transmitted on the serial line will be twice the time required to count down the value specified in the timer A latch. See the entry for the CIA serial data register (56332/$DC0C) for more information on the CIA's serial data I/O capabilities.

For CIA #1, timer A can be a source of IRQ interrupts to the processor. This feature is used during tape I/O to generate the IRQ interrupts which drive the reading and writing of data. CIA #1 timer A also controls the transmission rate for data sent over the fast serial bus. Thus, if you use the timer for your own timing applications, you should be aware that both tape and fast serial operations will change the settings of the timer. (The timer will be stopped upon completion of any tape or fast serial operation.) The IOINIT routine [$E109] calls the Kernal fast serial output setup routine, which starts the timer counting down from $0004, but then immediately calls the fast serial input setup routine, which halts timer A. Thus, upon completion of the reset or RUN/STOP-RESTORE sequence, the timer will contain some very low value: usually either $0001 or $0002. Unlike the Commodore 64, CIA #1 timer A is not the source of the 128's normal jiffy IRQ interrupts. That task is instead performed by raster interrupts from the VIC chip.

56326	$DC06	D1T2L
56327	$DC07	D1T2H

Timer B latch/counter registers

The operation of timer B is quite similar to that described above for timer A, but is a bit more flexible. In addition to counting system clock and CNT pulses, timer B can also count

timer A underflows. This effectively ties the two timers together to form a 32-bit countdown value, allowing countdown intervals of up to 70 minutes. The operation of timer B is controlled by the register at 56335/$DC0F.

For CIA #1, the IOINIT routine [$E109], part of both the reset and RUN/STOP-RESTORE sequences, loads the latch for this timer with 65535/$FFFF and starts the timer running in continuous mode. There is no obvious reason for this step. The only use of timer B by 128 ROM routines is during tape I/O, where it is used to generate IRQ interrupts to drive the reading and writing of data. Upon completion of a tape operation, the timer will be left halted and set for one-shot mode.

56328	SDC08	D1TOD1
56329	$DC09	D1TODS
56330	$DC0A	D1TODM
56331	$DC0B	D1TODH

Time-of-day clock registers

The time-of-day (TOD) clock is a special feature of the CIA. It keeps time in hours, minutes, seconds, and tenths of seconds—units more useful to humans than the jiffies of the system software clock or the fractional microseconds of timer A or B. There are actually two sets of registers at these locations, the time and the alarm. By storing an alarm value here, you can trigger an internal interrupt (and, optionally, an external IRQ request) when a specified time is reached. When you read the registers, you always see the time value (the alarm setting is never visible). When you're writing to the register, bit 7 of control register B (56335/SDCOF) determines whether the value being written will set the time or the alarm.

The time is kept in 12-hour format, with bit 7 of the hours register used as a flag to indicate AM or PM. The time data in the registers is in binary coded decimal (BCD) format. In this format, each half-byte (nybble) contains a value which represents one decimal digit. For example, if the minutes register contains the value %00100110, equivalent to 38/$26, the minutes digits should be interpreted as 2 and 6—26 minutes past the hour, rather than 38 minutes. The register bits should be interpreted as follows:

56328/$DC08: Bits 0-3 hold the 1/10-seconds digit. Bits 4-7 are unused. Writing to those bits has no effect, and they always return %0 when read.

56329/$DC09: Bits 0-3 hold the ones digit for seconds. Bits 4_6 of this register hold the tens digit (only three bits are needed, since this digit will never be greater than %101 = 5__the seconds count rolls over to $00 after reaching $59.) Bit 7 is unused. Writing to that bit has no effect, and it always returns a %0 when read.

56330/$DC0A: Bits 0-3 hold the ones digit for minutes. Bits 4-6 hold the tens digit. Bit 7 is unused. Writing to that bit has no effect, and it always returns a %0 when read.

56331/$DC0B: Bits 0-3 of this register hold the ones digit for hours. Bit 4 holds the tens digit for hours (only one bit is needed, since this digit will always be either 0 or 1). Bits 5-6 are unused. Writing to these bits has no effect, and they always return %0 when read. Bit 7 is the AM/PM flag. The 12-hour format time value is taken to represent AM (midnight-noon) when this bit is %0 and PM (noon-midnight) when the bit is %1. Be sure to remember this bit when setting an alarm time.

The order in which you read and write these registers is important. When reading the clock, the registers latch (remain constant) after you read the hours register until you read from the 1/10-seconds register. The clock continues to count internally; only the register values remain constant. Thus, each read of the hours register *must* be followed by a read of the 1/10-seconds register, even if you don't care about the 1/10-second value. Likewise, the clock stops whenever a value is written to the hours register, and does not start again until a value is written to the 1/10-seconds register. Thus, for both reading and writing you should start with the hours register and end with the 1/10-seconds register.

The following routine illustrates the use of the time-of-day clock by displaying the current time in the upper left corner of the screen:

```
D00  LDA  #$89    ;Load clock registers with desired
D02  STA  $DC0B   ; initial time. (This example uses
D05  LDA  #$05    ; 9:05:00.0 PM)
D07  STA  $DC0A
D0A  LDA  #$00
DOC  STA  $DC09
DflF LDA  #$00
```

```
D11  STA  $DC08  ;Gock starts when this register is written
D14  LDA  $DC0B  ;Read hours byte (this latches the time value)
D17  BMI  $OD1D  ;Add either an A or a P to the time string,
D19  LDX  #$41   ; depending on the state of the AM/PM bit
D1B  BNE  $0D1F
D1D  LDX  #$50
D1F  STX  $0D70
D22  AND  #$7F   ,Mask off the AM/PM bit
D24  JSR  $0D54  ;Convert the hours byte to two characters
D27  CMP  #$30   ;RepIace leading zero with a space
D29  BNE  $0D2D
D2B  LDA  #$20
D2D  STA  $0D67  ;Add hours-digit characters to string
D30  STX  S0D6S
D33  LDA  $DC0A  ;Read minutes byte
D36  JSR  $0D54  ;Convert to two characters
D39  STA  $0D6A  ;Add minutes-digit characters to string
D3C  STX  $0D6B
D3F  LDA  $DC09  ;Read seconds byte
D42  JSR  $0D54  ;Convert to two characters
D45  STA  $0D6D  ;Add seconds-digit characters to string
D48  STX  $0D6E
D4B  LDA  $DCO8  ;Read 1/10-seconds byte (to unlatch time)
D4E  JSR  $0D62  ;Print time string
D51  JMP  $0D14  ;Repeat indefinitely

                 ;Convert BCD byte to two ASCII characters
TJS4 PHA         ;Stash the byte
D55  AND  #$0F   ;Mask off all but the lower four bits
D57  ORA  #$30   ;Add base ASCII numeral value
D59  TAX         ;Leave low digit in X register
D5A  PLA         ;Retrieve original value
D5B  LSR         ;Move high four bits into low nybble
D5C  LSR
D5D  LSR
D5E  LSR
D5F  ORA  #$30   ;Add base ASCII numeral value
D61  RTS
D62  JSR  $FF7D  ;Print time string using Kernal PRIMM
>D65 13 12 30 30 3A 30 30 3A ;String characters:
>D6D 30 30 20 41 4D 20 00    ; {HOME}{RVS}00:00:00 AM
D74  RTS
```

Neither CIA's time-of-day clock is used by 128 mode, although CP/M does make use of the CIA #1 dock for time-keeping. None of these registers is initialized by any 128 ROM

routine. Thus, both are free for your own programming. All clock registers are reset to %0 when the system is reset.

56332 8DC0C **D1SDR**

Serial data register

The CIA supports serial (bit-by-bit) data transfers in hardware. This register holds the byte of data to be sent over the CIA's SP (serial port) line, or the byte read from the line. At any given time, the CIA serial line must be configured for either input or output. This is controlled by the setting of bit 6 of control register A (56334/$DC0E).

When the line is set for input, the state of the SP line is read as a data bit each time there is a low-to-high (0 to +5 volts) transition on the CNT input line. (CNT must be driven by the external device which is sending data.) As each bit is read, it is transferred into an internal serial shift register. When an entire byte has been read (after eight pulses of the CNT line), the value in the internal shift register is transferred to this register and an interrupt will be indicated in bit 3 of the CIA interrupt register (56333/$DC0D). At this point, the received byte can be read from the register.

When the line is set to be an output, data written to this register will be transferred into the internal shift register and then sent out a bit at a time over the SP line. Timer A determines the rate at which bits will be sent. The transmission will begin immediately if timer A is running; otherwise, it begins when the timer is started. Bits will be written on the serial line at one-half the countdown rate of timer A. That is, timer A must underflow twice for each bit sent. This clock signal appears as output on the CNT line. After all eight bits are sent, an interrupt will be indicated in bit 3 of the CIA interrupt register (56333/$DC0D) to indicate that another byte can be sent.

This hardware serial data communications feature went unused in the Commodore 64, but in the 128 it is used to support the fast serial bus. The only fast serial peripheral device currently in widespread distribution is the 1571 disk drive, but others may appear in the future. For CIA #1, the SP line is connected to the serial bus DATA line (with additional circuitry to prevent conflicts with slow serial communications) and the CNT line is connected to the SRQIN line. The CIA #1 SP and CNT lines are also available from the user port, at pins 5 and 4, respectively.

56333 SDCOD D1ICR

Interrupt control register

The CIA chip has five internal interrupt sources: timer A underflow, timer B underflow, time-of-day clock alarm, serial data buffer full or empty, and FLAG signal. The CIA can also generate an interrupt request output signal as a result of any of these conditions. The location actually has two different functions, depending on whether it is being read from or written to. When you read from this register, you see the contents of an internal interrupt data register that indicates which interrupts, if any, have occurred. When you write to this register, the value goes to an internal interrupt mask register that specifies which interrupts—if any—are to result in an external interrupt request being generated. The data register is read-only {it can't be written to), and the mask register is write-only (it can't be read from).

When you're reading from the register, bits 0-4 indicate which interrupts have occurred since the last time those bits were read. The bit is set to %1 when the corresponding interrupt occurs, regardless of whether or not the source is set to trigger an external interrupt request output. All these bits are automatically cleared to %0 after the register has been read. The interrupt type indicated by the individual bits is as follows:

Bit	Interrupt source
0	Timer A underflow
1	Timer B underflow
2	Time-of-day clock alarm
3	Serial data buffer full or empty
4	High-to-low transition on FLAG input line

A timer underflow occurs when the timer counts down below zero. A time-of-day clock alarm occurs when the time in the clock registers matches the value in the alarm registers. The serial-buffer-empty condition occurs during output after all eight bits for a byte have been written on the serial port (SP) line, and a buffer-full condition occurs during input after eight bits have been read from the SP line. The FLAG line is a special input provided on the CIA specifically for the purpose of generating interrupts. A FLAG interrupt occurs whenever the device connected to the FLAG line causes a high-to-low voltage transition (+ 5 to 0 volts) on the line.

Bits 5-6 are unused. Writing to them has no effect, and they always return %0 when read.

Bit 7 controls the interrupt mask register function. When vou're writing to this register, bit 7 determines which mask bits will be set or cleared. If bit 7 is set to %1 in a value written to this register and any of bits 0-4 in the value are also set to %1/ *then* the corresponding interrupt mask bits will be set and the specified interrupt or interrupts will generate an external interrupt request output. The mask bits correspond to the sources listed above the data register. For example, to enable timer B as an interrupt source you would write a value to the register which has bits 1 and 7 set to % 1 — LDA #$82:STA $DC0D. (Bits in the value which are %0 are not significant.) If bit 7 in the value written to the register is set to %0 and any of bits 0-4 in the value are %1, then the corresponding mask bits will be cleared and the specified interrupt or interrupts will be disabled. For example, you could use LDA #$0F:STA $DC0D to clear all except FLAG interrupts. (Again, bits in the value which are %0 have no effect.)

When external interrupt requests are enabled, you can read bit 7 to determine whether any interrupts have occurred. When read, bit 7 will be %0 if no CIA source has generated an interrupt request, or %1 if an interrupt request output has been generated as the result of one or more enabled internal CIA interrupt conditions. Remember, however, that all the data bits in this register are cleared to %0 after the register is read. Thus, you must preserve the read register value if you wish to determine which source produced the interrupt request. For example, you shouldn't test bit 7 with the machine language BIT instruction, since that will result in the loss of the data register bit settings.

For CIA #1, the interrupt request output is connected to the processor's IRQ input line, so interrupt requests from CIA #1 result in IRQ interrupts to the processor. The IOINIT routine [SE109] clears all interrupt mask bits, so initially no interrupts are enabled. Some operations enable interrupts for the duration of their activity. Tape I/O, which is interrupt-driven, uses timer A, timer B, and FLAG interrupts as IRQ sources. Fast serial bus I/O uses internal serial data buffer interrupts, but does not trigger external IRQ requests. See the tape and fast serial routines in Chapter 9 for details. You are, of course, 'ree to set up your own CIA #1 interrupts, but you must write your own interrupt service routine. The standard IRQ handler [5>FA65] merely reads this register to clear it, and ignores the

value read. See Appendix A for more information on interrupts. In Commodore 64 mode (and in the original Commodore 64), timer A interrupts from CIA #1 were the source for the system's 1/60-second jiffy IRQ. However, 128 mode instead uses raster interrupts from the VIC chip for that function.

56334 $DCOE D1CRA

Control register **A**

This register controls the operation of timer A, except for bits 6-7, which control serial data port and time-of-day clock functions, respectively. See the entry at 56324-56325/$DC04-$DC05 for more information on timer A.

Bit 0: This bit acts as the ignition switch for timer A. Writing a %0 here stops the timer. If the timer is currently stopped, writing a %1 here causes the counter to begin decrementing its current contents (unless the force load strobe, bit 4 of this register, is also set to %1—in that case, the latch value is loaded into the counter before counting resumes). Writing a %1 here when the bit is already set to %1 has no effect. Setting the bit to %0 when it has previously been %1 halts the timer, leaving the counter holding the value it has reached when stopped. When the timer is set for one-shot mode, this bit will automatically be reset to %0 when the count underflows. In this case, the counter will be reloaded with the latch value.

For CIA #1, the IOINIT routine sets this bit to %0, so timer A is initially stopped. The ROM routines for tape I/O use this timer to generate IRQ interrupts for reading and writing tape bits, but the bit is reset to %0 upon completion of any tape operation. The Kernal SPOUT routine [$E5D6], which prepares the fast serial bus for output, starts the timer and leaves it counting. (Timer A determines the rate at which data is sent over the fast serial lines.) The Kernal fast serial output routines end with calls to the SPIN routine [$E5C3], which resets this bit to %0, halting the timer.

Bit 1: This bit controls whether or not timer A generates output on the PB6 line of port B. Writing a %1 here enables PB6 output. When PB6 is selected for timer A output, the line automatically becomes an output, regardless of the setting of the data direction register bit for that line. The output on PB6 will be either short pulses or a regular toggling of the line be-

tvveen high and low states. The output mode is controlled by bit 2 of this register. Writing a %0 here disables timer A output and restores PB6 to the state and function defined in the port B data and data direction registers.

For CIA #1, this bit is set to %0 during the IOINIT routine [$E109], and is not changed by any 128 ROM routine. The PB6 output feature is not useful on CIA #1, since the PB6 line for that CIA is connected only to a keyboard row scanning line, and is not available externally. However, the PB6 line from CIA #2 is available at pin K of the user port, so this feature could be used with timer A of that CIA.

Bit 2: When bit 1 of this register is set to allow timer A to generate output on the PB6 line of port A, this bit controls the type of output. (This bit has no effect when bit 1 is %0.) The two selections for output type are pulse and toggle. For pulse output, selected when this bit is set to %0, the output line is held low except for a very brief high pulse each time timer A overflows. The line will be held high for one cycle of the O2 clock rate. For the 128, that is equal to 0.978 microseconds in NTSC (North American) systems, or 1.01 microseconds on PAL (European) systems. For toggle output, selected when this bit is set to %1, the PB6 line switches state—low-to-high or high-to-low—each time the timer underflows, starting from a high state.

For CIA #1, this bit is set to %0 during the IOINIT routine [$E109], and is not changed by any 128 ROM routine.

Bit 3: This bit controls whether timer A runs in continuous or one-shot mode. In continuous mode, selected when this bit is set to %0, the timer will perform repeated countdowns. After each underflow, the counter is reloaded with the latch value and restarted. In one-shot mode, selected when this bit is set t° %1, the timer counts down to underflow only once, at which time bit 0 of this register is reset to %0 to halt further counting. However, the latch value is still transferred to the counter when the underflow occurs.

For CIA #1, the IOINIT routine [$E109] initializes this bit to %1, Tape I/O routines use this timer, but also in one-shot wode, so the bit should still be set to %1 after any tape operation is performed. The Kernal fast serial output routines will change this bit to %0 to run the timer in continuous mode. However, upon completion of any fast serial output operation,

the port is reset for fast serial input, which includes resetting this bit to %1 for one-shot mode.

Bit 4: Writing a %1 to this bit, called the force load strobe, causes the contents of the timer A latch to be transferred to the counter, regardless of whether the timer is currently running or stopped. Using the strobe bit while the timer is running allows you to modify the counter contents in the middle of a countdown. Writing a %0 to this bit has no effect. This bit is write-only; it always returns a %0 when read.

Bit 5: This bit controls which of two possible events will drive timer A. Two different input signals can be used to make the timer decrement. The timer will be decremented once for each cycle of the specified event, but only if bit 0 of this register is set to %1 to allow counting. When this bit is set to %0, the timer will be driven by the system O2 clock, which provides a "tick" every 0.978 microseconds in NTSC (North American) systems, or 1.01 microseconds on PAL (European) systems. The maximum delay between underflows with this clock rate is in the neighborhood of 1/15 second. Setting the bit to %1 makes the CIA's CNT line the clock source, so that an external source can drive the count rate. In this case, the counter will be decremented once each time the external device connected to CNT provides a low-to-high transition on the line. For CIA #1, the CNT line is available at pin 4 of the user port. The CNT line is also used as the clock source for the fast serial bus, and is connected to the SRQIN line of the serial port. However, the line from SRQIN will be an input only when the FSDIR bit (bit 3) of the MMU mode configuration register at 54533/$D505 is set to %0.

For CIA #1, this bit is initialized to %0 during the IOINIT routine [$E109], and that setting is not changed by any other 128 ROM routine. Timer A must be set to count system clock pulses in order for tape operations to perform properly.

Bit 6: This bit does not control a timer A function, but instead specifies the direction of data flow on the CIA's serial port (SP) line. When this bit is set to %0, the SP line is an input. Data read on the line will be collected in the serial shift register until a full byte has been received; then an interrupt will be generated. When this bit is set to %1, the SP line is an output, and data written to the serial data register at 56332/$DC0C will be sent out on the SP line at a rate depending on

timer A. See the discussion of the serial data register for more information.

For CIA #1, this bit is initialized during the IOINIT routine to %0. The 128 uses the CIA #1 serial data register for fast serial bus communications, so the bit must be set to %1 whenever fast serial output is being performed. However, after the output has been completed the Kernal fast serial output routines reset the bus for input, which includes resetting this bit to %0.

Bit 7: This bit controls a time-of-day clock function rather than a timer A function. It determines the rate at which the time-of-day clock will be incremented. The clock is driven by the CIA's TOD input pin, which, in the 128, is connected to circuitry which produces a clock signal from the AC power supply. This signal will have the same frequency as the local power supply, generally 60 hertz (60 cycles per second) in North America and 50 hertz in Europe. Setting this bit to %0 specifies that the 1/10-seconds digit of the clock time should be incremented once for every 6 cycles of the TOD signal (for a 60-hertz source), while setting it to %1 specifies that the digit should be incremented on every fifth cycle (for a 50-hertz source). Specifying an incorrect rate will make the time-of-day clocks in your system count either too fast or too slow.

For CIA #1, this bit is initialized to %0 during the IOINIT routine [$E109], and that setting is not changed by any other 128 ROM routine. This is the proper setting for North America, but unless there is a different version of the ROM in European 128s, overseas users will need to change this bit to get proper timekeeping.

56335 $DC0F D1CRB

Control register B

This register controls the operation of timer B, except for bit *7,* which controls a time-of-day clock function. See the entry at 56324-56325/$DC04-$DC05 for more information on timer B.

Bit 0: This bit acts as the ignition switch for timer B. While the bit is %0, the timer is stopped. If the timer is currently stopped, writing a %1 here starts the counter. The countdown will resume with the current counter contents unless bit 4 of this register is also set to %1 to force the latch value to be reloaded. Writing a %1 here when the bit is already set to %1

has no effect. Setting the bit to %0 when it has previously been %1 halts the timer, leaving the counter holding whatever count value it has reached when stopped. When the timer is set for one-shot mode, this bit is automatically reset to %0 when the count underflows. In this case, the counter is reloaded with the latch value.

For CIA #1, the IOINIT routine sets this bit to %1, so timer B is normally running (although there is no obvious reason for this). The timer is used in one-shot mode for tape I/O, so this bit will be set to %0 (and the timer will be stopped) upon completion of any tape operation.

Bit 1: This bit controls whether or not timer B generates output on the PB7 line of port B. Writing a %1 here enables PB7 output. When PB7 is selected for timer B output, the line automatically becomes an output, regardless of the setting of the data direction register bit for that line. The output on PB7 will be either short pulses or a regular toggling of the line between high and low states. The output mode is controlled by bit 2 of this register. Writing a %0 here disables timer B output and restores PB7 to the state and function defined in the port B data and data direction registers.

For CIA #1, this bit is set to %0 during the IOINIT routine [$E109], and is not changed by any 128 ROM routine. The PB7 output feature is not useful on CIA #1, since the PB7 line for that CIA is connected only to a keyboard row scanning line, and is not available externally. However, the PB7 line from CIA #2 is available at pin L of the user port, so this feature could be used with timer B of that CIA.

Bit 2: When bit 1 of this register is set to allow timer B to generate output on the PB7 line of port B, this bit controls the type of output. (This bit has no effect when bit 1 is %0.) The two selections for output type are pulse and toggle. For pulse output, selected when this bit is set to %0, the output line is held low except for a very brief high pulse each time timer B overflows. The line will be held high for one cycle of the O2 clock rate. For the 128, that is equal to 0.978 microseconds in NTSC (North American) systems, or 1.01 microseconds on PAL (European) systems. For toggle output, selected when this bit is set to %1, the PB7 line switches state—low-to-high or high-to-low—each time the timer underflows, starting from a high state.

For CIA #1, this bit is set to %0 during the IOINIT routine [$E109], and is not changed by any 128 ROM routine.

Bit 3: This bit controls whether timer B runs in continuous or one-shot mode. In continuous mode, selected when this bit is set to %0, the timer will perform repeated countdowns. After each underflow, the counter is reloaded with the latch value and restarted. In one-shot mode, selected when this bit is set to %1/ the timer counts down to underflow only once, at which time bit 0 of this register is reset to %0 to halt further counting. However, the latch value is still transferred to the counter when the underflow occurs.

For CIA #1, the IOINIT routine [$E109] initializes this bit to %0, so timer B starts running continuously. However, the tape I/O routines use this timer in one-shot mode, and this bit will be left set to %1 after any tape operation has been performed.

Bit 4: Writing a %1 to this bit, called the force load strobe, causes the contents of the timer B latch to be transferred to the counter, regardless of whether the timer is currently running or stopped. Using the strobe bit while the timer is running allows you to modify the counter contents in the middle of a countdown. Writing a %0 to this bit has no effect. This bit is write-only; it always returns a %0 when read.

Bits 5-6: These bits control which of four possible events will drive timer B. That is, four different input signals can be used to make the timer decrement. The timer will be decremented once for each specified event, but only if bit 0 of this register is set to %1 to allow counting. The four possible selections are as follows:

Bits	
6 5	Timer B driving source
0 0	System <&2 clock
0 1	CNT line transitions
1 0	Timer A underflows
1 1	Timer A underflows while the CNT line is high

The default source, the system O2 clock, provides a "tick" every 0.978 microseconds in NTSC (North American) systems, °r 1.01 microseconds on PAL (European) systems. The maximum delay between underflows with this clock rate is in the neighborhood of 1/15 second. The CNT option (%01) allows an external source to drive the count rate. In this case, the

counter will be decremented once each time some external device causes a low-to-high transition on the CIA's CNT line. For CIA #1, the CNT line is available at pin 4 of the user port. The CNT line is also used as the clock source for the fast serial bus, and is connected to the SRQIN line of the serial port. However, the line to SRQIN will be an input only when the FSDIR bit (bit 3) of the MMU mode configuration register at 54533/$D505 is set to %0. The option to count timer A underflows is convenient for creating longer delays. By setting timer A to count system O2 clock pulses and timer B to count timer A underflows, you can achieve a countdown interval of up to 70 minutes when both timers start with maximum counts.

For CIA #1, these bits are initialized by the IOINIT routine to %00 to have the timer count system clock pulses. This setting is not changed by any other Kerna] routine. Tape operations will function properly only when CIA #1 timer B is counting system clock pulses.

Bit 7: Unlike the other bits of this register, this one does not control a function of timer B. Instead, it specifies whether values written to the time-of-day clock registers at 56328-56331/ $DC08-$DC0B will be directed to the clock time latch or to the alarm latch (see the discussion of the time-of-day clock registers for more information on the alarm function). While the bit is %0, values written to the registers affect the clock time latch. Setting this bit to %1 allows you to set the alarm time. This bit affects only writing to the time-of-day clock registers. When read, the registers always return the clock time, never the alarm time.

56336-56575 $DC10-$DCFF

CIA #1 register images

Due to incomplete address decoding, images of the CIA chip registers appear repeatedly every 16 bytes throughout the remainder of this page of memory. That is, storing a value in any location in this range with an address that is an exact multiple of 16 greater than one of the base register addresses has the same effect as storing the same value in one of the base register locations. For example, storing a value in 56336/$DC10 or 56560/$DCF0 has the same effect as storing a value in 56320/SDC00. However, it's better programming practice to use the officially designated register addresses.

CiA #2 Registers

This CIA is used to support the serial bus, and to provide the RS-232 interface. It also provides programmable I/O lines at the user port for custom interfacing projects. Another vital function of this unit is to select which area of memory is used as the current VIC video bank.

56576	SDDOO	**D2PRA**
56577	**$DD01**	**D2PRB**

I/O port data registers

These registers are used to read data from port lines which are configured as inputs and to write data to port lines configured as outputs. Refer to the entry for the CIA #1 ports at 56320-56321/$DC00-$DC01 for details of how these registers and ports operate. For CIA #2, the ports are used as follows:

Port A (56576/$DD00)

Bits 0-1 of this port (lines FA0-PA1) are normally configured as outputs and are used to specify which 16K area of the 64K RAM block will be used for the current VIC video bank. The four possible selections are as follows:

Bits 1	0	Video bank	System address range
0	0	3	49152-65535/$C000-$FFFF
0	1	2	32768-49252/$8000-$BFFF
1	0	1	16384-32767/$4000-$7FFF
1	1	0	0-16383/$0000-$3FFF

These bits are initialized to %11 by the IOINIT routine [$E109] during the reset and RUN/STOP-RESTORE sequences. This selects video bank 0, the default bank, and the 128 itself never selects another bank. The 64K block from which the video bank is seen is determined by bit 6 of the MMU register at 54534/$D506.

Bit 2 (port line PA2) is connected to pin M of the user port. The line is normally configured as an output, and this bit is initialized by the IOINIT routine to %1, allowing the line to go high (+5 volts). The Kernal RS-232 routines use this line as the transmitted data (TXD) output. You can also use the user port line for your own I/O functions.

The remaining bits are used for the serial bus, the 128's avenue of communications with disk drives and printers. The lines are connected as follows:

Bit	Port line	Function
3	PA3	Handles output on the serial bus ATN line.
4	PA4	Handles output on the serial bus CLK line.
5	PAS	Handles output on the serial bus DATA line {for slow serial mode).
6	PA6	Handles input from the serial bus CLK line.
	PA7	Handles input from the serial bus DATA line {for slow serial mode).

The bits for the output lines are all initialized to %0 by the IOINIT routine, causing the output lines to be pulled low {0 volts). However, all these lines have inverters between the CIA and the serial port connector, so pulling the port lines low allows the lines at the serial port connector to go high. Refer to the discussion of Kernal serial input and output routines in Chapter 9 for more information on how these lines are used for serial bus communications.

Port B (56577/$DD01)

All the lines from port B are tied directly to the user port, a 24-pin connector located on the back of the 128. The lines are connected as follows:

Bit	Port B line	User port pin
	PBO	C
1	PB1	D
2	PB2	E
3	PB3	F
4	PB4	H
5	PBS	J
6	PB6	K
7	PB7	L

All port lines are initialized as inputs, but this can be changed by changing the value in the port's data direction register at 56579/$DD03. The handshake line (PC) for port B is also available at pin 8 of the user port, These lines provide a full eight-bit parallel I/O port for your own interfacing projects. However, the port B lines also have another use. The 128 lacks a true RS-232 hardware interface, so the Kernal RS-232 routines program the user port lines to support RS-232 communications (refer to Chapter 9 for details). For RS-232, the port lines are used as follows:

Bit	Line	Pin	RS-232 function
	FLAG	B	This interrupt input is tied to the received data line. The high-to-low transition on the line at the beginning of the start bit for an incoming byte will cause a FLAG interrupt to initiate the reception of the byte.
0	FBO	C	RXD (received data). This input line is used to read incoming data bits from the modem or other external device.
\	PB1	D	RTS (request to send). This output line is used to signal the modem that the 128 is ready to send a byte.
1	PB2	E	DTR (data terminal ready). This output line is used to signal the modem that the 128 RS-232 interface is active.
3	PB3	F	Rl (ring indicator). This input line is not supported by the Kernal RS-232 routines, but is intended to allow the modem to signal the 128 that a ringing signal has been detected on the phone line. The line, normally high, goes low when a ring is detected. (Commodore modems support this line.)
4	PB4	H	DCD (carrier detected). This input line is used to allow the modem to signal the 128 that it has detected another modem on the other end of the telecommunications link. The line, normally high, goes low when the incoming carrier signal is detected.
5	PBS	J	This line is not formally assigned, but Commodore modems use it to control whether or the not the modem is connected to the phone line. Since the line is configured by the IOINIT routine as an input, you must change the corresponding data direction register bit to %1 to use the line as an output for this control function. Writing a %0 to this bit will connect the modem to the phone line (the off-hook state), while writing a %1 will disconnect (hang up) the modem.
6	PB6	K	CT5 (clear to send). This input line is used to allow the modem to signal the 128 that it is ready to accept another byte.
7	PB7	L	DSR (data set ready). This input line is used to allow the modem to signal the 128 that it is active.

(The following port A line is also used:)

	PA2	M	TXD (transmitted data). This output line is used to send data bits to the modem or external device.

56578 $DD02 D2DDRA
56579 $DD03 D2DDRB
Data direction registers

These registers specify whether the lines of ports A and B will be inputs or outputs. Refer to the discussion of the CIA #1 data direction registers at 56322-56323/$DC02-$DC03 for details of how these registers operate. For CIA #2, the IOINIT routine [$E1O9]—part of both the reset and RUN/STOP-RESTORE sequences—initializes the port A DDR (56578/$DD02) to 63/S3F, making port A lines PA0-PA5 outputs and lines PA6-PA7 inputs, and the port B DDR (56579/$DD03) to 0/$00, making all port B lines inputs. The port A setting is not changed by any other ROM routine. Bits 0-1 of the port A register should always remain set to %1 to keep PA0-PA1 outputs for the VIC video bank selection function. You can change the setting of the port B register freely to achieve the desired user port I/O configuration. The routine to set up CIA #2 for R5-232 communications [$F0B0] will set the port B DDR to 6/$06, which changes lines PB1-PB2 to outputs and all the others to inputs. See the discussion of the data registers at 56576-565 77/$DD00-$DD01 for more information on the uses of the port lines.

56580 SDD04 D2T1L
56581 $DD05 D2T1H
Timer A latch/counter registers

Refer to the discussion of CIA #1 timer A at 56324-56325/$DC04-$DC05 for details of how these registers operate. For CIA #2, the latch value for the timer is not specifically initialized during the reset or RUN/STOP-RESTORE sequences, although resetting the system will automatically set the latch count to 65535/$FFFF. The only use for this timer in system ROM is during the routines which transmit bits over the RS-232 interface, where it is used to determine the duration of outgoing bits. After RS-232 transmission is completed, the timer will continue running with a latch count value dependent on the baud rate used in the RS-232 communications.

£fi582 $DD06 D2T2L
56583 $DD07 D2T2H
Timer B latch/counter registers

Refer to the discussion of CIA #1 timer B at 56326-56327/ $DC06-$DC07 for details of how these registers operate. For CIA #2, the latch value for the register is not specifically initialized during the reset or RUN/STOP-RESTORE sequences, although resetting the system will automatically set the latch count to 65535/$FFFF. The only use for this timer in system ROM is during the routines which receive bits from the RS-232 interface, where it is used to determine the duration of incoming bits. After RS-232 reception is completed, the timer will continue running with a latch count value dependent on the baud rate used in the RS-232 communications.

56584 $DD08 D2TOD1
56585 $DD09 D2TODS
56586 $DD0A D2TODM
56587 $DD0B D2TODH
Time-of-day clock registers

Refer to the discussion of the CIA #1 time-of-day clock registers at 56328-56331/$DC08-$DC0B for details of how these registers operate. Like the CIA #1 time-of-day clock, these registers are unused in 128 mode, and are available for your own timekeeping projects.

56588 $DD0C D2SDR
Serial data register

Refer to the discussion of the CIA #1 serial data register at 56332/$DC0C for details of how this register operates. Unlike the serial data line from CIA #1, the serial port (SP) line from CIA #2 is not used by the 128. It is, however, available at pin 7 of the user port on the back of the 128, along with the CNT Une (at pin 6), so you can use this port for your own interfacing projects.

56589 $DD0D D2ICR
Interrupt control register

Refer to the discussion of the CIA #1 interrupt control register at 56333/$DC0D for details of how this register operates. Because of the way CIA #2 is wired into the 128 system, the in-

terrupts it generates trigger processor NMI interrupts instead of the IRQ interrupts generated by CIA #1. The IOINIT routine [$E109], executed during both the reset and RUN/STOP-RESTORE sequences, initializes the interrupt mask for this register to 0/$00, disabling all interrupt sources. Any CIA #2 interrupt source can trigger an NMI interrupt, but 128 ROM routines use only three of the possible sources: NMI interrupts generated by FLAG, timer A, and timer B are used to drive RS-232 communications. You can use any CIA #2 source to generate an NMI interrupt for your own purposes. (The FLAG interrupt input line is available at pin B of the user port.) However, you'll have to write your own interrupt handling routine. The standard NMI handler [$FA40] assumes that any CIA #2-generated interrupt it encounters is for RS-232,

56590 $DD0E D2CRA
56591 $DD0F D2CRB
Control registers A and B

Refer to the discussions of CIA #1 control registers A and B at 56334/SDCOE and 56335/$DC0F, respectively, for details of how these registers operate. For CIA #2, these registers are both initialized to 8/$08 by the IOINIT routine [$E109], part of both the reset and RUN/STOP-RESTORE sequences. This value leaves both rimers stopped and set for one-shot mode. The only ROM routines which change those settings are the Kernal's RS-232 I/O routines. Timers A and B are used to generate the NMI interrupts which drive the transmission and reception of bits over the RS-232 interface. When RS-232 transmission is started, timer A is started, and it will continue running in continuous mode even after the transmission is completed and the logical file for RS-232 is closed (56590/$DD0E will be set to 1/S01). Likewise, timer B is started when the first RS-232 byte is received, and will continue running in continuous mode even after the reception is complete and the logical file has been dosed (56591/$DD0F will be set to **1/901).**

56592-56831 $DD10-$DDFF
CIA #2 register images
Due to incomplete address decoding, images of the CIA chip registers appear repeatedly every 16 bytes throughout the re-

mainder of this page of memory. That is, storing a value in any location in this range with an address that is an exact multiple of 16 greater than one of the base register addresses has the same effect as storing the same value in one of the base register locations. For example, storing a value in 56592/SDD10 or 56816/SDDFO has the same effect as storing a value in 56576/$DD00. However, it's better programming practice to use the officially designated register addresses.

I/O Expansion Slot #1

56832-57087/$DE00-$DEFF

This range of addresses is available for future additional I/O chips. No Commodore peripherals currently use this area, but it is possible that some third-party devices addressed in this area will appear. The original releases of CP/M for the 128 (those dated prior to December 6, 1985) expect to find a UART chip here for RS-232 serial communications, which is why the RS-232 portion of those versions doesn't work. No expansion card with a UART at this address was ever introduced, and more recent versions of CP/M properly support RS-232 communications in the standard fashion (via software). When no hardware chip is addressed here, all locations in this range will appear to contain unpredictable changing values when read, and writing to addresses in this range will have no effect.

I/O Expansion Slot #2

57088-57343/$DF00-$DFFF

This range of addresses is available for additional I/O chips. The REC (RAM expansion controller) chip in Commodore's RAM expansion modules is currently the only device to use this area, but it is possible that some other third-party devices addressed in this area will appear. See the following section for more information on the REC. When no hardware chip is addressed here, all locations in this range will appear to contain unpredictable changing values when read, and writing to addresses in this range will have no effect.

REC (RAM Expansion Controller) Chip Registers

57088-57098/$DF00-$DF0A

The REC chip is different from the other chips described in this chapter in that it isn't part of the 128's internal hardware. No REC chip will be present unless you have a model 1700 or 1750 RAM Expansion Module plugged into the memory expansion port, since the REC is part of the hardware in the module. However, with an expansion module installed, the registers for the REC appear here just like the other hardware chip registers. Other chips could be addressed in this I/O slot, but the Kernal DMA_CALL routine [$F7A5] and the BASIC STASH, FETCH, and SWAP statements all attempt to store values in the REC registers listed below.

Some would claim the expansion modules increase the amount of available RAM in the system. This claim is a bit misleading. The 8502 processor cannot directly access any of the 128K of RAM in a model 1700 expansion module or any of the 512K in **a** 1750. RAM from the modules never appears in the 128's normal address space; it cannot be used to fill in the missing RAM blocks 2 and 3 in the memory banking specifications. The REC chip is the 128's only gateway to the expansion module RAM. The REC is actually a highly specialized processor dedicated to the task of transferring the contents of blocks of memory. It can perform four bask operations: transferring data from system RAM to expansion RAM (called a stash operation), transferring data from expansion RAM to system RAM (called a fetch operation), exchanging the contents of an area of system RAM and an area of expansion RAM (called a swap operation), and comparing the contents of an area of system RAM and an area of expansion RAM (called **a** verify operation).

You'll notice that these are essentially the same basic functions performed by a disk drive, which is why the expansion module is sometimes referred to as a RAMdisk. The latest version of CP/M supports the expansion module as a virtual disk drive, referenced as drive M:. The 128-mode operating system doesn't have this feature built-in, but a program to provide a 128-mode RAMdisk would not be impossible to write.

The REC is a DMA (direct memory access) device, meaning that when called upon to perform a transfer operation it

actually shuts down the 8502 microprocessor and takes complete control of the system. Because the REC is optimized for the task of transferring data, it can perform its transfers with blinding speed. The specifications for the 1700 and 1750 modules claim transfer rates of 1 million bytes per second for stash, fetch, or verify operations, and half that rate for swap operations. In more practical terms, the REC can completely fill the 8502's 64K address space in about 1/16 second, or load an 8K bitmapped screen in 1/128 second—faster than the 8502 itself can move the equivalent amount of data.

Table 8-9 lists the REC registers. A detailed description of each register follows.

Table 8-9. REC Chip Registers

Address	**Register**
57088/$DFO0	Status register
57089/$DF01	Command register
57090/$DF02	System RAM base address (low byte)
57091/$DF03	System RAM base address {high byte)
57092/$DF04	Expansion RAM base address (low byte}
57093/$DF05	Expansion RAM base address (high byte)
57094/$DF06	Expansion RAM bank
57095/$DF07	Count of bytes to transfer (low byte)
57096/$DF08	Count of bytes to transfer (high byte)
57097/SDF09	Interrupt mask register
57098/$DF0A	Address control register

57088 SDFOO DMA ST

Status register

This register is read-only, meaning that writing values to this location has no effect on the setting of any register bits.

Bits 0-3: These bits hold a constant number (similar to a ROM location) indicating the version of the REC chip installed in the expansion module. In the initial release of expansion modules the value here is 0 (%0000), but this may change as revised versions of the REC are introduced.

Sit 4: This bit indicates the amount of RAM available in the expansion module. A %0 here indicates that the REC is in a mode] 1700 module with 128K of RAM, while a %1 here indicates that the REC is part of **a** model 1750 module with 512K of RAM.

Bit 5: This bit, the fault flag, is set to %1 when a mismatch is detected during a verify operation. The bit is set whenever a verify error occurs, regardless of whether or not the corresponding bit in the interrupt mask register (57097/$DF09) is set to trigger an interrupt on this condition. The bit is cleared to %0 whenever the register is read.

Bit 6: This bit, the end-of-block flag, is set to %1 when all bytes for the operation have been transferred or verified. The bit is set whenever an operation is successfully completed, regardless of whether or not the corresponding bit in the interrupt mask register (57097/$DF09) is set to trigger an interrupt on this condition. The bit is cleared to %0 whenever the register is read.

Bit 7: This bit, the interrupt pending flag, signals that the REC has generated an IRQ interrupt. The REC can generate an IRQ in response to two conditions: a verify error (fault) and the normal completion of an operation (end-of-block). Since the 8502 processor is inactive while the REC is performing a transfer, the IRQ is actually generated after the REC returns control to the 8502, No IRQ will be generated unless the interrupt enable bit in the register at 57097/$DF09 is set to %1 and one or both of the interrupt condition bits in that register are set.

The normal IRQ interrupt handling routine [$FA65] doesn't consider the REC as a source of interrupts, so you will have to write your own routine to process IRQs from the REC. See Appendix A for more information on interrupt handling. Bits 5-7 of this register are cleared to %0 whenever the register is read, so when testing this bit to determine whether an interrupt has occurred you must save the value read from the register if you wish to subsequently test bits 5 and 6 to determine which event has caused the interrupt.

57089 SDF01 DMA CMD

Command register

This register determines the type of operation to be performed by the REC. While it is possible to write directly to this register, the preferred practice is to use the Kernal DMA_CALL routine [$FF50], To use DMA_CALL, first set all the other REC registers to their desired values; then call the routine with the X register containing the system bank number for the op-

eration and the Y register containing the REC command (the value to be stored in this register). For example, BASIC supports stash, fetch, and swap operations, but not verify. The following routine is an example of how the current BASIC program text could be verified against the expansion RAM contents:

```
499 REM * SET SYSTEM STARTING ADDRESS TO START-OF-
    PROGRAM VALUE
500 BANK 15:POKE 57090,PEEK(45):POKE 57091,PEEK{46
    )
509 REM * SET NUMBER OF BYTES TO END-OF-PROGRAM MI
    NUS START-OF-PROGRAM
510 POKE 57095,PEEK(4624)-PEEK{45):POKE 57096,PEEK
    (4625)-PEEK(46)
519 REM * SET EXPANSION ADDRESS AND BANK
520 POKE 57092,0:POKE 57093,0:POKE 57094,0
529 REM * USE KERNAL DMA CALL ROUTINE {BANK = $00,
     COMMAND = $83)
530 SYS 65360,,0,131
539 REM * CHECK STATUS REGISTER FOR VERIFY ERROR F
    LAG
540 IF (PEEK(57088) AND 32 ) = 0 THEN PRINT"VEHIFY OK
    ":ELSE PRINT"BYTE MISMATCH AT ADDRESS";PEEK(45)
    +256*PEEK(46)+PEEK(57095)+256*PEEK(57096)-l
```

Of course, a verify error will occur if a copy of the program has not been previously stashed in the corresponding area of expansion memory.

Bits 0-1: These bits determine the type of operation to be performed by the REC. The four possible operations are specified as follows:

Bits
1 0 Operation
0 0 Stash (transfer from system memory to expansion RAM)
0 1 Fetch (transfer from expansion RAM to system memory)
1 0 Swap (exchange contents of system memory and expansion RAM)
1 1 Verify (compare contents of system memory and expansion RAM)

These bits are set to %00 during system reset. They retain their settings after an operation is completed.

pits 2-3: These bits are described in Commodore literature as "reserved," meaning that they have no function in the present version of the REC, but may in some future version. For now,

the bits seem merely to hold whatever value is written to them. They are set to %00 during system reset.

Bit 4: This bit, called the $FF00 flag, controls the execution mode for REC operations. When this bit is %0, operations do not begin immediately when bit 7 of this register is set to %1, but rather are deferred until a write to the MMU mode configuration register (65280/$FF00) occurs. This is convenient because it allows the memory configuration to be changed after the MMU registers are set up. When this bit is %1, the operation specified in bits 0-1 begins immediately when bit 7 is set. This bit is set to %1 during reset, and also whenever an operation is completed. Thus, the bit must always be specifically written with a %0 to enable the deferred execution option.

Bit 5: This bit, called the load flag, controls a special feature known as autoload mode. During a REC operation the system and expansion base registers and expansion bank register are incremented and the byte-count registers are decremented. When this bit is %0 {the default state after a reset), the registers are left at the end of an operation containing the final addresses and byte count. However, setting this bit to %1 enables the autoload feature, in which case the address, bank, and byte-count registers are automatically reloaded with their starting values after the operation is completed. This can be handy if you are repeatedly performing an operation involving the same area of memory.

Bit 6: This bit is described in Commodore literature as "reserved," meaning that it has no function in the present version of the REC, but may in some future version. For now, the bit seems merely to hold whatever value is written to it.

Bit 7: This bit, known as the execute flag, is used to signal the REC to begin the operation specified in bits 0-1. However, the start of the operation can be deferred. If bit 4 of this register is set to %1, the DMA process begins immediately when this bit is set. When bit 4 is %0, the operation does not actually start until the next time a value is stored in location 65280/$FF00 (the MMU configuration register). In this case, the bit remains set to %1 until the operation begins. In either case, the bit is cleared to %0 once the operation begins.

57090	$DF02	DMA ADL
57091	$DF03	DMA ADH

System memory base address registers

This register pair specifies the starting address in the 128's address space for the current REC operation. The first register (57090/$DF02) holds the low byte and the second (57091/$DF03) holds the high byte. The memory configuration in which this address is seen is determined by the values in the MMU configuration register (65280/$FF00) and RAM configuration register (54534/$D506). The configuration register setting determines which ROM, if any, will be seen in the configuration, and whether or not the I/O block will be visible. However, for REC data transfers the configuration register determines only whether or not RAM is visible—not the block from which the RAM is seen. That is, the setting of configuration register bits 6-7 is irrelevant to the REC. For DMA operations like REC transfers, the RAM block is instead determined by the setting of bits 6-7 of the RAM configuration register. If you want to transfer data to or from block 1 of RAM, you'll need to set bit 6 of location 54534/SD506 to %1.

That RAM configuration register bit also controls the block from which the VIC chip (another DMA device) gets its screen and character information, so be sure to reset the MMU register to its original value immediately after the REC operation is completed. Switching the VIC to a bank with no prepared screen data will turn the 40-column display to garbage, but in practice REC operations are completed so quickly that as long as the RAM configuration register bit is restored immediately after the REC operation is completed, the result of the switching is merely a barely noticeable flash of the screen. Of course, the VIC block switching has no visible effect whatsoever on the 80-column display.

One interesting consequence of the fact that the REC uses the VIC block setting rather than the configuration register block setting is that the REC has no trouble seeing the lowest IK of block 1 RAM, which is normally hidden from the processor under the common area from block 0. While bit 6 of the RAM configuration register is set to %1, the REC can freely transfer data to and from locations 0-1023/$0000-$03FF in block 1.

Both system base address registers are set to 0/$00 during system reset. During REC operations the address in the registers is usually incremented after each byte is transferred or verified, so that at the end of the operation the registers will hold a value one location higher than the last system memory address involved in the operation. If the address reaches 65535/$FFFF, it will roll over to 0/$0000, but the configuration will not change (the bank number will not be incremented). For verify operations that terminate with a fault (byte mismatch), the registers will hold an address which is one location beyond the one at which the mismatch was detected.

It is possible to have these registers automatically reloaded with the starting system memory address upon completion of an operation. See the entry for bit 5 of the command register at 57089/$DF01 for details of the autoload feature.

It also is possible to fix the system memory address so that it does not increment. In this case, all bytes in the operation will be read from or written to the same 128 memory location. See the entry below for the address control register at 57098/$DF0A for more information on this feature.

57092 $DF04 DMA LO
57093 $DF05 DMA HI

Expansion memory base address registers

This register pair specifies the starting address in the expansion module's address space for the current REC operation. The first register (57092/$DF04) holds the low byte and the second (57093/$DF05) holds the high byte. The expansion memory bank in which this address is seen is determined by the value in the expansion bank register at 57094/$DF06.

Both registers are set to 0/$00 during system reset. During REC operations the address in the registers is usually incremented after each byte is transferred or verified, so that at the end of the operation the registers will hold a value one location higher than the last expansion memory address involved in the operation. When the address in these registers exceeds 65535/$FFFF, the address wraps to 0/$0000, but the expansion bank register is also incremented. For example, the next byte after 65535/$FFFF in bank 3 of expansion RAM will come from location 0/$0000 in bank 4, The address will also

wrap from the last address in the last bank to the first address of the first bank. Thus, the REC scarcely notices the boundaries between the expansion memory banks. For verify operations that terminate with a fault (byte mismatch), the registers will hold an address which is one location beyond the one at which the mismatch was detected.

It is possible to have these registers automatically reloaded with the starting expansion memory address upon completion of an operation. See the entry for bit 5 of the command register at 57089/$DF01 for details of the autoload feature.

It is also possible to fix the expansion memory address so that it does not increment. In this case, all bytes in the operation will be read from or written to the same expansion module memory location. See the entry below for the address control register at 57098/$DF0A for more information on this feature.

57094 $DF06 DMA BNK

Expansion bank register

This register holds the expansion bank number for REC operations.

Bits 0-2: These bits determine the 64K bank of expansion memory for the address in the registers at 57092-57093/ $DF04-$DF05. The 1700 module has two 64K banks, so only the first two selections are valid for that model. The 1750 has eight 64K banks. The possible bank selections are as follows:

Bits	Expansion bank
2 1 0	0
0 0 0	1
0 0 1	2
0 1 0	3
O i l	4
1 0 0	5
1 0 1	6
1 1 0	7
1 1 1	

tf all the bytes involved in a REC operation are located within one bank, this register will still hold its original value after the operation is completed. However, when the value in the expansion address register rolls over from 65535/$FFFF to 0/$0000, the value here will be incremented, unless this register already

holds the highest bank number. In that case the bank number will be reset to 0. These bits are reset to %000 when the computer is reset.

Bits 3-7: Unused. Writing to these bits has no effect. The bits always return %1 when read, so the value returned when this register is read will always be at least 248/$F8. To get the true bank number when reading the register, you should mask off these bits. (Use AND 3 in BASIC or AND #$03 in machine language.)

It is possible to have this register automatically reloaded with the starting bank number upon completion of an operation. See the entry for bit 5 of the command register at 57089/ $DF01 for details of the autoload feature.

57095 $DF07 DMA DAL
57096 $DF08 DMA DAH

Count of bytes to transfer

This pair of registers holds the number of bytes to be transferred or verified in the current operation. The first register (57095/$DF07) is the low byte of the count, and the second (57096/$DF08) is the high byte. As each byte is transferred or verified, the value in these registers is decremented. The registers will always hold the value 1 ($01 $00) after an operation is successfully completed. If a verify operation stops because of a fault (byte mismatch), the value in the registers will be the original value minus the number of bytes which have been successfully verified when the fault occurs. All bits in these registers are set to %1 (equivalent to a byte count of 65535/ $FFFF) during system reset.

It is possible to have these registers automatically reloaded with the starting byte count upon completion of an operation. See the entry for bit 5 of the command register at 57089/$DF01 for details of the autoload feature.

57097 $DF09 DMA SUM

Interrupt control register

The REC can generate an 8502 IRQ interrupt on two conditions: When a byte mismatch is detected during a verify operation and when all bytes have been successfully transferred or verified. No IRQ will be generated unless bit 7 and one or both of bits 5-6 are set to %1. Bit 7 of the status register at

57088/$DF00 signals when the REC has generated an IRQ, and bits 5-6 of that register indicate which condition caused the interrupt. After being set to %1, the bits in this register remain set until specifically cleared. Thus, once REC interrupts are enabled, they remain enabled until at least bit 7 of this register is reset to %0. Bits 5-7 are all reset to %0, disabling interrupts, when the system is reset.

Bits 0-4: Unused. Writing to these bits has no effect. The bits always return %1 when read, which means that the value returned when this register is read will always be at least 31/$1F.

Bit 5: Setting this bit to %1 specifies that an IRQ interrupt is to be generated when a verify operation terminates because the bytes being compared do not match. However, no interrupt will occur unless bit 7 of this register is also set to %1.

Bit 6: Setting this bit to %1 specifies that an IRQ interrupt is to be generated if the current REC operation is completed without errors, the end-of-block condition. However, no interrupt will occur unless bit 7 of this register is also set to %1.

Bit 7: Setting this bit to %1 allows IRQ interrupts to be generated on either of the conditions specified in bits 5-6. No interrupt will occur unless this bit is set, regardless of the settings of bits 5-6. However, setting this bit to %1 won't produce interrupts unless one or both of bits 5-6 are also set to %1. Changing this bit to %0 does not affect the setting of bits 5-6.

57098 $DF0A DMA VER

Address control register

Bits 0-5: Unused. Writing to these bits has no effect. The bits always return %1 when read, which means that the value returned when this register is read will always be at least 63/$3F.

Bits 6-7: These bits control the incrementing of the base address registers. The four possible settings are as follows:

Bits	
7 6	Address register status
0 0	Both addresses increment
0 1	Only the system (128) address increments
1 0	Only the expansion memory address increments
1 1	Neither address increments

The first option, both addresses increment, is the default setting after power on or reset. With this setting, both registers are incremented after each byte is transferred or verified, so that the operation involves a sequential series of locations in both system and expansion memory. However, it is possible to fix either the system or expansion address. This allows you to fill either system or expansion memory with a repeating value. For example, to fill an area of expansion memory with a particular value, you would set these bits to increment only the expansion address register, then store the value in a system memory location and transfer the value to expansion memory the desired number of times. The following example uses BASIC to fill bank 1 of expansion memory with zeros:

```
500 POKE 254,0: REM PLACE A ZERO IN SYSTEM MEMORY
510 BANK 15:POKE 57098,128: REM FIX SYSTEM ADDRESS
520 STASH 65535,254,0,1: REM FILL EXPANSION MEMORY
    WITH THE VALUE IN SYSTEM LOCATION 254
530 POKE 57098,0: REM RESTORE REC TO NORMAL MODE
```

You could use a similar technique to fill an area of system memory with the contents of an expansion memory location. The final option, neither address incrementing, is useful only when a single byte is being transferred,

57099-57119 $DFOB-$DF1F Unused

When the REC is present, all the unused register addresses in this range return the value 255/$FF when read. Writing to these addresses has no effect.

57120-57343 $DF20-$DFFF
REC chip register images

Due to incomplete address decoding, images of the REC chip registers appear repeatedly every 32 locations throughout the remainder of this page of memory. That is, storing a value in any location in this range with an address an exact multiple of 32 greater than one of the REC register base addresses has the same effect as storing that value in the base register location. For example, the effect of storing a value in 57121/$DF21 or 57313/$DFE1 is the same as storing the value in 57089/$DF01. However, it's better programming practice to use the officially designated register addresses.

Character Pattern ROM

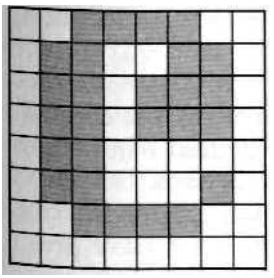
53248-57343/$D000-$DFFF

This ROM contains character shape information for the VIC 40-column video chip, and also, indirectly, for the VDC 80-column video chip. Because character shapes are drawn in an 8-dot X 8-dot matrix, each character pattern consists of 64 dots, each of which can be on or off. To store pattern information, on-dots are represented by %1 bits and off-dots are represented by %0 bits. Thus, the pattern for each character requires 64 bits, or eight 8-bit bytes. Since the 128 can display 256 different characters, a complete character set requires 256 * 8 = 2048 bytes (2K). This 4K ROM has room for two complete character sets. The first, at 53248-55295/$D000-$D7FF, is known as the uppercase/graphics set. It is the default character set for the 128—the one you see when you turn the computer on.

The second set, at 55296-57343/$D800-$DFFF, is known as the lowercase/uppercase set. Character patterns are stored in the ROM in screen code order. In fact, that's what screen codes are—indexes into character ROM. See Appendix C for a table of character patterns for both sets.

As an example, consider the first character pattern in the ROM, consisting of the eight bytes at 53248-53255/ $D000-$D007. As the first pattern, it corresponds to the character with screen code 0/S00, the @ character. The pattern is shown in Figure 8-18.

Figure 8-18. Typical Character Pattern



```
% 00111100 •» 53C
% 01100110 - S66
% 01101110 - $6E
Vn 01101110 - $6E
% 01100000 - $60
% 01100010 - S62
% 00111100 = $3C
% 00000000 = $00
```

For the VIC chip, this character ROM is the normal source of character data. Bits 1-3 of the VIC register at 53272/$D018 control which 2K block within the current 16K VIC video bank is seen as the character source. Since the VIC's default video bank 0 corresponds to locations 0-16383/$0000-$3FFF in the processor's address space, it would seem impossible for the character ROM at 53248/$D000 to be used as the normal character source. However, the 128's memory manipulation capabilities are used to make the ROM appear to the VIC at an address within the video bank. The uppercase/graphics set appears at 4096-6143/$1000-$17FF and the lowercase/uppercase set appears at 6144-8191/$1800-$lFFF. Only the VIC chip sees the character ROM at those addresses; when the MMU configuration register is set to allow the microprocessor to see the ROM, the processor always sees it at 53248/$D000. The VIC chip, on the other hand, never sees the ROM at its actual address. That couldn't be possible; remember, the VIC itself resides at 53248-53296/$D000-$D030.

The VIC can see the character ROM in any of its four video banks. In each case, the uppercase/graphics set appears to the VIC at an offset of 4096/$1000 from the starting address of the video bank, and the lowercase/uppercase set appears at an offset of 6144/$1800 from the first address of the video bank. It is possible to disable the ROM image feature so that the VIC never sees character ROM at all. This would be useful, for example, if you wished to place a custom character set in the free RAM at 6144-7167/$1800-$1BFF. Bit 2—called the CHREN bit—in the 8502's on-chip I/O port at location l/$01 determines whether or not the ROM image is visible in the current VIC video bank. See Chapter 2 for more information on the I/O port bit and its associated shadow location, 217/$D9.

When setting up custom character sets, you may want to copy all or part of the character pattern data from this area of ROM into RAM. Remember that the only standard bank configuration in which character ROM is visible is bank 14, The following BASIC lines will copy the uppercase/graphics set into RAM at 8192-10239/$2000-$2800. (You must have previously protected this area from BASIC. Using GRAPHIC 1:GRAFHIC 0 to reserve the bitmapped screen area is a simple way to do this.)

```
100 BANK 14:CA = 53248:CS = 8192
110 FOR 1=0 TO 2047
120 POKE CS + LFEEK(CA+D
130 NEXT I
```

To copy the lowercase/uppercase set, change the value of the variable CA to 55296. The following routine shows one method for accomplishing the same thing in machine language:

```
1400  LDA #$08     ;number of 256-byte pages
1402 STA $FA       ; to be copied
1404  LDA #$00     ;load pointer at $FB with
1406 STA $FB       ; target address ($2000)
1408  LDA #$20
140A STA $FC
140C  LDA #$00     ;load pointer at $FD with
140E STA $FD       ;source address ($D000)
1410  LDA #$D0
1412 STA $FE
1414  LDY #$00     initialize offset
1416  LDX #$0E     ;read from bank 14
1418  LDA #$FD     ;use $FD as source pointer
141A  JSR $FF74    ;Kernal INDFET routine
141D  STA ($FB),Y  ;store in target address area
141F INY
1420 BNE $1416     ;repeat for 256 bytes per page
1422 INC $FC
1424 INC $FD       ;increment page pointers
1426 DEC $FA
1428 BNE $1414     ;decrement block count
142A  RTS
```

Refer to the discussion of the VIC chip earlier in this chapter for more information on creating and using custom character sets.

As mentioned, this ROM indirectly supplies character patterns for the VDC (8563) 80-column video chip. That chip has no character ROM of its own, so the contents of this ROM are copied into the VDC's private block of RAM during the reset sequence (see the INIT80 routine [$CE0C] in Chapter 7). Refer to the section on the VDC earlier in this chapter for more information on how the 80-column character set is managed.

Kernal ROM

In software engineering jargon, the collection of subroutines that perform basic input and output functions for a computer is referred to as the operating system kernel. The developers of the kernel for the original Commodore PET spelled (or misspelled) the term as *kernal,* and Commodore operating systems have been referred to as the *Kernal* ever since. There are both similarities and significant differences between the 128 Kernal and the Kernals of earlier models.

The Kernal handles input from or output to five basic sources: the keyboard, the video screen, the tape drive (Datassette), the RS-232 port, and the serial bus (to which disk drives and printers are connected). In the 128 Kernal, all keyboard and video functions have been transferred to a separate block of ROM, the screen editor, at 49152/$C000 (see Chapter 7 for details).

Significant enhancements include the addition of an 80-column screen, ESC-key screen-editing sequences, and keyboard table pointers in RAM that make it easy to customize the keyboard. Tape and RS-232 support is largely unchanged from that provided in the Commodore 64 Kernal. Serial bus operation is significantly enhanced by the addition of a new fast serial mode which can transfer data much more quickly than the old system, now fittingly referred to as slow serial mode.

Additional new features of the 128 Kernal include routines to handle the storage, retrieval, and comparison of data from the various memory banks supported by the system, and support for DMA (Direct Memory Access) transfer operations to and from the 1700 and 1750 Memory Expansion Modules.

The heart of the Kernal is the collection of routines called by the Kernal jump table at 65409-65525/$FF81-$FFF5. The routines called from that table, and their supporting subroutines, make up the bulk of the Kernal and provide access to the majority of the 128's input/output *(I/O)* capabilities. Almost any I/O operation can (and should) be performed through the appropriate jump table entry. The Kemal jump table has been a feature of all Commodore operating systems.

The 128 adds an additional jump table with 19 new entries at 65351-65407/$FF47-$FF7E These entries provide access to most of the 128 Kernal's new or enhanced features,

57344 $E000 RESET

Performs main system initialization sequence.
Resets the processor stack pointer to the top of the stack, disables IRQ interrupts, and insures that the processor is not in decimal mode; then sets the MMU configuration register for bank 15. Other MMU registers are initialized from the table at 57419/$E04B. Next, the initialization status flag (2564/$0A04) is reset to 0/$00 to indicate that all variables and vectors need to be initialized. The interrupt and reset handling routines at 65285-65348/$FF05-$FF44 in Kernal ROM are copied to that same area in all RAM banks, and the INDFET, INDSTA, INDCMP, J5RFAR, JMPFAR, and DMA_CALL routines are copied from the table at 63488/$F800 into bank 0 RAM.

Locations 65525-65527/$FFF5-$FFF7 in bank 1 are then examined to see if they contain the character codes for the letters CBM, If not, those locations are initialized with that character pattern, and the soft reset vector at 65528/$FFF8 in bank 1 is initialized. However, if the test pattern is found (indicating that RESET has already been performed at least once), the routine jumps to the address in the vector. Normally, the vector points to the routine at 57892/$E224, which simply reinitializes the test pattern and vector. You can change the address to add your own extra steps to the reset sequence. See the soft reset vector entry in Chapter 4 for details.

The subroutine at 57922/$E242 is called to check for the presence of a Commodore 64 cartridge. If one is detected, the system is switched to 64 mode and the computer becomes a Commodore 64. (You must press the RESET button or turn the computer off and back on to return to 128 mode.) Otherwise, the subroutine records the presence of any 128 function ROMs in the table at 2753-2756/$0ACl-$0AC4. If a logged function ROM is autostarting, its cold start routine is called. It is possible that an autostarting ROM will retain control of the system and will not return to complete the reset sequence.

Next, the IOINIT routine [$E109] is called to initialize the video, CIA, and SID chip registers. The keyboard column which includes the RUN/STOP and Commodore keys is scanned. If RUN/STOP has been pressed, the Kernal memory

initialization flag (2562/S0A02) is checked. If the flag contains a nonzero value, the following memory initialization step is skipped. Since the flag is given the value 165/SA5 after the first call to RAMTAS [$E093], zero-page values and memory pointers are preserved if RUN/STOP is held down during a subsequent reset. Otherwise, RAMTAS [$E093] is called to clear all zero-page RAM locations and reestablish Kernal pointers.

RESTOR [$E056] is called to load default Kernal indirect vectors into 788-819/$0314-$0333. CINT [$CO0O] is called to initialize the screen editor, after which IRQ interrupts are once again allowed. If the RUN/STOP key has been pressed, the monitor is entered through its cold start entry point [$B000]. If the Commodore key has been held down, 64 mode is entered using the C64_MODE routine [$E24B], Otherwise, BASIC 7.0 is entered via the restart vector at 2560/S0A00. If the RAMTAS step has been performed, the restart vector will point to BASIC'S cold start entry point [$4000].

It is possible to perform most of the reset sequence without losing the BASIC program currently in memory. Simply hold down RUN/STOP while pressing the RESET button, This will skip the RAMTAS step, which would wipe out important program pointers. You will land in the monitor after the reset; type X to exit to BASIC, where the current program should still be intact.

Note that the reset routine does not explicitly attempt to boot a disk or to initialize function ROMs that are not autostarting. These tasks are performed by the Kernal PHOENIX routine [$F867]. In 128 ROM, PHOENIX is called only during the BASIC cold start routine [$4023]. However, as long as the RUN/STOP or Commodore key is not held down, the reset routine ends by jumping to the BASIC cold start routine, so those actions are implicitly part of the normal reset sequence.

57419 $E04B

Table of default MMU register settings.
The 11 values in this table are copied into the MMU chip registers at 54528-54538/$D500-$D50A by the system reset routine [$E000]. See Chapter 8 for details of the function and default settings of the registers.

57430 SE056 RESTOR

Restores Kernal indirect vectors to their default values.
{This routine has a jump table entry at 65418/$FF8A.)

Loads the X and Y registers with the value 57459/$E073, the address of the default vector table, then clears the status register carry bit, and falls through into the following routine to load default vector values.

57435 $E05B VECTOR

Loads or copies Kernal indirect vector values.
(This routine has a jump table entry at 65421/$FF8D.)

Transfers the address value in the X and Y registers upon entry into a working pointer (195-196/$C3-$C4). If the carry bit is clear, 32 bytes starting at the specified address are copied to the Kernal indirect vectors at 788-819/$0314-$0333. If carry is set, the contents of the indirect vectors are copied to 32 locations starting at the specified address. In either case, the target address must be visible in bank 15.

57459 $E073

Table of default Kernal indirect vector values.

The 16 two-byte values in this table are copied to the Kernal indirect vectors (788-819/$0314-0$333) by the RESTOR routine [$E056], part of the reset sequence. See Chapter 2 for the default vector target addresses.

57491 SE093 RAMTAS

Initializes zero page and Kernal pointers.
(This routine has a jump table entry at 65415/$FF87.)

Sets all zero page RAM locations (2-255/$02-$FF) to 0/$00, then initializes the cassette buffer pointer (178-179/$B2-$B3) to 2816/S0B00, the RS-232 input buffer pointer (200-201/$C8-$C9) to 3072/$0CO0, and the RS-232 output buffer pointer (202-203/$CA-$CB) to 3328/SODOO. The MEMSIZ pointer (2567-2568/$0A07-$0A08) is set to 65280/$FF00 to mark the top of free RAM, and the MEMSTR pointer (2565-2566/$0A05-$0A06) is set to 7168/$1COO to mark the bottom of free RAM in bank 0. The BASIC restart indirect vector (2560-2561/$0A00-$0A01) is loaded with 16384/$4000, the address of BASIC'S cold start entry point. Finally, the Kernal memory initialization flag (2562/$0A02) is set to 165/$A5 to indicate that this routine has been performed.

57549 $E0CD

Initializes all RAM-resident Kernal routines.
Copies the interrupt- and reset-handling routines at 65285-65348/$FF05-$FF44 in Kernal ROM into the same addresses in both RAM banks. These routines redirect interrupts and reset to the proper handling routine in Kernal ROM (bank 15). It's necessary to have a copy in each bank, because an interrupt or reset can occur while the system is configured for any bank. The routines are actually copied into banks 0-3, even though there isn't unique RAM in banks 2 and 3 in the current 128.

Next the code for the RAM-resident portions of vital indirect access routines—INDFET [$02A2], INDSTA [$02AF], INDCMP [$02BE], JSRFAR [$02CD], and JMPFAR [$02E3]—is copied from the table at 63488/$F800 into page 2 of the common area of RAM. Finally, the code for the DMA_CALL execution routine [$03F0] is copied from 63578/$F85A into page 3 of the common area of RAM.

57609 $E109 **IOINIT**

Initializes I/O chip registers.
(This routine has a jump table entry at 65412/SFF84.)

Begins by disabling all interrupt sources on both CIA chips and halting all CIA timers. All lines from the four CIA I/O ports (ports A and B on both chips) are assigned their default directions, input or output. Output lines are also assigned their default states, high or low. See Chapter 8 for more information on the functions and default settings of these lines. The direction and status of lines connected to the processor's built-in I/O port are also established. See Chapter 2 for more information on the functions of these lines.

The VIC-II video chip's raster compare register is tested to determine whether the raster line count ever reaches the value 264/$108. This indicates which video system, NTSC (North American) or PAL (European), is being used. The video system depends on the VIC-II chip currently installed. There are separate versions for NTSC and PAL. The highest possible raster line for an NTSC system is 263, so a value of 264 here means that the system is using PAL. The NTSC/PAL flag (2563/$0A03) is set accordingly—to 0/$00 for NTSC or 255/$FF for PAL. By designing the Kernal to adjust itself for either system. Commodore's engineers avoided the need for

separate versions of the Kernal for North American and European 128s.

Next, a number of Kerna! I/O flags are cleared: clock mode storage (2615/$0A37), IRQ vector storage (2570/$0A0A), custom mode setting (2618/$0A3A), and jiffy count compensation (2614/$0A36), The keyboard (device 0) is made the current input device (153/S99), and the screen (device 3) is made the current output device (154/$9A).

VIC-II chip registers (53248-53296/$D000-$D030) are initialized from the table at 58055/$E2C7, and 8563 video chip registers are initialized using the subroutine at 5782O/$E1DC. If the version number of the 8563 chip (bits 0-2 of the register at 54784/$D600) indicates that one of the newer revisions of that chip is installed, the subroutine is called again to adjust the horizontal scrolling register (R25). If the PAL video system is in use, the subroutine is called yet again to adjust vertical display registers (R4 and R7). Bit 7 of the initialization status flag (2564/$0A04) is tested. If the bit is set to %1, IOINIT has been called at least once before, so the following step, which sets up the 80-column character set, is skipped. If the bit is %0 (as will normally be the case when this routine is called as part of the reset sequence), the screen editor INIT80 routine [$C027] is called to copy the ROM character definitions into 8563 RAM. Then bit 7 of the initialization status flag will be set to %1 to indicate that the step has been performed.

All registers for the SID sound chip (54272-54296/$D400-$D418) are cleared to zero to disable any sound output; then VIC-II raster interrupts are enabled. The raster interrupt (set by the default table value to occur at scan line 255) is the normal source of jiffy IRQ interrupts for the 128. The fast serial flag (2588/$0AlC) and RS-232 activity flag (2575/$0A0F) are cleared to zero. Timer B of CIA #1 is loaded with 65535/$FFFF and started counting continuously. Finally, a fast serial mode setup sequence is performed.

57820 $E1DC

Initializes 80-column video chip registers.

Retrieves a byte from the position in the table at 58104/$E2F8 specified in the X register. The value is used as a register number and the next value in the table is written to that 8563 register, Once called, the routine repeatedly reads register

numbers and initializes registers until a value greater than 127/$7F is read for the register number. Normal X register values upon entry are 0/$00 (to initialize the 8563), 59/$3B (to adjust horizontal scrolling for different versions of the 8563 chip), or 62/S3E (to adjust register settings for a PAL video system).

57840 $E1FO

Initializes or jumps through the soft reset vector.

Examines the contents of locations 65525-65527/$FFF5-$FFF7 in bank 1 to determine whether those locations contain the character codes for the letters *CBM*. If not, a branch is taken to the following routine to initialize the test pattern and vector. If the pattern is found, the reset vector has already been initialized, so the address in the vector at 65528-65529/$FFF8-$FFF9 in bank 1 is loaded into locations 2-3/$02-$03. The routine then takes an indirect jump to the specified address. (The system will still be configured for bank 15, so the target routine must in that bank.)

57892 **$E224**

Initializes the soft reset vector.

Loads the system soft reset vector, locations 65528-65529/$FFF8-$FFF9 in bank 1, with the value 57892/$E224, the address of this routine. Next, the character codes for the letters *CBM* are copied to locations 65525-65527/$FFF5-$FFF7 in bank 1 to indicate that the vector has been initialized.

57922 $E242

Checks for the presence of 64 cartridges or 128 function ROMs.

Tests the GAME and EXROM lines from the memory expansion port (reflected by bits 4 and 5 of the MMU mode configuration register at 54533/$D505). If either of the lines is grounded, the routine falls through to enter 64 mode. Otherwise, a branch is taken to the routine at 57963/$E26B to check for 128 function ROMs.

57931 **$E24B** C64_MODE

Switches the system into 64 mode.

(This routine has a jump table entry at 65357/$FF4D.)

Loads the processor data direction and I/O port registers (locations 0-l/$00-$01) with their standard Commodore

64 settings, copies the reset routine from 57955-57962/ $E263-$E26A to 2-9/$02-$09, stores a zero in the clock rate register (53296/$D030) to insure that the system is in slow (1 MHz) mode, then jumps to the reset-to-64 routine at 2/$02. The short routine copied there stores the value 247/$F7 in the MMU mode configuration register. Compared to the default setting, that value clears bit 3 (prohibiting fast serial output) and sets bit 6 (making 64 ROM visible while making 128 ROM and the MMU chip registers invisible). The routine then initializes 64 mode by jumping through the hardware reset vector (65532/$FFFC) in the now-visible Commodore 64 ROM.

57963 $E26B

Logs 128 function ROMs.

Clears the function ROM ID table (2 753-2 *75* 6/$0ACl-$0AC4), then checks the seventh through ninth bytes beyond the starting address in each of the four possible memory slots for function ROM: 32768/$8000 and 49152/$C000 in bank 8 for external (cartridge) ROM, and 32768/$8000 and 49152/$COO0 in bank 4 for internal ROM (in the free socket on the 128's main circuit board). If the bytes are the character codes for the letters *CBM,* a valid function ROM is present in the slot, and the sixth byte beyond the starting address (the cartridge ID) is retrieved and stored in the ID table. If the ID value is 1, indicating an autostarting ROM, the starring address of the ROM is loaded into the JSRFAR pointer (3-4/$03-$04), and JSRFAR [$02CD] is used to call the ROM's cold start routine. (It is possible that the autostarting ROM will retain control of the system and not return.) ROMs that don't autostart are initialized during the PHOENIX routine [$F867J.

58052 $E2C4

Initialization test pattern.

These three bytes are the character codes for the letters *CBM,* used as a test pattern in various operations: testing whether the soft reset vector has been initialized [$E1FO], checking for 128 function ROM [$E26B], and checking for boot disks [$F890].

58055 SE2C7

Table of default VIC chip register values.
The 49 values in this table are copied to the V1C-II 40-column video chip registers during the IOINIT routine [$E109] to establish default register settings. See Chapter 8 for more information on the registers and their default settings.

58104 $E2F8

Table of default 8563 chip register values.
The first value in each two-byte table entry is the number of the 8563 80-column video chip register into which the second value is to be copied. The register settings are initialized by the routine at 5782O/$E1DC.

58171 $E33B TALK

Sends TALK command to a serial device.
(This routine has a jump table entry at 65460/$FFB4.)

Sets bit 6 of the device number value in the accumulator to %1 (the serial bus TALK command has the format %010?jnrittn, where *%nnnnn* is the number of the device being commanded to talk). A BIT opcode is used to fall through into the next routine to send the byte as a serial bus command.

58174 $E33E LISTEN

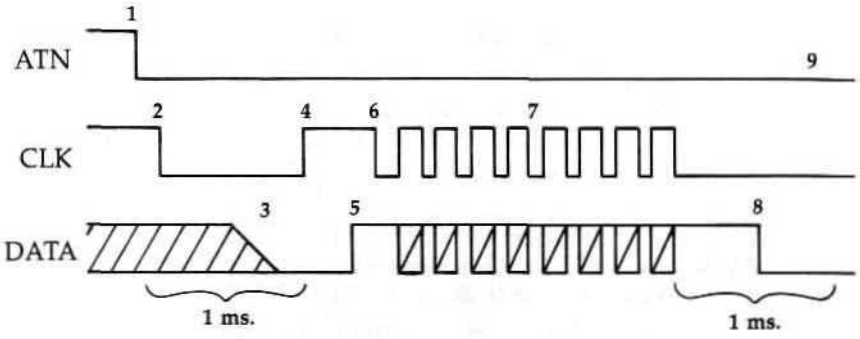
Sends LISTEN command to a serial device.
(This routine has a jump table entry at 65457/$FFBl.)

Sets bit 5 of the device number value in the accumulator to %1 (the serial bus LISTEN command has the format *%001nnnnn*, where *%nnnnn* is the number of the device being commanded to listen). RS-232 activity is disabled. If a serial byte is awaiting transmission in the buffer at 149/$95, it is sent with an EOI (end-or-identify) handshake; then the command byte is placed in the buffer.

The subroutine at 58739/$E573 is used to disable IRQ interrupts and standardize timing. Then the serial bus DATA line is allowed to go high. If the serial bus ATN line is not currently low, the routine attempts to establish fast serial mode. It does this by setting the serial port for fast serial output and sending out the value 255/$FF (eight %1 bits) using the fast serial hardware. The port is then set for fast serial input mode, and a short delay loop is executed. If a serial device

capable of fast serial communications is present, it should respond by sending back a byte. This will cause a serial register interrupt on CIA #1, which will be detected later to determine that fast serial mode is available.

Figure 9-1. Sending a Serial Command Byte



1. 128 pulls ATN line low.
2. 128 pulls CLK line low and allows the DATA line to go high.
3. The external device must respond by holding the DATA line low. If the DATA line is still high after approximately one millisecond, it is assumed that the device is not present.
4. If the external device responded, the 128 allows the CLK line to go high.
5. The external device must now allow DATA to go high again. (The 128 will wait indefinitely for this to happen.)
6. Once DATA goes high, the 128 responds by pulling the CLK line low.
7. The command byte is then sent one bit at a time, starting with the least significant bit (bit 0). Command bytes are always sent in slow serial mode. To send a bit, the DATA line is set either low or high, depending on whether the bit being sent is %0 or %1. Then the CLK is allowed to go high briefly to signal that a valid bit can be read on the DATA line.
8. After the last data bit is sent, the 128 checks the DATA line. The external device must pull that line low within approximately one millisecond or a write-timeout error will be indicated.
9. The status of the ATN line after a command is sent depends on the command. If it is TALK or LISTEN, ATN remains high so that the secondary address can be sent as a command as well. However, ATN is immediately pulled low following UNTALK and UNLISTEN commands.

Next, the serial bus ATN line is pulled low to indicate that the following byte will be a command. The serial bus CLK line is pulled low, and the DATA line is allowed to go high. A delay loop of approximately one millisecond is executed, and the routine falls through into the following routine to send the command byte. Figure 9-1 illustrates the process.

Note that there is no corresponding routine for the 128 to receive a serial bus command. The 128 has no ATN input line, so it cannot be commanded to listen (it only listens "voluntarily"). It must always be the only master device on the serial bus.

58252 SE38C

Sends buffered **byte to** a serial device.
Begins by calling the subroutine at 58739/$E573 to disable IRQ interrupts and standardize timing. If the system is set for the fast {2 MHz) clock mode, it will be temporarily reset to the normal (1 MHz) mode, which explains why serial communications are not significantly affected by the clock mode.

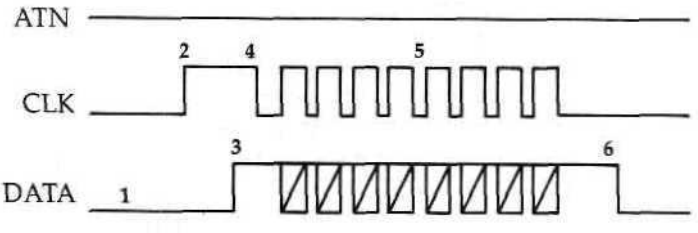
The routine makes sure that the serial bus DATA line is free to go high, then tests the state of the line. If the external device is not holding DATA low, the device is considered not present, so bit 7 of the serial status flag (144/$90) will be set to %1, the ATN and CLK lines will be allowed to go high, and the routine will exit. Otherwise, the routine will allow the CLK line to go high. If bit 7 of the EOI flag (163/$A3) is set to %1, the routine performs the EOI (end-or-identify) handshake by waiting for the external device to set the DATA line high and then low. Next, the routine waits for the external device to release the DATA line to a high state. While it's waiting, the routine checks whether a CIA #1 serial register interrupt has occurred, indicating that a byte has been received via the fast serial hardware. If so, the external device can accept fast serial input, and the fast serial flag (2588/S0A1C) is set to 192/$C0 to indicate this.

Once the DATA line goes high, the routine immediately pulls the CLK line low and proceeds to send the byte of data. If fast serial mode is available, the process is simple: The serial port is set for fast serial output, and the buffered byte from 149/$95 is stored in the CIA #1 serial data register (56332/$DC0C). After that, the transfer is automatic, handled by the

CIA chip hardware. The routine simply waits until a serial register interrupt indicates that the byte has been completely sent, then resets the port for fast serial input (its default state).

The process for sending a byte in standard (slow) serial mode is more complicated because it is handled in software. Bits are pulled from the buffered byte (149/$95), one at a time, starting with the least significant tit (bit 0). For each bit, the serial bus DATA output line is set either high or low, depending on whether the bit to be sent is %0 or %1. Then the CLK line is allowed to go high to signal to the external device that a valid data bit can be read from the DATA line. After a brief delay, the CLK line is pulled low, and the DATA line is

Figure 9-2. Sending a Serial Data Byte

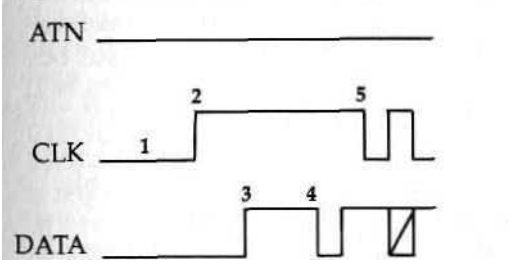


1. The 128 allows the DATA line to go high. If the external device does not continue to hold the line low, it is assumed that the device is not present.
2. The 128 allows the CLK line to go high.
3. The external device must now allow DATA to go high. (The 128 will wait indefinitely for this to happen.)
4. Once DATA goes high, the 128 responds by pulling the CLK line low.
5. The data byte is sent one bit at a time using either fast or slow serial mode. To send a bit in slow serial mode {illustrated above), the DATA line is set either low or high, depending on whether the bit being sent is %0 or %1. Then the CLK line is allowed to go high briefly to signal **that** a valid bit can be read on the DATA line. Fast serial mode works in a similar manner, except the transfer is managed by CIA chip hardware rather than ROM software, and the bits are clocked by pulses on the SRQ line rather than on the CLK line,
6. After all of the byte is sent, the 128 checks the DATA line. The external device must pull that line low within approximately one millisecond, or a write-timeout error will be indicated.

allowed to go high again. If the external device holds the DATA line low between bits, a write-timeout occurs (the routine sets bits 0 and 1 in the serial status flag to %1, allows the ATN and CLK lines to go high, and exits).

After the byte has been sent (in either mode), the routine waits for the external device to pull the DATA line low. If this does not happen within approximately one millisecond, a write-timeout occurs, and the routine sets bits 0 and 1 in the serial status flag to %1, allows the ATN and CLK lines to go high, and exits. Otherwise, the routine at 58783/$E59F is used to restore interrupts and the clock mode setting, then exits with the status register carry bit clear. The Y register is unused during the routine, and the X register value is preserved. Figure 9-2 illustrates the process of sending a data byte. Figure 9-3 illustrates the EOI handshake.

Figure 9-3. Serial EOI Handshake



200 *is.*

1. As in a normal byte transfer, the external device must hold the DATA line low or it will be assumed that the device is not present.
2. The 128 allows the CLK line to go high.
3. The external device should then allow the DATA line to go high. (The 128 will wait indefinitely for this to happen.)
4. When the 128 does not respond by pulling the CLK line low within 200 microseconds, the external device should recognize that an EOI handshake is being performed and should respond by pulling DATA low again. (The 128 will wait indefinitely for DATA to go low.)
5. The external device should then release DATA high again. The 128 will respond by pulling CLK low, and the final byte of the file is then sent in the usual manner. (See steps 5-6 of Figure 9-2.)

58430 8E43E ACPTR

Reads a byte from a **serial** device.
(This routine has a jump table entry at 65445/$FFA5.)

Begins by calling the subroutine at 58739/$E573 to disable IRQ interrupts and standardize timing. If the system is set for the fast (2 MHz) clock mode, it will be temporarily reset to the normal (1 MHz) mode, which explains why serial communications are not significantly affected by the clock mode. The routine insures that the serial bus CLK line is free to go high, then tests the state of the line and waits in a loop until it goes high. Next, a delay counter is initialized, and the DATA line is allowed to go high. If the external device responds by pulling the CLK line low before the delay count expires, the routine begins to process the incoming data bits. Otherwise, it assumes that the external device is requesting an EOI (end-or-identify) handshake. So the EOI bit (bit 6) of the serial status flag (144/$90) is set to %1, and the DATA line is pulled low, then allowed to go high again. The external device must acknowledge the EOI handshake by pulling the CLK line low before another delay period expires. If it doesn't, a read-timeout occurs, and the routine sets bit 1 in the serial status flag to %1, allows the ATN and CLK lines to go high, and exits.

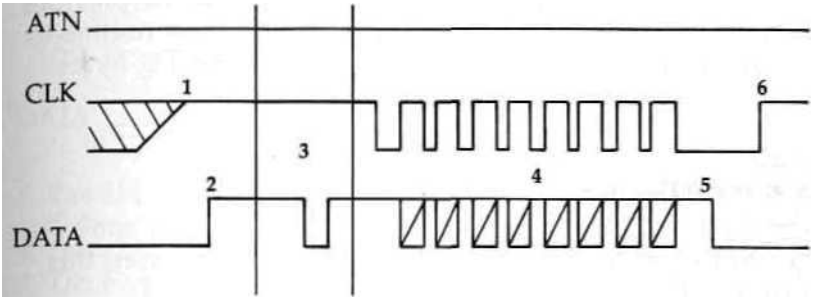
If a serial register interrupt has occurred on CIA #1, a byte has been received via the fast serial hardware. In this case, the byte is retrieved from the serial data register (56332/$DC0C) and stored in 164/$A4; then the fast serial flag (2588/$0AlC) is reset to 192/$C0.

The process for receiving a byte in standard (slow) serial mode is more complicated because it is handled in software. The routine waits for the data line to go high and then for the CLK line to go low. Then, when CLK goes high again, a bit is read from the DATA line and shifted into the working byte (164/$A4). The routine waits until CLK goes low before attempting to read the next bit, and the process is repeated for each of the eight bits of the byte.

After a byte has been received, the DATA line is pulled low to mark the end of the frame. If the EOI handshake has been performed, the CLK line is allowed to go high. The routine at 58783/$E59F is used to restore interrupts and the clock mode setting. The routine exits with the status register carry bit clear and with the received byte in the accumulator. The Y register is unused during the routine, and the X register value

is preserved. Figure 9-4 illustrates the process of receiving a data byte.

Figure 9-4. Receiving a Serial Data Byte



1. The 128 allows the CLK line to go high and waits for it to go high if the external device is holding it low.
2. Once the CLK line goes high, the 128 allows the DATA line to go high.
3. The 128 waits for the external device to respond by pulling the CLK line low. If this hasn't happened after a delay of approximately 260 microseconds, the 128 assumes that an EOI handshake is requested and briefly pulls the DATA line low. The external device must respond by pulling CLK low before another 260-microsecond delay expires; otherwise, a read-timeout error will occur,
4. Once the CLK line goes low, the 128 prepares to read data. To receive a bit in slow serial mode (illustrated above), the 128 waits until the CLK line goes high, reads the CIA port bit connected to the DATA input line, then waits for the CLK line to go low again. Fast serial mode works in a similar manner, except the transfer is managed by CIA chip hardware rather than ROM software, and the bits are clocked by pulses on the SRQ line rather than the CLK line.
5. After the byte is received, the 128 pulls the DATA line low to indicate the end of the byte frame.
6. If the EOI handshake is performed, the CLK line is allowed to go high after the byte is received.

58578 $E4D2 SECOND

Sends secondary address after LISTEN.
{This routine has a jump table entry at 65427/$FF93.)

Stores the secondary address value from the accumulator into the serial byte buffer at 149/$95, sends the buffered byte as a command (the ATN line should still be high from the previous LISTEN command), then falls through into the next routine to allow the ATN line to go high again so that following bytes will be seen as data instead of commands.

58583 $E4D7

Allows the serial bus ATN output line to go high.

Forces bit 3 of CIA #1 port A to %0. Since that bit is connected to the serial bus ATN output line via an inverter, this will set the ATN output line to a high state (+ 5 volts).

58592 $E4E0 TKSA

Sends secondary address **after TALK.**
(This routine has a jump table entry at 65430/$FF96.)

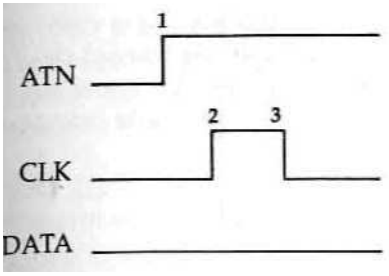
Stores the secondary address value from the accumulator into the serial byte buffer at 149/$95, then sends the buffered byte as **a** command (the ATN line should still be high from the previous TALK command). If the device is not present, the routine exits (after allowing the ATN and CLK lines to go high again). Otherwise, the routine falls through into the next one to make the 128 the listener and recognize the external device as the talker,

58601 $E4E9

Performs talk-listen **turnaround.**

Begins by calling the subroutine at 58739/SE573 to disable IRQ interrupts and standardize timing. The DATA line is held low, and the ATN line is allowed to go high (signaling the end of the command). The CLK line is then allowed to go high. The device which is commanded to talk should respond by pulling CLK low. The routine waits for this to happen, then jumps to the routine at 58783/$E59F to restore interrupts and the clock mode. Figure 9-5 illustrates the talk-listen turnaround.

Figure 9-5. Talk-Listen Turnaround



1. The 128 holds the DATA line low and allows the ATN line to go high.
2. The 128 allows the CLK line to go high.
3. The external device must respond by pulling CLK low, (The 128 will wait indefinitely for this to happen.) The external device is now the talker and the 128 is the listener.

58627 $E503 CIOUT

Sends a byte to a serial device.
(This routine has a jump table entry at 65448/$FFA8.)

Tests bit 7 of the serial buffer flag (148/S94). If the bit is %0, indicating that the one-byte serial buffer (149/$95) is empty, the byte in the accumulator is simply stored in the buffer and the flag bit is set to %1 before exiting. However, if the flag bit is already %1, a character is currently waiting in the buffer. In this case, the routine at 58252/$E38C is used to send the previously buffered character before the new character is added to the buffer. Carry will always be clear upon exit, and the byte value will still be in the accumulator. The X and Y register values are also preserved. The success of the operation can be determined from the value in the serial status flag (144/$90). The purpose of the buffering scheme is to make it possible to perform the EOI (end-or-identify) handshake with the final character of a file,

58645 $E515 UNTLK

Sends UNTALK command to a serial **device.**
(This routine has a jump table entry at 65451/$FFAB.)

Begins by calling the subroutine at 58739/$E573 to disable IRQ interrupts and standardize timing. The CLK line is pulled low; then ATN is pulled low as well to indicate that the byte

will be a command. The accumulator is loaded with the UNTALK command value, 95/$5F. Then a BIT opcode is used to fall through into the next routine to send the byte as a command. Unlike TALK, which affects only a specified serial device, UNTALK affects all serial devices. However, this shouldn't cause problems because the 128 serial bus allows only one active talker at any given time.

58662 8E526 UNLSN
Sends UNLISTEN command to a serial device.
(This routine has a jump table entry at 65454/SFFAE.)

Loads the accumulator with the UNLISTEN command value, 63/$3F, then clears bit 7 of the serial mode flag (2588/$0AlC) to disable fast serial mode, and sends the byte in the accumulator as a command on the serial bus. Afterward, the ATN line is allowed to go high (signaling the end of the command), and, after a short delay, the CLK and DATA lines are allowed to go high as well. Unlike LISTEN, which affects only a specified serial device, UNLISTEN affects all serial devices. However, this shouldn't cause problems because the 128 serial bus normally has only one listener at any given time.

58693 $E545
Allows serial bus CLK output line to go high.
Sets bit 4 of CIA #2 port A to %0. Since the output line for that port bit is connected to the serial bus CLK output line via an inverter, this will set the output line to a high state (+ 5 volts).

58702 $E54E
Pulls serial bus CLK output line low.
Sets bit 4 of CIA #2 port A to %1. Since the output line for that port bit is connected to the serial bus CLK output line via an inverter, this will set the output line to a low state (0 volts).

58711 $E557
Allows serial bus DATA output line to go high.
Sets bit 5 of CIA #2 port A to %0. Since the output line for that port bit is connected to the serial bus DATA output line via an inverter, this will set the output line to a high state (+ 5 volts).

58720 $E560

Pulls serial bus DATA output line low.
Sets bit 5 of CIA #2 port A to %1. Since the output line for that port bit is connected to the serial bus DATA output line via an inverter, this will set the output line to a low state (0 volts).

58729 $E569

Reads the serial bus DATA and CLK input lines.
Reads the value at port A of CIA #2, then shifts the value of bit 7, connected to the serial bus DATA input line, into the status register carry bit. This will also shift the value of bit 6, connected to the CLK input line, into bit 7. Upon return, the DATA bit can be tested with BCC/BCS or shifted into a working byte, and the CLK bit can be tested with BPL/BMI.

58739 $E573

Disables IRQ interrupts and standardizes timing during I/O operations.

Begins by disabling IRQ interrupts. Next (at 58740/$E574), the custom mode flag (2618/S0A3A) is checked. If bit 7 of this flag is set to %1, the routine exits without changing the clock mode setting or disabling sprites. The 128 sets this flag to zero during the IOINIT routine (part of both the reset and RUN/STOP-RESTORE sequences) and does not normally change the value. You can set bit 7 of the flag if for some reason you want to retain fast clock mode or sprites during an I/O operation. The clock mode storage flag (2615/$0A37) is then checked. The routine exits if bit 7 of this flag is set to %1, indicating that a clock mode has already been stored (bits 2-7 of $D030 are always %1). Otherwise, the value in the VIOII register at 53296/SD030, which determines the system clock frequency, is stored in the clock mode flag, and the value in the register at 53269/$D015, which determines which sprites are currently enabled, is stored in 2616/$0A38. Both registers are then reset to zero, which disables all sprites and sets the system to normal (1 MHz) clock speed. This is done to standardize system timing. Tape and serial data transfers rely on software delay loops for critical timing functions, so the system must be in a standard mode for the loops to provide the correct delay. If any sprites have previously been enabled, a delay loop is executed before the routine exits.

58783 $E59F

Reenables interrupts and restores clock mode after I/O operations.

Begins by checking the custom mode flag (2618/$0A3A). If bit 7 of this flag is set to %1, the routine skips ahead to reenable interrupts and exit. The 128 sets this flag to zero during the IOINIT routine (part of both the reset and RUN/STOP-RESTORE sequences) and does not normally change the value. You can set bit 7 of the flag if for some reason you want the clock mode or sprite enable setting preserved after an I/O operation. The clock mode storage flag (2615/$0A37) is then checked. If bit 7 of this flag is set to %0, indicating that no clock mode is stored, the routine skips ahead to reenable interrupts and exit. Otherwise, the value in 2616/$0A38 is restored to the VIC-II sprite enable register (53269/$D015), and the value in 2615/$0A37 is restored to the clock mode register (53296/$D030). The clock mode flag is then reset to zero. IRQ interrupts are reenabled before exiting.

58812 $E5BC

Performs fast serial turnaround.

Waits for a serial register interrupt on CIA #1, indicating that the current output byte has been completely sent, then falls through into the next routine to reset the serial port lines for fast serial input. The port is left set up for input so that incoming fast serial communications can be detected automatically.

58819 $E5C3 SPIN

Sets serial device for fast serial input.

Sets the serial line from CIA #1 for input and halts timer A on CIA #1; then clears bit 3 of the MMU register at 54533/SD505 to set the serial port's fast communications hardware for input.

58838 8E5D6 SPOUT

Sets serial device for fast serial output.

Sets bit 3 of the MMU register at 54533/$D505 to set the serial port's fast communications hardware for output; then clears all CIA #1 interrupts. Timer A is loaded with 4/$0004, the timing constant for fast serial output bits. The serial line for CIA #1 is set for output, and timer A is started. Bytes subsequently stored in the serial data register of CIA #1 will be sent via the fast serial hardware.

58875 SE5FB SPIN_SPOUT

Sets serial device for fast serial input or output.
Branches to one of the fast serial setup routines, depending on the setting of the status register carry bit. If carry is clear, the routine at 58819/$E5C3 is used to set up the port for fast serial input. If carry is set, the routine at 58838/$E5D6 is used to set up the port for fast serial output.

58879 $E5FF

prepares next bit for RS-232 transmission.
(Called by the NMI handling routine when a timer A interrupt occurs.)
If the count of bits remaining to be sent is zero, indicating that all stop bits for the current byte have been sent, a branch is taken to 58954/$E64A to prepare the next byte for transmission. If bit 7 of the count is set, a branch is taken to 58948/$E644 to prepare to send a stop bit. Otherwise, the next bit to be sent is pulled from the data byte storage (182/$B6) into the carry bit, and the parity flag (189/$BD) is updated accordingly. The count of remaining bits is decremented. If the result is zero, a branch is taken to 58907/SE61B to prepare parity or stop bits. Finally, bit 2 of 181/SB5 is set to the bit value to be sent.

58907 $E61B

Prepares parity and stop bits.
Checks bit 5 of the RS-232 command register (2577/$0A11) to determine whether a parity bit is to be sent. If not (if the bit is %0), the routine skips ahead to determine the number of stop bits. Otherwise, a parity bit is prepared. Bits 6 and 7 of the command register determine the parity type. These are possible types:

Bits 7 6	Parity type
0 0	Odd
0 1	Even
1 0	Mark
1 1	Space

If odd parity is specified, the parity flag (189/$BD) is tested. When the flag is nonzero, indicating that an odd number of %1 bits has already been sent in the current byte, the routine prepares a parity bit of %0. When the number of %1 bits already sent is even, a parity bit of %1 is prepared to

make the total odd. If even parity is specified, a parity bit of %0 will be prepared when the parity flag indicates that the number of %1 bits already sent is even, and a parity bit of %1 will be prepared when the number of %1 bits sent is odd so that the total number of %1 bits sent (including the parity bit) will always be even. Mark and space parity are simpler: In the former case, the parity bit is always %1; in the latter, it's always %0. (Early versions of the Commodore 64 Kernal incorrectly computed even and odd parity. All types of parity are handled properly in the 128 Kernal and in the version of the Commodore 64 Kernal used for 64 mode in the 128.)

Next, the routine prepares for the transmission of either one or two stop bits, depending on the setting of bit 7 of the RS-232 control register (2576/$0A10). If one stop bit is specified (if the register bit is %0), the count of bits remaining to be sent (180/$B4) is decremented once {to 255/$FF). If two stop bits are sent (if the register bit is %1), the count is decremented twice (to 254/$FE). The routine ends by branching back into the previous routine to set the prepared parity or stop bit as the next bit to send.

58948 $E644

Prepares to send a stop bit.

Increments the count of bits remaining to be sent (180/$B4), then prepares a %1 bit for transmission (stop bits are always %1).

58954 SE64A

Prepares to transmit next byte.

Checks bit 0 of the RS-232 command register (2577/$0All) to determine which handshaking mode is in use. If x-line handshaking is specified (if the register bit is %1), the RS-232 DSR and CTS lines (pins K and L of the user port) are tested. If the external device is not holding these lines high, the corresponding bit in the RS-232 status flag (2580/$0A14) is set—bit 6 for DSR missing or bit 4 for CTS missing—then the routine disables timer A interrupts to halt transmission.

For three-line handshaking, or for x-line handshaking when DSR and CTS are held high, the parity flag (189/SBD) is cleared and the current bit flag (181/$B5) is set to 0/$00 (start bits are always %0). The count of bits to send (180/$B4)

is loaded from 2581/S0A15. Tf the RS-232 output buffer is empty, the routine disables timer A interrupts and exits. Otherwise, the next available byte from the output buffer is loaded into 182/$B6, and the pointer to the head of the buffer (2586/S0A1A) is incremented,

59007 $E67F

Sets CIA interrupt register and RS-232 activity flag.

Sets or clears bits in the interrupt control register for CIA #2, depending on the value in the accumulator. If bit 7 of the accumulator value is %0, the interrupt register bits corresponding to the %1 bits in the accumulator value will be cleared. If bit 7 is %1, the interrupt register bits corresponding to any other %1 bits in the accumulator value will be set. The RS-232 activity flag (2575/$0A0F) is then updated to reflect the new setting of the interrupt register.

59022 $E68E

Computes bit count for the RS-232 operation.

Computes a bit count based on the setting of bits 5 and 6 of the RS-232 control register (2576/$0A10). The count value, which will be one greater than the number of data bits specified, will be returned in the X register.

Bits 6 5	Data bits	Bit count
0 0	8	9
0 1	7	8
1 0	6	7
1 1	5	6

59037 $E69D

Processes received bits.

Checks the start bit flag (169/$A9) to determine whether the start bit for a byte has been read yet. If not, a branch is taken to 59092/$E6D4 to see whether this is a start bit. Otherwise, the received bit count (168/SA8) is decremented. If the count has reached zero, all bits for a byte have been received, so a branch is taken to 59103/$E6DF to process the byte. Otherwise, the parity indicator is toggled, and the received bit in 167/$A7 is shifted into the work byte (170/$AA).

59058 $E6B2

Tests for stop bit.

Decrements the received bit count and checks whether the received bit is a %1 (stop bits are always %1). If it is, bit 7 of the RS-232 control register (2576/$0A10) is tested. If one stop bit is specified (if the register bit is %0), the routine falls through into the next one to prepare to receive the next byte. If two stop bits have been specified, the routine exits to look for another stop bit.

If a stop bit has not been received, the routine branches to set a bit in the serial status flag (2580/$0A14) according to the previously received byte (170/SAA): bit 1—the framing error bit—if the previously received byte is nonzero, or bit 7—the break error bit—if the previously received byte is zero (indicating that the received data line is being held low).

59074 $E6C2

Prepares to receive next byte.

Enables FLAG interrupts for CIA #1, then updates the RS-232 activity flag (2575/$0A0F) to indicate that FLAG interrupts are active. The nonzero flag value will also be stored in the start bit flag (169/$A9) to indicate that no start bit has been received. Timer B interrupts are then disabled, and the activity flag is updated to reflect this. (Timer B interrupts, used to time incoming bits, are reenabled after the FLAG interrupt occurs,)

59092 SE6D4

Tests for start bit.

Tests the received bit (in 16 7/$A 7). If it is not %0, it is not a start bit, so a branch is taken to 59074/$E6B2 to look for another byte. Otherwise, the zero value is stored in the start bit flag (169/$A9) to indicate that a start bit has been received, and the parity indicator flag (171/$AB) is initialized to 1/S01.

59103 $E6DF

Stores received character in buffer and checks parity.

Checks whether space for an additional character is available in the input buffer. If not, the routine branches to set the receiver buffer overflow bit (bit 2) in the RS-232 status flag (2580/$0A14) and to prepare for the reception of the next byte, (The received character is lost in this case.) Otherwise,

the received character is padded with %0 bits if it is less than eight bits long and then stored at the current tail of the input buffer. If no parity is used, a branch is taken to 59058/$E6B2 to check for a stop bit. Otherwise, the current bit received (167/$A 7) is taken to be a parity bit and is compared against the calculated parity for the byte. If the two do not correspond to the specified parity type, the parity error bit (bit 0) of the status flag will be set, and the routine will jump to reset for the reception of the next byte.

59177 SE729

Handles CKOUT for RS-232 device.

Sets the device number in the accumulator as the current output device (154/$9A), then tests bit 0 of the RS-232 command register (2577/$0All). If the bit is %0, three-line handshaking has been specified, so the routine exits at this point. For x-line handshaking, the routine tests the DSR line (pin L of the user port). If the external device is not holding this line high, a branch is taken to set bit 6 of the RS-232 status flag (2580/$0A14) and exit with the status register carry bit clear. If DSR is high, the state of the RTS line (pin D of the user port) is checked. If the 128 previously set this line high, the routine exits with carry clear. Otherwise, the routine waits until any current transmission is completed, then waits for the external device to pull the CTS line (pin K of the user port) low. The routine then sets the RTS line high to signal that it is ready to send a byte and waits until an external device sets the CTS line high to acknowledge that it is ready. (If DSR goes low while the routine is waiting, bit 6 of the status flag will be set). The routine then exits with carry clear.

59228 $E75C

Handles BSOUT for RS-232 device.

(The normal entry point for this routine is 59231/SE75F.)

Checks whether the output buffer is currently full. If no space is available, the routine loops to enable interrupts for RS-232 transmission and waits until space becomes available in the buffer. The buffer tail pointer (2587/$0AlB) is incremented, and the value in the accumulator is placed at the tail of the output buffer. If bit 0 of the RS-232 activity flag (2575/$0A0F) is set to %1, timer A interrupts are already enabled, so the routine exits at this point. Otherwise, timer A is loaded with

the bit timing constant value in 2582-2583/$0A16-$0A17; then timer A interrupts are enabled, and the activity flag is updated to reflect this. The routine at 58954/$E64A is called to prepare to transmit the byte; then timer A is started.

59285 $E795

Handles CHKIN **for** RS-232 device.

Sets the device number in the accumulator as the current input device (153/S99), then tests bit 0 of the RS-232 command register (2577/lOAll). If the bit is %0, three-line handshaking has been specified, so the routine skips ahead to test whether interrupts are enabled. For x-Iine handshaking, the routine tests bit 4 of the command register to determine the duplex mode in use. For full duplex (bit 4 is %0), the routine skips ahead to test whether interrupts are available. For half duplex, the RS-232 DSR line (pin L of the user port) is tested. If the external device is not holding this line high, a branch is taken to set bit 6 of the RS-232 status flag (2580/$0A14) and exit with the status register carry bit clear. If DSR is high, the state of the RTS line (pin D of the user port) is checked. If the 128 is currently holding this line high, the routine exits with carry clear, then pulls the CTS line (pin K of the user port) low. The routine then waits for DTR line to go high, after which FLAG interrupts are enabled to detect the start bit.

The final step of the routine is to test whether FLAG or timer B interrupts are enabled. If neither is enabled, FLAG interrupts are enabled. The routine exits with carry clear.

59342 $E7CE

Handles GETIN for RS-232 device-

Checks whether any characters are available in the input buffer. If so, bit 3 of the RS-232 status flag (2580/$0A14) is cleared, the buffer head pointer (2585/$0A19) is incremented, and the character from the buffer is returned in the accumulator. If no characters are available, bit 3 of the status flag is set to %1, and the value 0/$00 is returned in the accumulator. The carry bit will be set in this case.

59372 $E7EC

Disables RS-232 activity during tape or serial bus operations.

Exits immediately if the RS-232 activity flag (2575/$0A0F) contains the value 0/$00, indicating that no RS-232 operations

are being used. Otherwise, the routine waits until the transmission or reception of the current byte is completed, then disables FLAG interrupts so that no further bytes can be received and clears the activity flag so that no more bytes will be sent. The contents of the accumulator are preserved during this routine.

59397 SE805

Handles NMI interrupts for RS-232.

Controls the transmission and reception of data through the RS-232 port. Three CIA #2 interrupt sources are used in RS-232 communications. Timer A is used to establish the duration of bits being transmitted. The FLAG line, which triggers an interrupt when it detects a high-to-low transition, is used to initiate the reception of a byte when an incoming start bit is detected. Timer B is used to time the reception of subsequent bits.

The routine begins by comparing the CIA #2 interrupt register value at the time of the NMI interrupt (in the Y register when the routine is called by the main NMI handler at 65285/$FF05) against the value in the RS-232 activity flag (2575/$0A0F), If bit 0 is set in both, a valid timer A interrupt has occurred to indicate that it is time to send the next bit. So, the bit value (in bit 2 of 181/$B5) is written to bit 2 of CIA #2 port A, which is connected to the transmitted data line (pin M of the user port). RS-232 interrupts are then reenabled by writing the activity flag contents to the CIA interrupt register (56589/$DD0D). In addition, the routine checks whether bit 1 or 4 is set in both the interrupt register value and the activity flag, indicating that a valid FLAG or timer B interrupt occurred concurrently with a timer A interrupt (RS-232 devices must be capable of simultaneous transmission and reception). If neither has occurred, the routine skips ahead to prepare the next bit for transmission. For timer B interrupts, the subroutine at 59512/SE878 is called to read a bit. For FLAG interrupts, the subroutine at 59561/$EA81 is called to start reception of a byte. The subroutine at 58879/$E5FF is then called to prepare the next bit for transmission. CIA interrupt sources are again enabled before exiting.

If no timer A interrupt occurred, the routine checks whether bit 1 is set in both the interrupt register value and the activity flag, indicating that a valid timer B interrupt occurred. If so, the subroutine at 59512/$E878 is called to read a bit.

Otherwise, a test is made of bit 4 in both the interrupt register value and the activity flag. If the bit is set in both, a valid FLAG interrupt has occurred, so the subroutine at 59561/ $E8A9 is called to start reception of a byte. In either case, the activity flag value is stored in the CIA interrupt register to reenable RS-232 interrupts before exiting.

59472 $E850

Table of baud rate timing constants for NTSC systems.

The ten two-byte values in this table (in low-byte/high-byte order) are the CIA timer settings used to transmit and receive bits at the ten standard baud rates when the 128 is operating with NTSC clock frequency (1.02273 MHz). This is the formula for table values:

value = 1.02273E6 / (2 * baud rate) - 100

59492 $E864

Table of baud rate timing constants for PAL systems.

The ten two-byte values in this table (in low-byte/high-byte order) are the CIA timer settings used to transmit and receive bits at the ten standard baud rates when the 128 is operating with PAL clock frequency (0.985265 MHz). This is the formula for table values:

value = 0.985265E6 / (2 * baud rate) - 100

59512 $E878

Reads a bit from RS-232 device.
(Called by the NMI handling routine when a rimer B interrupt occurs.)

Reads the current status of the R5-232 received data line (pin C of the user port) and stores the bit value in 167/$A7. The interrupt time for the next bit is calculated and stored in the timer B latch (56582-56583/$DD06-$DD07); then timer B is restarted. CIA #2 interrupts are reestablished by storing the RS-232 activity flag (2575/$0A0F) in the interrupt register (56589/$DD0D), and the timer latch is reloaded with $FFFF. The routine ends by jumping to 59037/$E69D to process the received bit.

59561 $E8A9

Initiates reception of RS-232 byte.
(Called by the NMI handling routine when a FLAG interrupt occurs.)
Begins by copying the bit timing constant value (2578-2579/$0A12-$0A13) into the latch for timer B (56582-56583/$DD06-$DD07) and starting timer B. Then, FLAG interrupts are disabled and timer B interrupts are enabled. The RS-232 activity flag (2575/$0A0F) is updated to reflect the change. The latch for timer B is loaded with $FFFF so that the timer will count continuously after it counts down for the next bit. Finally, the count of bits to be received for the current character is loaded from 2581/$0A15 into the working counter (168/SA8).

59600 SE8D0

Reads next header block from tape.
Calls the subroutine at 59890/$E9F2 to fill the cassette buffer with the next block from tape, exiting with the status register carry bit set if the RUN/STOP key is pressed while the block is being loaded.

The first byte in the header block, the type identifier, is then examined. If the identifier value is 5, this is an end-of-tape marker, so the routine branches to exit with carry set (in this case, the Y register will hold the value 255/$FF). If the identifier value is something other than 1, 3, or 4, the routine loops back to read another block. If Kernal messages are allowed, FOUND is displayed, followed by 16 filename characters from the buffer. (The buffer is filled with space characters when the header is written, so the displayed name will be padded with spaces if it is fewer than 16 characters long.) A delay loop lasting for two increments of the middle byte of the jiffy clock (161/$A1), about 8-1/2 seconds, is then started. If the space key is pressed during this delay, the routine loops back to the beginning to read another header. If any other key in the same column (for example, Commodore or CONTROL) is pressed, the loop is terminated. At the end of the loop, the routine exits with carry clear. The X register will hold the type identifier value for the header.

59673 $E919

Writes a header block to tape.

Stores the type identifier value from the accumulator in 158/ $9E, then checks the address of the cassette buffer, exiting immediately if it is less than 512/$0200. Otherwise, the contents of the starting address pointer (193-194/ $C1-$C2) and ending address pointer (174-175/$AE-$AF) for the current operation are stored on the stack, and the cassette buffer is filled with space characters (32/$20). The type identifier value is placed in the first byte of the buffer, and the starting address and ending address values are placed in the next four bytes (each in standard low-byte/high-byte order). The characters, if any, of the current filename are then copied into the buffer following the filename (the filename can fill the remainder of the buffer, up to 187 characters). The buffer starring and ending addresses are then set as the operation starting and ending addresses; the leader flag (171/$AB) is loaded with 105/$69 for a long interfile leader; and the subroutine at 59932/$EAlC is called to write the buffer contents to tape as a header. The original starting and ending address pointer values are then restored from the stack. Carry will be clear upon exit unless the RUN/STOP key is pressed while the header is being written.

59776 8E980

Loads and tests cassette buffer address.

Loads the address value in the cassette buffer pointer (178-179/ $B2-$B3) into the X and Y registers; then compares the high byte (in the Y register) with the value 2/$02 to test whether the buffer address is greater than 511/$01FR Upon exit, the status register carry and Z bits will reflect the result of the comparison.

59783 $E987

Sets buffer address as block address

Loads the address of the cassette buffer into the pointer to the starting address of the block to be read or written (193-194/ $C1-$C2) and the ending address of the buffer (192 bytes beyond the starting address) into the pointer to the end of the block (174-175/$AE-$AF).

59802 SE99A

Searches for a specified header.
Calls the routine at 59600/$E8D0 to load the next header from tape into the cassette buffer; exits if carry is set upon return from that routine (indicating that the RUN/STOP key has been pressed or that an end-of-tape header has been read). Characters from the current filename (pointed to by 187-188/ $BB-$BC in the bank specified in 199/SC7) are compared against those in the buffer. If all characters match up to the end of the current filename, the names are considered matching, and the routine exits with carry clear, regardless of how many characters may remain untested in the buffer (there's nothing to indicate the length of the name read from tape). However, if a mismatch is found, the routine loops back to search for another filename.

59838 $E9BE

Checks for cassette buffer filled or emptied.
Calls the routine at 59776/$E980 to get the high byte of the cassette buffer address into the X register. The buffer index (166/$A6) is then incremented, loaded into the Y register, and compared against the value 192/$C0, the maximum count of characters in the buffer. Upon exit, the status register Z and carry bits will reflect the result of the comparison (both will be set if the end of the buffer is reached),

S9848 $E9C8

Requests PLAY button if necessary.
Checks whether a tape button is currently pressed; exits with the status register carry and Z bits set if any buttons are already pressed. Otherwise, if Kernal messages are allowed, PRESS PLAY ON TAPE is displayed. The routine then waits for a tape button to be pressed. If the RUN/STOP key is pressed in the meantime, the routine exits with the status register carry bit set. When a button is pressed, the routine displays OK (if messages are allowed) and exits.

59871 8E9DF

Checks tape buttons.
Tests the setting of bit 4 of the processor on-chip I/O port (l/$01), connected to the cassette button sense line. The status

register Z bit will be clear if no button is pressed or will be set if any button is detected.

59881 $E9E9

Requests RECORD and PLAY buttons if necessary.

Checks whether a tape button is currently pressed; exits with the status register carry and Z bits set if any buttons are already pressed. Otherwise, if Kernal messages are allowed, PRESS RECORD & PLAY ON TAPE is displayed. The routine then waits for a tape button to be pressed. If the RUN/STOP key is pressed in the meantime, the routine exits with the status register carry bit set. When a button is pressed, OK is displayed (if messages are allowed) and the routine exits.

59890 $E9F2

Reads next header or data block from tape.

Clears the tape status flag (144/$90) and operation flag (147/$93) to zero; then sets the starting and ending addresses of the cassette buffer as the starting and ending addresses for the current operation and falls through into the next routine.

59899 $E9FB

Reads or verifies a block from tape.

Calls the subroutine at 59848/$E9C8 to request that the PLAY button be pressed and exits (with carry set) if the RUN/STOP key is pressed during that subroutine. IRQ interrupts are disabled, and a series of variables and counters are initialized. The accumulator is loaded with 144/$90 (the value to enable FLAG interrupts), and the X register is loaded with 14/$0E (the offset for read interrupts), and the routine branches to the main tape I/O routine (59942/SEA26).

59925 $EA15

Writes a header or data block to tape.

Calls the subroutine at 59783/$E987 to set the buffer addresses as the starting and ending addresses for the operation-then falls through into the next routine to write the buffer block to tape.

!

59928 $EA18

Writes a block to tape.

Loads the leader flag (171/SAB) with 20/$14 to specify a short leader between the header and the block; then calls the subroutine at 59881/$E9E9 to request that the PLAY and RECORD buttons be pressed. (The routine will exit with carry set if the RUN/STOP key is pressed during the subroutine.) IRQ interrupts are disabled and the accumulator is loaded with 130/$82 (the value to enable timer B interrupts), and the X register is loaded with 8/$08 (the offset to write leader bits). The routine then falls through into the next one to perform the operation.

59942 $EA26

Initiates tape I/O operation.

Begins by disabling all VIC-II interrupt sources and clearing any pending VIC-II interrupts. The value in the accumulator upon entry is loaded into the interrupt control register for CIA #1 (56333/$DC0D), and timer B is started. Tape operations depend on precise timing, so all other activities that affect system timing are disabled: RS-232 interrupts are disabled, the 40-column screen is blanked, and the subroutine at 58740/ $E574 is called to switch to standard (1 MHz) clock mode and disable sprites. The current address in the IIRQ vector at 788-789/ $0314-$0315 is preserved in 2569-2570/$0A09-$0A0A; then the subroutine at 61083/$EE9B is called to load a new value into IIRQ, according to the value in the X register. This new IRQ service routine will be responsible for reading or writing data to tape. The count of blocks to be read or written (190/ $BE) is initialized to 2 (blocks are always read or written in pairs). The subroutine at 60762/$ED5A is called to initialize variables; then the tape motor is started, the interlock location (192/$C0) is set to keep it on, and a delay loop is executed to allow the motor to get up to normal speed. IRQ interrupts are then enabled to begin the reading or writing process.

Since the actual tape operations are performed during IRQ interrupts, the routine must now wait in a loop for the operation to be completed. It continually tests the IRQ storage flag (2570/S0A0A), waiting for the IIRQ vector to be reloaded with the address stored there, which will happen after the IRQ-driven tape routines are finished. In the meantime, the RUN/ STOP key is also tested. If that key is pressed, the operation is

halted and the routine exits with carry set. Otherwise, when the original IIRQ address is restored at the end of the operation, the vector storage flag will be reset to zero, and the routine will exit with carry clear.

60047 $EA8F

Checks for RUN/STOP keypress during tape operations.

Calls the Kernal STOP routine [$FFE1] to determine whether the RUN/STOP key has been pressed. If it has been, the cassette motor is stopped, the default IIRQ vector address is restored, the return address of the calling routine is removed from the stack, and the IRQ vector storage flag (2570/$0A0A) is cleared.

60065 $EAA1

Sets timer A to check for FLAG interrupts.

The reception of tape dipole is normally initiated by a FLAG interrupt from CIA #1, which is triggered by the low-to-high transition on the cassette read line at the start of a dipole. Timer A of CIA #1 is loaded with a timing value for the type of dipole being read, and it's used to check whether too much time elapses between FLAG interrupts (which should be equally spaced) in an attempt to determine whether any dipoles might have been missed. If no FLAG interrupt occurs before the timer counts down to zero, the timer will trigger an interrupt. This prevents the 128 from waiting indefinitely for a FLAG interrupt.

60139 **$EAEB**

Reads or verifies a block of data from tape.
{This is a tape IRQ service routine.)

Reads magnetic patterns (known as *dipoles)* from tape, assembles them into bytes, and loads the bytes into memory (or compares the bytes against memory) until a specified ending address is reached. The routine for reading from tape is the longest and most complex one in the Kernal, and will not be discussed in detail here. For a thorough description of the process, refer to *COMPUTERS VIC-20 and Commodore 64 Tool Kit: Kernal,* by Dan Heeb.

Briefly, the routine reads dipoles from tape and determines whether they represent leaders, word markers, or data bytes. The routine does not demand that the dipoles have an absolutely exact duration, but rather it employs a concept known as an adjustable baseline to determine whether the dipole is within an acceptable range for a particular type. This makes it possible to compensate for minor variations in the motor speed of different Datassette units. Because two complete copies of the data block are recorded, error correction is possible. If an error is detected while a byte is being read from the first block, the address of the byte which could not be read is recorded in page 1. Up to 31 error addresses can be recorded in 256-317/$0100-$013D. When the second block is read, the bytes can be corrected if no error is encountered when the byte is read from that block.

When a byte is successfully read from tape, the handling of the byte depends on the value in the operation flag (147/$93). If the flag value is zero, the byte is stored in the location pointed to by 172-173/$AC-$AD in the bank specified in 198/$C6. If the value is nonzero, the byte is compared against the contents of the location pointed to by 172-173/$AC-$AD in the bank specified in 198/$C6 (a verify operation).

Errors which cannot be corrected are recorded in the tape status flag (144/S90). If the end of a block is reached before the specified ending address (a short-block error), bit 2 of the flag is set to %1. If more than 31 errors occur while the first block is being read, or if an error recorded during the first block cannot be connected during the second block, an unrecoverable-read error occurs, and bit 4 of the flag is set to %1. That bit is also set during a verify operation if the byte in memory doesn't match the corresponding byte in either block on tape. If the byte read from tape as a checksum doesn't match the calculated checksum for the bytes previously read from tape, a checksum error occurs, and bit 5 of the flag is set to %1.

60753 $ED51

Loads working pointer with starting address.

Transfers the starting address of the current block (193-194/$C1-$C2) to the tape working pointer (172-173/$AC-$AD).

60762 $ED5A

Initializes tape variables between each byte.
Resets the count of bits for the next byte (163/$A3) to 8; then clears the dipole flag (164/$A4), the word marker half-dipole flag (168/$A8), the parity work byte (155/$9B), and the word marker flag (169/$A9).

60777 $ED69

Initiates writing of a tape half-dipole.
Reads the bit value for the current half-dipole to be sent (the current value of the rightmost bit in 189/$BD) and loads CIA #1 timer B with the appropriate value: $0060 for a short half-dipole if the bit is %0 or $00B0 for a long half-dipole if the bit is %1. The routine then clears the CIA interrupt register, starts timer B, and toggles the cassette write line (bit 3 of the 8502's on-chip I/O port at location l/$01) to begin writing the current half-dipole. Upon exit, the status register 2 bit will be set if the line is currently low or will be clear if the line is currently high.

60816 6ED90

Writes a block of data to tape.
(This is a tape IRQ service routine.)

Writes bytes of data from memory to tape until the specified ending address is reached. The system used for representing the bytes on tape is rather complex. Each bit of a byte is represented by a magnetic pattern called a *dipole,* which is generated by holding the cassette write line (bit 3 of the 8502's on-chip I/O port at location l/$01) high for a period, then low for a different period. The duration of the periods is determined by the value loaded into CIA #1 timer B, which controls the amount of time between IRQ interrupts for tape. The routine at 60777/$ED69 actually writes each half of the dipole. A %0 bit is represented by a short half-dipole followed by a long one, while a %1 bit is represented by a long half-dipole followed by a short one. Each byte is preceded by a word marker dipole, which consists of an extra-long word marker half-dipole followed by a long half-dipole. Each byte is followed by a parity bit dipole. The parity bit will be either %0 or %1, as necessary to provide an odd total number of %1 bits in the byte and parity bit combined. The routine writes two complete copies of the data block, separated by a short

leader. As each byte is written, it is also excIusive-ORed with a checksum work byte. This checksum byte is written to tape following the second copy of the block to provide an additional error check. For a more thorough description of the process, refer to *COMPUTE'S VIC-20 and Commodore 64 Tool Kit: Kernal,* by Dan Heeb.

60974 $EE2E

Writes a leader to tape and prepares to write a data block. (This is a tape IRQ service routine.)

Writes leader dipoles to tape until the count specified in 171/$AB is decremented to zero, about 9.5 seconds for the leader before a header, 1.9 seconds for the leader between a header and the first data block, or 0.045 seconds between data blocks. The IIRQ vector is then loaded with the address of the routine to write the data block [$ED90], but if both blocks have been written, the routine branches to 61077/$EE95 to restore normal IRQs and exit from the operation. Otherwise, the starting address for the block to be written is loaded into the working pointer (173-174/$AC-$AD), and a branch is taken into the block write routine to write the block countdown characters. Each block is preceded by a countdown character pattern—8 7 6 5 4 3 2 1 0—to mark the end of the leader and the beginning of data. (For the second block of a pair, bit 7 of the countdown character codes will be set to %1.)

61015 6EE57

Restores IRQ vector and operating modes after tape operation. Reenables the screen (it is normally blanked during tape operations), then checks the custom mode flag (2618/$0A3A). If the flag has bit 7 set, indicating that a special operation mode has been specified, the routine skips ahead to turn off the cassette motor and restore the IRQ vector. Otherwise, the VIC-II chip sprite enable register is reloaded with the value stored in 2616/$0A38 at the start of the operation. The clock mode register is reloaded with its original value from 2615/$0A37; then the clock mode storage location is reset to zero to indicate that no value is stored. The routine to turn off the cassette motor [SEEBO] is called; then raster interrupts are reenabled, and the IIRQ vector (788-789/$0314-$0315) is reloaded with its original address, stored in 2569-2570/$0AO9-$0A0A at the beginning of the operation.

61077 6EE95

Ends tape write interrupts.

Calls the subroutine at 61O15/$EE57 to restore the IRQ interrupt vector address and system operating modes to their original values; then exits from the interrupt to return to normal processing.

61083 $EE9B

Loads IIRQ vector for tape operation.

Loads the IIRQ vector (788-789/$0314/$0315) with a value from the table at 61096/$EEA8, depending on the offset specified in the X register:

Offset	Vector address	Function
8/$08	60974/$EE2E	Write a leader to tape
10/$0A	60816/$ED90	Write a block to tape
12/$0C	64101/$FA65	Restore normal IRQ functions
14/$0E	60139/$EAEB	Read a block from tape

61104 $EEB0

Turns cassette motor off-

Sets bit 5 of the processor I/O port (location l/$01) to %1, which has the effect of turning off power to the cassette motor (pins 3 and C of the cassette port).

61111 $EEB7

Tests whether ending address has been reached.

Compares the value in the working pointer for tape and serial operations (172-173/$AC-$AD) against the ending address in 174-175/$AE-$AF. Upon exit, the status register carry and Z bits will reflect the result of the comparison.

61121 $EEC1

Increments the working pointer-

Increments the address in 172-173/$AC-$AD.

61128 $EEC8

Handles FLAG interrupts for tape.

Insures that the B bit in the status register value on the stack is clear, then jumps to the IRQ interrupt handler [$FF17].

61136 $EED0

Controls tape motor interlock.

Checks bit 4 of the processor I/O port (location l/$01) to determine whether any buttons are pressed on the Datassette. If no button is pressed (indicated when the bit is %1), the interlock byte (192/$C0) is cleared and the tape drive motor is turned off by setting bit 5 of the processor I/O port to %1. If a button is pressed (if bit 4 is %0), the interlock byte is checked. If that byte already contains a nonzero value, the routine exits. Otherwise, the tape motor is turned on by setting bit 5 of the processor I/O port to %0.

This routine is part of the normal IRQ sequence, so it's not possible to control the tape motor by directly changing bit 5 of the processor I/O port. If you turn on the motor while no buttons are pressed, it will be turned off again during the next IRQ interrupt. If you turn off the motor while a button is pressed, it will be turned on again during the next IRQ interrupt (unless the interlock byte contains a nonzero value).

61163 $EEEB GETIN

Retrieves a byte from the current input device.

(This routine is the normal target of the jump table entry at 65508/$FFE4, via the indirect vector at 810/$032A.)

Checks whether the current input device (153/$99) is the keyboard (device 0) or RS-232 (device 2). If it is neither of these, the routine branches to 61205/SEF15 in the BASIN routine to accept a byte from the specified device. For keyboard and RS-232, the significant difference between GETIN and BASIN is that GETIN returns the value 0/$00 if no character is available, whereas BASIN will wait until a valid character becomes available.

For keyboard input, the routine checks the number of characters available in the keyboard buffer (208/$D0) and the number of characters available from the current programmable key string (2O9/$D1). If both are zero (no characters available), the routine exits with 0/$00 in the accumulator and carry clear. However, if characters are available, the screen routine at 49158/SC006 is used to retrieve the next available character and return it in the accumulator. Neither the X nor the Y register values are preserved in this case.

For RS-232 input, the routine at 59342/SE7CE is used to retrieve a byte from the RS-232 input buffer and return it in

the accumulator. If the buffer is empty, the accumulator will contain 0/$00, and bit 3 of the RS-232 status flag (2580/$0A14) will be set to %1. For RS-232 input, the X register is unused, and the Y register value is preserved.

61190 $EF06 BASIN

Accepts a byte from the current input device.
(This routine is the normal target of the jump table entry at 65487/$FFCF, via the indirect vector at 804/$0324.)
If the current input device (153/$99) is the keyboard (device 0), the routine sets the current cursor row and column (232/$E8 and 233/$E9) as the starting row and column for input (235/SEB and 236/$EC); then it calls the screen editor routine to retrieve a character from a keyboard input line [$C009], For input from the screen (device 3), the routine stores the device number in the input source flag (214/$D6) and sets the current right window margin (231/$E7) as the ending column for input (234/$EA); then it calls the screen editor routine to retrieve a character from the screen line [$C009], Unfortunately, this is not the proper setup procedure for screen input, so BASIN from the screen does not work in the current 128 Kernal. For screen input, the ending row for input (2608/$0A30) must be specified, and the setting of bit 7 of the input source flag must be preserved so that the screen editor routine will know when the end of the input line is reached. See the entry at 49819/$C29B for details of the proper procedure.

To retrieve a byte from a serial device (device number greater than 3), a branch is taken to the routine at 61276/$EF5C. For RS-232 input (device 2), a branch is taken to the routine at 61287/$EF67. For tape (device 1), this routine falls through into the next routine.

61224 $EF28

Accepts a byte from tape.
Retrieves a byte from the cassette buffer (2816-3007/$OBO0-$0BBF), then checks the next character in the buffer. If that character is the zero byte marking the end of the file, bit 6 of the tape status flag (144/S90) will be set to %1 to indicate that the end has been reached. If all characters have been retrieved from the buffer, the next block of data will be read. The retrieved byte will be in the accumulator upon exit and

the X register contents will be preserved. Carry will be clear unless the RUN/STOP key has been pressed (the accumulator will contain 0/$00) or an error is encountered when reading the data block from tape (the accumulator will contain the error code).

61276 SEF5C

Accepts a byte from a serial device.
Checks the serial status flag (144/$90) and returns the code for the RETURN character (13/$0D) if the status is nonzero (indicating error or end of file). Otherwise, the routine jumps to the Kernal ACPTR routine [$E43E] to retrieve a byte from a serial device.

61287 6EF67

Accepts a byte from RS-232.
Calls the RS-232 GETIN routine at 61181/SEEFD to retrieve a character from the RS-232 input buffer. If the character value is nonzero, indicating that a valid character has been returned, the routine exits with that value in the accumulator. However, because of a branch to an incorrect address, the routine neglects to clear the carry bit, which will be set as a result of the comparison with zero. Thus, contrary to other versions of the Kernal, the carry bit will be set when the character code is valid. If the value returned is zero, the routine checks bit 6 of the RS-232 status flag (2580/$0A14) to determine whether the DSR signal is still present (indicating that the external device is still active). If the flag value indicates that the DSR signal is present, the routine loops back to see whether a character has arrived in the input buffer. If DSR is missing, the character code for RETURN, 13/$0D, is returned in the accumulator with the carry bit clear. In any case, the X register value is unused and the Y register value is preserved.

61305 $EF79 BSOUT

Sends a **byte** to **the current** output device.
(This routine is the normal target of the jump table entry at 65490/$FFD2, via the indirect vector at 806/$0326.)
If the current output device (154/S9A) is the screen (device 3), the routine jumps to the screen editor routine to display the character in the accumulator [$C00C]. If a serial device is specified (device number greater than 3), the routine jumps to the

CIOUT routine [$E503] to send the byte in the accumulator over the serial bus. Otherwise, the value to be sent is stored in 158/$9E, and the X and Y register contents are placed on the stack for later restoration. For RS-232 (device *2),* a branch is taken to 61367/$EFB7. The branch will also be taken if the keyboard (device 0) is specified as the output device, but that shouldn't happen if you use normal Kernal calls to set the device number—CKOUT [$F14C] won't accept the keyboard as an output device. For tape (device 1), this routine falls through into the following routine.

61332 $EF94
Sends a byte to tape.

Checks whether the cassette buffer has been filled. If not, the byte stored in 158/$9E is placed in the next available position in the buffer. However, if the buffer is currently full, the current block must be written before the character can be placed in the buffer. The subroutine to write the block to tape [$EA15] is called; then the type identifier value for a data block (2/$02) is placed in the first byte of the buffer, and the buffer pointer (166/SA6) is initialized to the next buffer position, If the RUN/STOP key is pressed while the block is being written, the routine will exit with the status register carry bit set and with the accumulator containing the value 0/$00.

After the character is placed in the buffer, the X and Y register values will be restored from the stack, and the accumulator will be reloaded with the stored byte value so that all registers have the same value on exit that they did on entry. The carry bit will be clear upon exit.

61367 SEFB7
Sends a byte to the RS-232 port.

Calls the routine at 59231/$E75F to store the character code from the accumulator into the RS-232 output buffer; then jumps into the preceding routine to restore the accumulator and X and Y register values from the stack and exits with carry clear.

61373 $EFBD OPEN
Opens a logical file to a specified device.
(This routine is the normal target of the jump table entry at 65472/SFFCO, via the indirect vector at 794/$031A.)

Checks whether the specified logical file number (184/$B8) is already used for a currently open file and exits with a Kernal file-open error if it is (the status register carry bit will be set, and the accumulator will hold the error code, 2/$02).

Next, the routine checks the number of files currently open (152/$98) and exits with a Kernal too-many-files error if ten files are already open (the carry bit will be set and the accumulator will hold the error code, l/$01). Otherwise, the number of open files is incremented and the logical file number is added to the file number table (866/$0362), The current secondary address (185/$B9) is ORed with the value 96/S60 and placed in the corresponding position in the secondary address table (886/$0376). The OR step is significant; some Kernal routines expect bits 5 and 6 of the secondary address to be set. The current device number (186/$BA) is placed in the corresponding position in the device number table (876/S036C).

For files opened to the keyboard (device 0) or screen (device 3), the routine exits at this point, since no further setup steps are required for those devices. For RS-232 files (device 2), a branch is taken to the routine at 61504/$F040. For tape (device 1), a branch is taken to the routine at 61430/$EFF6. For a serial device (device number greater than 3), the subroutine at 61643/$F0CB is called; then the routine exits with carry clear (except in the case where the specified device is not present).

61430 SEFF6

Opens a file for input or output to tape.

Checks the cassette buffer address and exits with a Kernal illegal-device-number error if it is less than 512/$0200. The lower four bits of the secondary address are then tested. If all are %0, the file has been opened for reading, so the routine searches for the specified tape header (or simply for the next header if no filename is specified). If the RUN/STOP key is pressed while the routine is searching for the header, the routine will exit with carry set and with the accumulator holding the value 0/$00. If an end-of-tape header is read before the data header is found, the routine exits with a Kernal file-not-found error (the carry bit will be set, and the accumulator will contain the error code, 4/$04), If a file header is found, the routine sets the buffer pointer (166/$A6) to 191/$BF {the value to indicate that the buffer is empty) so that the first

block of data will be read when BASIN is called for the first time; then it exits with carry clear.

This routine does have a flaw: It neglects to check whether the loaded header block has a type identifier of 4 (the type identifier for data file headers). Thus, if a program file (type identifier of 1 or 3) is found with the specified name, or before the next data header if no name is specified, it will be opened. However, tape program files cannot be read or written, only saved or loaded, so it does no good to open them for reading. When a program file is opened, a long block error occurs after you try to read the first block.

If any of bits 0-3 of the secondary address are %1, a data file will be opened for writing. The subroutine to request that the PLAY and RECORD buttons be pressed is called. If the RUN/STOP key is pressed during that subroutine, the routine will exit with carry set and with the accumulator containing 0/$00. The subroutine at 59673/$E919 is then called to write a data header for the file (type identifier of 4). After the header is written, the data block type identifier value (2/$02) is placed in the first position of the cassette buffer; then the buffer pointer is initialized to the next position, and the routine exits with carry clear.

61504 $F040

Opens a file for RS-232 communications.

Calls the subroutine at 61616/$F0B0 to initialize the CIA #2 port lines to the user port; then clears the RS-232 status flag (2580/$OA14) and retrieves up to four characters from the current filename (if more are specified, those after the first four are ignored). The first character, if present, is placed in the RS-232 control register (2576/$0A10), the second into the command register (2577/$0All), and the last two into the baud rate timing constant (2578-2579/$0A12-$0A13). The number of bits to be sent or received is calculated and stored in 2581/$0A15. The lower four bits of the control register value are checked to determine the baud rate to use. If the bits are all %0, a custom rate is indicated, so the following step to load a value from the tables is skipped. Otherwise, the value from bits 0-3 is used as an index into one of the baud rate timing constant tables, depending on the video system in use (59470/$E84E for NTSC or 59490/$E862 for PAL), and the specified table entry is loaded into 2578-2579/$0A12-$0A13.

No error checking is done, so if you specify a value greater than 10 (%1010) in those bits, you'll wind up with an invalid timing constant and won't be able to send or receive data. Also, for higher baud rates, the time between interrupts is so short that the system may not have time to process received bytes. In general, stick to rates of 1200 baud or lower.

The rate factor value in 2578-2579/$0A12-$0A13 is then converted into a bit timing constant, the number of CIA timer counts for the bit duration, and the result is stored in 2582-2583/$0A16-$0A17. The formula is *bit time* equals *rate factor* times 2 plus 200.

Next, the routine checks the handshaking setting specified by bit 0 of the command register. For x-line handshaking, the routine checks the CIA port bit connected to the DSR line (pin L of the user port). If the external RS-232 device is not holding this line high, bit 6 of the RS-232 status flag will be set. However, there's a bug here: If the line is high, indicating that the external device is present, the routine will proceed with carry set, which is normally used to indicate a problem. (Carry clear upon return from OPEN is supposed to mean that the file has been opened successfully.)

If you're calling OPEN from machine language, you can just remember that a set carry can be ignored when x-line handshaking is in use. However, BASIC still thinks a problem has occurred and will give a DEVICE NOT PRESENT error when the device is present. The converse side of the problem is that carry will be clear when DSR is low, so BASIC will think everything is okay when the device is absent.

The final step is to set the input buffer tail pointer (2585/S0A19) equal to the head pointer (2584/$0A18), and the output buffer head pointer (2586/S0A1A) equal to the tail pointer (2587/$0AlB). This effectively makes both buffers initially empty.

61616 $F0B0

Sets up CIA #2 ports for RS-232 communications.

Disables all interrupt sources from CIA #2; then makes the user port lines connected to bits 0 and 3-7 of port B inputs, while making the lines for bits 1-2 outputs and setting them high (+5 volts). The line for bit 2 of port A (initialized to an output line during IOINIT) is set high. This is the transmitted data line, which is normally held high when no data is being

sent. For information on the usage of the other lines, refer to the discussion of CIA #2 in Chapter 8. Finally, the RS-232 activity flag (2575/S0A0F) is cleared to indicate that no CIA #2 interrupts are enabled.

61643 $FOCB

Opens a file for serial bus communications.

Checks the current secondary address (185/$B9) and exits immediately (with the status register carry bit clear) if the secondary address value has bit 7 set (if the value is greater than 127). The routine also exits without performing any other actions if the length of the name specified for the file being opened (183/SB7) is zero. Otherwise, the serial status flag (144/$90) is cleared to zero, and the specified device (186/$BA) is commanded to listen. If the serial status flag indicates that the device has not responded, the return address of the calling routine will be discarded and the routine will exit with a Kernal device-not-present error (the carry bit will be set and the accumulator will contain the error code, 5/$05). If the device has responded, the secondary address is sent as a command using the SECOND routine [$E4D2]. (The upper four bits of the secondary address are masked to %0000, so only the lower four bits are significant.) Again, if the device does not respond, the routine exits OPEN with a Kernal device-not-present error. If a filename has been specified, the characters of the name are sent to the serial device. (Since the ATN line is allowed to go high after the secondary address is sent, the filename characters are not seen as commands.) The routine ends by commanding the device to unlisten.

61702 $F106 CHKIN

Sets the current input file for GETIN and BASIN.
(This routine is the normal target of the jump table entry at 65478/$FFC6, via the indirect vector at 798/$031E.)

Checks whether a file with the logical file number specified in the X register is currently open by verifying that a matching entry exists in the logical file table at 866-875/$0362-$036B. If no match is found, the routine exits with a Kernal file-not-open error (the status register carry bit will be set and the accumulator will hold the error code, 3/$03). If an entry for the file is found in the table, the file number is loaded into 184/SB8, and the corresponding device number and secondary

address values are loaded into 186/SBA and 185/SB9, respectively. If the device number is 0 (keyboard) or 3 (screen), the routine skips ahead to set this as the input device number and exit, since that's all that's necessary for input from those sources. For device numbers greater than 3 (serial devices), a branch is taken to the routine at 61735/$F127. For device 2 (RS-232), a jump is taken to the routine at 59285/$E795 to prepare for RS-232 input. For device 1 (tape), the secondary address is tested. If it is not 96/$60 (0/$00 before the OR in the OPEN routine), the file has been opened for writing, so the routine exits with a Kernal not-input-file error (the status register carry bit will be set and the accumulator will hold the error code, 6/$06). Otherwise, the device number is set as the current input device (153/$99), and the routine exits with carry clear.

61735 $F127

Prepares a serial device file for input.

Sends a TALK command to the serial device specified in the accumulator, then tests the serial status flag (144/$90) to see whether the device has responded. If the flag indicates that the device is not present, the routine exits with a Kernal device-not-present error (the status register carry bit will be set and the accumulator will contain the error code, 5/$05). If the device has responded, the routine checks the value of the secondary address (185/$B9). If bit 7 of the value is %1 (if the secondary address is 128 or greater), no secondary address is sent—the routine simply performs the talk-listen turnaround. Otherwise, the TKSA routine [$E4E0] is called to send the secondary address as a command on the serial bus and perform the talk-listen turnaround. If the device responds again, the routine sets the device number as the current output device (154/$9A) and exits with carry clear. If the device does not respond, the routine exits with a Kernal device-not-present error.

61772 $F14C CKOUT

Sets the current output file for BSOUT.

(This routine is the normal target of the jump table entry at 65481/$FFC9, via the indirect vector at 800/$0320.)

Checks whether a file with the logical file number specified in the X register is currently open by verifying that a matching entry exists in the logical file table at 866-875/$0362-$036B.

If no match is found, the routine exits with a Kernal file-not-open error (the status register carry bit will be set and the accumulator will hold the error code, 3/$03). If an entry for the file is found in the table, the file number is loaded into 184/$B8, and the corresponding device number and secondary address values are loaded into 186/$BA and 185/$B9, respectively. If the device number is 0, the routine exits with a Kernal not-output-file error (the status register carry bit will be set and the accumulator will hold the error code, 7/$07), since it's not possible to send output to the keyboard. If the device number is 3, the routine skips ahead to set this as the output device number and exit, since that's all that's necessary to route output to the screen. For device numbers greater than 3 (serial devices), a branch is taken to the routine at 61805/$F16D. For device 2 (RS-232), a jump is taken to the routine at 59177/$E729 to prepare for RS-232 output. For device 1 (tape), the secondary address is tested. If it is 96/$60 (0/$00 before the OR in the OPEN routine), the file has been opened for reading, so the routine exits with a Kernal not-output-file error. Otherwise, the device number is set as the current output device (1154/$9A), and the routine exits with carry clear.

61805 8F16D

Prepares a serial device file for output.

Sends a LISTEN command to the device specified in the accumulator, then tests the serial status flag (144/$90) to see whether the device has responded. If the flag indicates that the device is not present, the routine exits with a Kernal device-not-present error (the status register carry bit will be set and the accumulator will contain the error code, 5/$05), If the device has responded, the routine checks the value of the secondary address (185/$B9). If bit 7 of the value is %1 (if the secondary address is 128 or greater), no secondary address is sent—the routine simply allows the serial bus ATN line to go high. Otherwise, the SECOND routine [$E4D2] is called to send the secondary address as a command on the serial bus. If the device responds again, the routine sets the device number as the current output device (154/$9A) and exits with carry clear. If the device does not respond, the routine exits with a Kernal device-not-present error.

61832 $F188 CLOSE

Closes a specified logical file.
(This routine is the normal target of the jump table entry at 65475/$FFC3, via the indirect vector at 796/$031C.)
Shifts the value of the status register carry bit upon entry into bit 7 of 146/$92; then checks whether a file with the logical file number specified in the accumulator is currently open. If not, the routine simply exits with carry clear. If an entry for the file is found in the file number table, the file number is loaded into 184/$B8, and the corresponding device number and secondary address values are loaded into 186/$BA and 185/$B9, respectively. The file's position in the tables is placed on the stack for later retrieval. The routine then checks whether the file is open to the keyboard (device 0) or screen (device 3). In either of these cases, simply removing the table entries for the file is sufficient to close the file, so a branch is taken to the routine at 61924/$F1E4. For device numbers greater than 3 (serial devices), a branch is taken to the routine at 619O3/$F1CF. For device 1 (tape), a branch is taken to the routine at 61865/$F1A9. For device 2 (RS-232), the table entries for the file are deleted, then a jump is taken to the routine at 61616/$F0B0 to reinitialize the CIA ports and disable the interrupts that drive RS-232 communications.

61865 $F1A9

Closes a tape file.
Checks the secondary address for the file (185/$B9). If it is 0, indicating that the file has been opened for reading, all that is required is to delete the table entries for the file, so a branch is taken to 61924/$F1E4. If the file has been opened for writing, the routine adds a zero following the last data byte in the buffer (which contains the final block of data for the file), then writes the final block to tape. If the RUN/STOP key is pressed while the last block is being written, the routine will exit with status register carry bit set and with the accumulator holding 0/$00. If bit 1 of the secondary address is set to %1, then an end-of-tape header will be written following the final data block. The routine then jumps to 61924/$F1E4 to remove the table entries for the file.

61903 SF1CF

Closes a file on a serial device.

Checks the preserved status of the cany bit when the CLOSE routine is entered. If carry is set, and if the device number (186/$BA) is 8 or greater, and if the secondary address for the file is 15 (indicating that the file has been opened as a command channel to the drive), a branch is taken to 61924/$F1E4 to simply remove the file entries rather than actually closing the file. This provides a solution to a problem in previous versions of the Kernal. Closing the command channel to a drive also closes all other open files on the drive, possibly an unwanted side effect. By calling CLOSE with carry set, this can now be avoided.

If carry is clear when CLOSE is called, the normal closing steps are performed: The subroutine at 62878/$F59E is called to close the file on the serial device; then the routine falls through into the next one to remove the table entries for the file.

61924 SF1E4

Removes an entry from the logical file tables.

Retrieves the offset to the file's position in the logical file number, device number, and secondary address tables from the stack and decrements the number of open files (152/$98). Next, the routine checks whether the file to be deleted is the last entry in the tables. If so, decrementing the number of open files is sufficient to effectively remove the file's table entries; and the routine exits at this point. Otherwise, the current last entry in the table is copied to the specified file's position, overwriting the entries for the file to be deleted. Carry will always be clear upon exit.

61954 $F202

Checks whether a file is already open.

Clears the serial status flag (144/$90), then searches the logical file number table (866-875/$0362-$036B) for an entry with the same number as the value specified in the X register. If no match is found, or if no files are open, the routine exits with the status register N bit set (test with BMI). If a file using the specified number is already open, the status register Z bit will be set (test with BEQ).

61970 $F212

Load parameters for a logical file.
Loads the current logical file number (184/$B8), current device number (186/SBA), and secondary address (185/$B9) with the values for a specified open file from the tables at 866-895/$0362-$037E The routine should be called with the X register containing the offset (0-9) into the tables for the file's entry.

61986 8F222 CLALL

Clears file table entries,
(This routine is the normal target of the jump table entry at 65511/$FFE7, via the indirect vector at 812/$032C.)

Resets the number of open files (152/$98) to zero, then falls through into the next routine to reestablish default I/O channels. Note that this routine doesn't actually close any files that may be open to external devices. Unclosed tape or disk files may cause problems and should be avoided. See the CLOSE_ALL entry at 62013/$F23D for a routine that properly closes all files opened to a specified serial device.

61990 $F226 CLRCH

Resets default I/O channels.
(This routine is the normal target of the jump table entry at 65484/$FFCC, via the indirect vectoT at 802/$0322.)
Sends an UNLISTEN command over the serial bus if the current output device number (154/S9A) is greater than 3, and an UNTALK command if the current input device number (153/$99) is greater than 3. The channels are then reset to the default devices: 3 (screen) for output and 0 (keyboard) for input.

62013 $F23D CLOSE_ALL

Closes all open files for a specified serial device.
(This routine has a jump table entry at 65354/$FF4A.)

Stores the value in the accumulator as the current device number (186/$BA). If the specified device number is the current input or output device, the input or output channel is reset to the default device (keyboard or screen). Next, the routine searches the device number table at 876-885/$036C-$0375 for files that might be open to the specified device. Any that are found are closed by using the Kemal CLOSE routine 65475/$FFC3.

62053 $F265 LOAD

Loads or verifies a program file from disk or tape.
(This routine has a jump table entry at 65493/SFFD5.)

Stores the address value from the X and Y registers into the pointer to the starting address for the load (195-196/ $C3-$C4), then takes an indirect jump through the ILOAD vector (816/$0330). The vector normally points back to the location immediately following the jump, but you can modify the action of the LOAD routine by redirecting this vector to a routine of your own. See the ILOAD entry for details.

The operation type value in the accumulator is stored in the load/verify flag'(147/$93), and the tape/serial status flag (144/$90) is cleared. The device number (186/$BA) is then tested. If it's 4 or greater, a branch is taken to the routine at 62075/$F27B to attempt a load from the specified serial device. Otherwise, a jump is taken to the routine at 62246/ $F326 to attempt a load from tape.

62075 $F27B

Loads or verifies a file from a serial device.

Begins by clearing bits 6 and 0 of the fast serial flag ($2588/$0AlC). Bit 6 is cleared so that the routine which attempts fast serial output will have to verify for itself that fast communications are available; there is no apparent reason for clearing bit 0. The current secondary address (185/$B9) is placed in temporary storage (158/$9E). If the length of the current filename (183/$B7) is zero, the routine exits with the Kernal missing-filename error (the status register carry bit will be set and the accumulator will hold the error code, 8/$08). Otherwise, the length value is placed in temporary storage (159/$9F), and (if Kernal control messages are allowed) SEARCHING FOR, followed by the filename, is displayed. The subroutine at 62369/$F3Al is called to attempt a fast serial load or verify of the specified file. If carry is clear upon return (indicating that a fast load or verify has been performed), the routine exits with carry clear and with the X and Y registers holding the ending address for the data. Otherwise, the routine proceeds with the standard serial load or verify.

The filename length is restored from temporary storage, and the current secondary address is set to 96/$60—corresponding to a secondary address of zero, the value for reading a file. The serial OPEN routine [$F0CB] is called to send

the filename to the specified device; then the device is commanded to talk and is sent the secondary address. The first two bytes, which, for a program (PRG) file, contain the starting address, are loaded into the working pointer (174-175/$AE-$AF). If the original secondary address value in 158/$9E is 0/$00, indicating that a relocating load has been specified, the starting address in the pointer is replaced with the one in 195-196/$C3-$C4. If Kernal messages are allowed, LOADING or VERIFYING is displayed.

The main reading step consists of clearing the read timeout bit (bit 1) of the serial status flag, checking for a RUN/STOP keypress, calling ACPTR [$E43E] to read a byte from the program file, and testing whether the read timeout bit has been set during the read. If it has been set, the routine loops back to try to read the byte again. If RUN/STOP is pressed, the routine goes to 62901/$F5B5 to halt the operation.

For a verify operation (nonzero value in 147/$93), the byte read from disk is compared to the one at the address pointed to by 174-175/$AE-$AF in the bank specified in 198/$C6. If the bytes do not match, the verify error bit (bit 4) in the serial status flag is set to %1. For a load operation (0 in 147/$93), the byte read from disk is stored at the address pointed to by 174-175/$AE-$AF in the bank specified in 198/$C6. In either case, the address in the pointer is then incremented and compared against the value 65280/$FF00. If it reaches this value, a Kernal out-of-memory error occurs (the routine exits with carry set and with the accumulator holding the error code, 16/$10). If the EOI bit in the serial status flag (bit 6) is not %1, the routine loops back to read another byte. When EOI is set, indicating that the end of the file has been reached, the routine sends an UNTALK command, closes the file, loads the address from 174-175/$AE-$AF (which will point to the location immediately following the last byte loaded) into the X and Y registers, then exits with carry clear.

62246 $F326

Loads or verifies a program file from tape.

Exits with the Kernal illegal-device-number error if the device number in the accumulator is not 1/S01, or if the cassette buffer address is less than $0200. (The status register carry bit will be set and the accumulator will hold the error code,

9/$09.) Otherwise, a request for the PLAY button will be printed if no buttons are currently pressed, and if Kernal messages are allowed. If the RUN/STOP key is pressed while waiting for a button press (or at any other time during the routine), the routine will exit with carry set and with 0/$00 in the accumulator. If Kernal messages are allowed, SEARCHING is displayed; if a nonzero filename length is specified, that will be followed by FOR and the filename. If a filename is supplied, the routine looks for a specified header block [$E99A], Otherwise, it merely loads the next header block [$E8D0], In either case, if an end-of-tape header (type identifier of 5) is encountered, the routine exits with a Kernal hie-not-found error (the carry bit will be set and the accumulator will hold the error code, 4/$04). The routine also exits with carry set if bit 4 of the tape status flag (144/$90) is %1 after the header is loaded, indicating that a read error has occurred. If the type identifier byte for the header is not 1 or 3, this is not a program file, so the routine loops back to read another header.

If the type identifier is 3, indicating that this is a nonrelocatable program file, or if the secondary address (185/$B9) is nonzero, the current starting address pointer value in 195-196/$C3-$C4 is replaced with the one specified in the two header bytes following the type identifier. Next, the ending address for the file is calculated and stored in 174-175/$AE-$AF. If the ending address value is greater than 65279/$FEFF, the routine exits with a Kernal out-of-memory error (the carry bit will be set and the accumulator will hold the error code, 16/$10). The starting address is transferred to a working pointer (193-194/$C1-$C2) and, if Kernal messages are allowed, LOADING or VERIFYING is displayed. The subroutine at 59899/$E9FB is used to read the program data from tape into the specified area of memory (carry should be clear upon return if the load is successful). Finally, the ending address is loaded into the X and Y registers before exiting.

62369 $F3A1

Attempts to set up fast serial load or verify.

Checks the first character of the filename to be loaded and exits immediately if it's $, indicating that a disk directory rather than a program is to be loaded. Otherwise, the routine opens a command file (secondary address of 15) to the specified serial device. If the file is not opened successfully, the routine

exits with the Kernal device-not-present error (tVie status register carry bit will be set and the accumulator will hold the error code, 5/$05). If the command channel has been opened, the burst mode load command—UO, followed by the value 31/$1F to specify a program file, followed by the codes for the characters of the filename—is sent to the drive. The CLRCH routine [$FFCC] is called to reset the default I/O channels. Bit 7 of the fast serial flag at 2588/$0AlC is then tested. If the serial device is capable of fast serial communications, the bit will have been set to %1 by the BSOUT routine calls used to send characters to the device. In this case, the routine branches to 62442/SF3EA to perform the operation using the very high speed burst mode. If fast serial communications are not available, the command channel file is closed, and the routine returns with carry set, indicating that a standard (slow) load or verify must be performed.

62442 $F3EA

Loads or verifies a file using fast serial burst mode.

Performs a high-speed load or verify of the file specified in the preceding routine. Burst mode loads are quite different from standard loads. In burst mode, data is sent a sector (254 bytes) at a time. A status byte is sent before the data bytes for each sector. The serial bus CLK line must be toggled between bytes to acknowledge receipt of the byte.

The routine begins by restoring the filename length (183/$B7) from temporary storage (159/$9F), then disabling IRQ interrupts, allowing the serial bus CLK line to go high and setting the serial port for fast serial input. The status byte for the first sector is read. If the status value is 2/$02, the specified file has not been found, so the command channel to the drive is closed and the routine exits with a Kernal file-not-found error (the status register carry bit will be set and the accumulator will hold the error code, 4/$04). If the status byte is 31/$1F, the file is only one block long. The LOADING or VERIFYING message is then displayed (if Kernal control messages are allowed), and the first two data bytes from the first sector are then read and stored in the starting address pointer (174-175/$AE-$AF), If the stored secondary address (158/$9E) is zero, a relocating load has been specified, so the pointer is reloaded with the value from 195-196/$C3-$C4. The starting address is then transferred to the working pointer (172-173/$AC-$AD).

Before each sector is read, the RUN/STOP key is tested. If that key has been pressed, a branch is taken to the routine at 62630/$F4A6 to halt the operation. Otherwise, the count of data bytes in the sector (165/$A5) is initialized—252 bytes for a full first sector, 254 bytes in following full sectors, and a variable number of bytes in the last sector—and the subroutine at 62661/$F4C5 is called to load or verify the data bytes. After each sector is loaded or verified, the status byte for the next sector is read. If the status value is 0 or 1, the routine loops to read the next sector. If it is 31/$IF, the next block is the last one for the file, so an additional byte is read from the device. This value will be the number of bytes in the final sector. If the status value is 2 or greater (but not 31), an error has occurred, so a branch is taken to the routine at 62616/$F498 to handle the error.

After all bytes for the file have been read, the routine allows the CLK line to go high, reenables IRQ interrupts, and closes the command channel file. (The status register carry bit is set before CLOSE [$FFCC] is called so that the special command file close is performed.) The status register carry bit will be clear upon exit for a successful load. Bit 4 of the serial status flag (144/$90) will reflect the success of a verify operation.

62616 $F498

Handles read error during burst mode load/verify.

Sets bit 1 of the serial status flag (144/$90) to %1 to indicate a read timeout. Then it discards the return address of the calling routine and exits with the status register carry bit set and the accumulator holding the value 41/$29, the code for BASIC'S FILE READ error message.

62630 SF4A6

Stops burst mode load/verify if RUN/STOP key pressed.

Closes the file, then changes the current secondary address to zero, discards the return address of the calling routine, and exits with the status register carry bit set and the accumulator holding the value 0/$00.

62642 $F4B2

Aborts burst mode load/verify if maximum address exceeded. Closes the file, then discards the return address of the calling routine and exits with the Kernal out-of-memory error (upon exit, the status register carry bit will be set and the accumulator will hold the error code, 16/$10).

62650 SF4BA

Reads a byte using fast serial hardware.

Waits for a serial register interrupt to occur on CIA # 1 (indicating that a byte has been received via the fast serial hardware), then retrieves the byte from the serial data register and returns it in the accumulator.

62661 $F4C5

Loads or verifies a block of data using burst mode.

Waits for a serial register interrupt to occur on CIA #1 (indicating that a byte has been received via the fast serial hardware); then retrieves the byte from the serial data register and toggles the CLK line to acknowledge reception of the byte. If the operation flag (147/S93) contains a nonzero value, indicating verify, the byte read from the serial bus is compared with the one pointed to by 174-175/$AE-$AF in the bank specified in 198/$C6. If the bytes do not match, the verify error bit (bit 4) of the serial status flag (144/$90) is set to %1. For a load operation, the byte read from the bus is stored at the address pointed to by 174-175/$AE-$AF in the bank specified in 198/$C6. The address value in the pointer is then incremented; if it exceeds 65279/$FEFF, the routine will exit with the status register carry bit set. The count of bytes to be read from this sector (165/$A5) is decremented. If bytes remain to be read, the routine is repeated. Otherwise, it exits with carry clear and with the accumulator holding the high byte of the next load address.

62723 $F503

Toggles state of serial bus CLK line.

Reverses the value of bit 4 of CIA #2 port A. Since the line for this bit is connected to the serial bus CLK output line, this will reverse the state of the line. This is the handshake for burst mode loads from disk.

62735 $F50F

Displays SEARCHING FOR message.

Checks the Kernal message flag (157/$9D), exiting immediately if messages are not allowed. The message SEARCHING is displayed; then the current filename length (183/$B7) is tested. If it's nonzero, the message FOR and the characters of the filename are displayed following SEARCHING.

62771 $F533

Displays LOADING or VERIFYING message.

Loads the accumulator with the offset for either the LOADING or VERIFYING message, depending on whether the value in the operation flag (147/S93) is zero or nonzero. The routine at 63262/$F7lE is then called to display the message (if Kernal messages are allowed).

62782 $F53E SAVE

Saves a block of memory to tape or disk.
(This routine has a jump table entry at 65496/$FFD8.)
Stores the address value from the X and Y registers into the pointer to the ending address for the save (174-175/$AE-$AF), and stores the address from the two-byte zero-page pointer specified in the accumulator into the pointer to the starting address for the save (193-194/$C1-$C2). The routine then takes an indirect jump through the ISAVE vector at 818/$0332. That vector normally points back to the location immediately following the indirect jump, but you can modify the actions of the SAVE routine by redirecting the vector to your own routine. See the ISAVE entry in Chapter 2 for details.

The device number (186/$BA) is then tested. If it's 1 (tape), a branch is taken to the routine at 62920/$F5C8. If it's 4 or greater (serial device), a branch is taken to the routine at 62817/$F561. Otherwise, the routine exits with the Kernal illegal-device-number error (the status register carry bit will be set and the accumulator will hold the error code, 9/$09).

62817 SF561

Saves a block of memory to a serial device.

Begins by checking the filename length (183/$B7). If the length value is zero, the routine exits with the Kernal missing-filename error (the status register carry bit will be set and the accumulator will contain the error code, 8/$08). The second-

ary address is set to 97/$61, the value to open a program file for writing; then a file is opened using the current filename (pointed to by 187-188/$BB-$BC). A LISTEN command is sent to the current device (186/$BA) along with the secondary address. The low and high bytes of the starting address for the save are then written as the first two bytes of the file.

The subroutine at 63436/$F7CC is used to retrieve a byte from the specified memory area, and the subroutine at 58629/$E503 is used to write the byte to the file (using fast serial mode if it is available). After each byte, the routine checks whether the RUN/STOP key has been pressed. If it has been pressed, a branch is taken to the routine at 62901/$F5B5 to abort the SAVE. Otherwise, the address pointer is incremented, and the process repeats until all bytes have been written. (Before this routine is called, the starting address must be loaded into 193-194/$C1-$C2 and the ending address plus 1 into 174-175/$AE-$AF.) When all bytes have been written, an UNLISTEN command is sent to the serial device; then the routine falls through into the following one to close the file.

When fast serial communications are available, files are loaded by sectors (254-byte chunks of data) using a special feature of the 1571 drive known as burst mode. However, fast mode SAVEs are still done byte by byte. This is the reason more time is required to save a file using fast serial mode than to load a file of the same length,

62878 $F59E

Closes a file on a serial device.

Exits immediately if the secondary address is greater than 127/$7F (if bit 7 of 185/$B9 is %1). Otherwise, a LISTEN command is sent to the current serial device; then the upper four bits of the secondary address are set to %1110/$E to form the CLOSE command for the logical file. This command is then sent to the current serial device, followed by an UNLISTEN command. The routine then exits with the status register carry bit clear. The serial status flag (144/S90) will reflect the success of the operation.

62901 $F5B5

Aborts LOAD or SAVE to serial device.

Calls the subroutine at 62878/SF59E to close the file, then loads the accumulator with zero and exits with the status register Z and carry bits set.

62908 $F5BC

Displays SAVING message and filename.
Checks the Kernal message flag (157/$9D) and exits immediately if Kernal messages are not allowed. Otherwise, the message SAVING is printed. If a filename is being used (indicated by a nonzero length value in 183/$B7), the name is also printed following SAVING.

62920 $F5C8

Saves a block of memory to tape.

Begins by checking the tape buffer address, exiting immediately if it is less than $0200. If no buttons are currently pressed on the tape drive, the subroutine to print PRESS PLAY &. RECORD ON TAPE (if Kernal messages are allowed) and wait for a button press is called. If the RUN/STOP key has been pressed while waiting for the tape button (or at any other time during this routine), the routine exits with the status register carry bit set and with the accumulator holding 0/$00. Otherwise, if Kernal messages are allowed, the SAVING message is displayed, followed by the filename (if one is used). Next, a header type identifier value is selected for the file, according to bit 0 of the secondary address (185/$B9). If that bit is %0, a type identifier of 1 (relocatable file) is used. If the bit is %1, the type identifier byte will be 3 (nonrelocatable file). The subroutine at 59673/$E919 is used to write the header for the file. The subroutine at 59928/$EA18 is used to write the data to the file. (Before this routine is called, the starting address must be loaded into 193-194/$C1-$C2 and the ending address plus 1 into 174-175/$AE-$AF.) Finally, bit 1 of the secondary address is tested. If that bit is %0, the routine exits with carry clear. However, if the bit is %1, an end-of-file header (type identifier of 5) is written to tape following the program.

62968 $F5F8 UDTIM

Updates jiffy timers and checks RUN/STOP key column.
(This routine has a jump table entry at 65514/$FFEA.)

Increments the software jiffy clock at 160-162/$A0-$A2 (part of the normal IRQ sequence). If the timer has reached a count of 5184OO1/$4F1AO1 (corresponding to a time of 24:00:00), all three timer bytes are reset to zero. Next, the jiffy timer at

2589-2591/$0AlD-$0AlF is decremented. The 128 uses this timer only when executing the BASIC SLEEP statement; otherwise, it is available for your own timing applications.

If the PAL/NTSC flag (2563/S0A03) indicates that PAL (European) video is in use, the jiffy clock compensation counter at 2614/$0A36 is decremented. Each time this counter rolls over from 0 to 255/$FF, it is reset to 5 and the routine is repeated. Thus, in a PAL system the timers are updated 6 times for every 5 IRQ interrupts, or 60 times for every 50 interrupts. As a result, the clock is incremented 60 times per second regardless of whether the system interrupts occur at the NTSC rate (60 times per second) or the PAL rate (50 times per second). The routine then falls through into the following one.

63037 $F63D

Scans RUN/STOP key column.

Reads the CIA port connecting the rows of the keyboard matrix. The keyboard scan routine [$C55D], normally performed earlier in the IRQ sequence, leaves the CIA port connected to the columns of the keyboard matrix set to scan column 7, the column containing the RUN/STOP key (see Figure 7-1 in Chapter 7). If the RUN/STOP key is pressed, the matrix rows for columns 1 and 6 (containing the SHIFT keys) are tested. If any key in those columns is pressed, the routine exits (so SHIFT-RUN/STOP will not be registered as a RUN/STOP keypress). Otherwise, the row register value is stored in 145/$91.

63070 $F65E RDTIM

Reads the software jiffy clock.

{This routine has a jump table entry at 65502/$FFDE.)

Returns the values in the software jiffy clock locations (160-162/$A0-$A2), which hold the count of jiffies (1/60 second intervals) since the system has been turned on or since the clock time has last been reset. Upon return, the accumulator will hold the low byte of the clock value (from 162/$A2), the X register will hold the middle byte (from 161/$A1), and the Y register will hold the high byte (from 160/$A0).

63077 $F665 SETTIM

Sets the software jiffy clock.
{This routine has a jump table entry at 65499/$FFDB.)

Stores the value in the accumulator upon entry in 162/$A2, the low byte of the clock. The X register contents will be placed in 161/$A1, the middle byte of the clock value, and the Y register contents will be placed in 160/$A0, the high byte of the clock value.

63086 $F66E STOP

Tests for a RUN/STOP keypress.
(This routine is the normal target of the jump table entry at 655O5/$FFE1 via the 1ST0P vector at 808/$0328.)

Checks the STOP key flag (145/191), set during the UDTIM routine [$F5F8] in the IRQ sequence. If the flag contains the value 127/$7F, indicating that RUN/STOP has been pressed, the Kernal CLRCH routine [$FFCC] is called to restore default I/O, and the count of characters in the keyboard buffer (208/ $D0) is reset to zero. Upon exit, the status register Z bit will be set if the RUN/STOP key has been pressed or clear otherwise.

63100 $F67C

Handles Kernal I/O errors.

Loads the accumulator with an error number depending on the entry point into the routine, then uses BIT opcodes to fall through to handle the error.

Entry point	Error number	Meaning
63100/$F67C	1	Too many files
63103/SF67F	2	File open
63106/$F682	3	File not open
63109/$F685	4	File not found
63112/$F688	5	Device not present
63115/$F68B	6	Not input file
63118/$F68E	7	Not output file
63121/$F691	8	Missing filename
63124/$F694	9	Illegal device number
63127/$F697	16	Out of memory

Next the CLRCH routine [$FFCC] is used to reset default I/O (output to screen). If Kernal error messages are allowed, I/O ERROR # is printed, followed by the character code for the digit corresponding to the error number. Upon exit, the error number will be in the accumulator and the carry bit will be set.

1

Kernal error messages are normally disabled when BASIC is active. (BASIC substitutes its own, more verbose error messages.) However, Kernal error messages are enabled while the machine language monitor is active,

63152 $F6B0

Table of Kernal control messages.

The messages in this table are displayed by the following routine. The end of each message is marked by a character with its high bit set to %1.

Offset	Message
0/$00	I/O ERROR #
12/SOC	SEARCHING
23/$17	FOR
27/$lB	PRESS PLAY ON TAPE
46/$2E	PRESS RECORD & PLAY ON TAPE
73/$49	LOADING
81/$51	SAVING
89/$59	VERIFYING
99/$63	FOUND
106/$6A	OK

63262 $F71E

Handles Kernal control messages.

Begins by checking the Kernal message flag (157/$9D), exiting immediately if bit 7 of the flag is %0 (indicating that Kernal control messages are disabled). If control messages are allowed, the value in the Y register is used as an offset to the first character of the message in the table at 63152/$F6B0. Characters from the message string are printed until a character with its high bit set is encountered. Upon exit, the carry bit will be clear,

63281 SF731 SETNAM

Sets the length and address of filename for I/O operations. (This routine has a jump table entry at 65469/$FFBD.)

Stores the value in the accumulator upon entry as the length of the current filename (183/$B7), and the value in the X and Y registers as the starting address of the character codes for the current filename (187-188/$BB-$BC). The low byte of the address should be in the X register and the high byte in Y.

63288 $F738 SETLFS

Sets logical file number, device number, and secondary address for I/O operations.
(This routine has a jump table entry at 65466/$FFBA.)
Stores the value in the accumulator upon entry as the current logical file number (184/$B8), the value in the X register as the current device number (186/$BA), and the value in the Y register as the current secondary address (185/$B9).

63295 $F73F SETBNK

Sets data and filename banks for I/O operations.
(This routine has a jump table entry at 65384/$FF68.)

Stores the value in the accumulator upon entry as the bank number for data being saved or loaded (198/$C6), and the value in the X register as the bank where the current filename can be found (199/SC7).

63300 SF744 READSS

Reads the tape/serial or RS-232 status byte.
(This routine has a jump table entry at 65463/$FFB7.)

Checks the current device number (186/$BA); if it's 2 (RS-232), the value in the RS-232 status flag (2580/$0A14) is returned in the accumulator and the status flag is reset to zero. For other device numbers, the value in the tape/serial status flag (144/$90) is returned in the accumulator.

63324 $F75C SETMSG

Sets the Kernal message control flag.
(This routine has a jump table entry at 65424/SFF90.)

Stores the value in the accumulator upon entry as the Kernal message flag (157/S9D).

63327 $F75F SETTMO

Sets the IEEE timeout flag.
(This routine has a jump table entry at 65442/$FFA2,)
Stores the value in the accumulator upon entry as the IEEE timeout (2574/$0A0E). This location is unused by the 128. The routine is a holdover from the original PET/CBM Kernal; the IEEE-488 parallel interface is not implemented in the 128.

63331 SF763 MEMTOP

Sets or reads the system's top-of-memory pointer.
(This routine has a jump table entry at 65433/$FF99.)
Begins by checking the status register carry bit. If carry is clear, the values in the X and Y registers are loaded into the svstem top-of-memory pointer (2567-2568/$OA07-$OA08), the low byte from X and the high byte from Y. If carry is set, the current top-of-memory pointer value is returned in the X and Y registers, the low byte in X and the high byte in Y.

63346 $F772 MEMBOT

Sets or reads the system's bottom-of-memory pointer.
(This routine has a jump table entry at 65436/$FF9C.)

Begins by checking the status register carry bit. If carry is clear, the values in the X and Y registers are loaded into the system bottom-of-memory pointer (2565-2566/$0A05-$0A06), the low byte from X and the high byte from Y. If carry is set, the current bottom-of-memory pointer value is returned in the X and Y registers, the low byte in X and the high byte in Y.

63361 $F781 IOBASE

Returns base address of I/O block.
(This routine has a jump table entry at 65523/$FFF3.)

Returns the value 53248/$D000, the lowest address in the system's I/O block, in the X and Y registers, 0/$00 in X and 208/$D0 in Y.

63366 $F786 LKUPSA

Checks whether a secondary address value is used.
(This routine has a jump table entry at 65372/$FF5C.)

Searches the secondary address table (886-895/$0376-$037F) for an open file with the secondary address value specified in the Y register upon entry. If no match is found, the status register carry bit will be set upon exit. If an open file with the same secondary address is found, the corresponding logical file number will be returned in the accumulator, the device number in the X register and the secondary address in the Y register. In this case the carry bit will be clear upon exit.

63389 $F79D LKUPLA

Checks whether a logical file number value is used.
(This routine has a jump table entry at 65369/$FF59.)

Searches the logical file number table (866-875/$0362-$036B) for an open file with the logical file number value specified in the accumulator upon entry. If no match is found, the status register carry bit will be set upon exit. If an open file with the same file number is found, the logical file number will be returned in the accumulator, with the corresponding device number returned in the X register and the secondary address in the Y register. In this case the carry bit will be clear upon exit.

63397 $F7A5 DMA-CALL

Performs a DMA operation.
(This routine has a jump table entry at 65360/$FF50.)

Translates the bank number for the current operation, in the X register upon entry, into the equivalent MMU configuration register setting value; then forces bit 0 of the setting value to %0 to insure that the I/O block will be visible at 53248-57343/$D000-$DFFF. With that value in the accumulator, and with the DMA chip command register value in the Y register, the routine then jumps to the DMA request routine in common RAM [$03F0].

This routine is provided for the purpose of passing commands to the REC (RAM Expansion Controller) chip in the Commodore 1700 and 1750 RAM Expansion Modules. The chip appears in 128 memory at 57088-57098/$DF00-$DF0A in the I/O block when one of the modules is plugged in. Additional setup steps may be required, depending on the command. See the REC chip description in Chapter 8 for more information.

63406 $F7AE

Retrieves a character from the current filename.

Loads a character from the current filename address (pointed to by 187-188/$BB-$BC) in the bank specified in 199/$C7. The Y register value is used as an offset into the filename. The character will be returned in the accumulator; the X register value upon entry will be preserved during this routine.

\

63420 $F7BC
Writes a byte value to memory.
Stores the value in the accumulator upon entry into the location specified by the address in 172-173/$AC-$AD (plus the offset specified in the Y register) in the bank specified in 198/SC6.

63423 $F7BF
Writes a byte value to memory.
Stores the value in the accumulator upon entry into the location specified by the address in 174-175/$AE-$AF (plus the offset specified in the Y register) in the bank specified in 198/SC6.

63433 $F7C9
Reads a byte value from memory.
Returns with the accumulator holding the value from the location specified by the address in 174-175/$AE-$AF (plus the offset specified in the Y register) in the bank specified in 198/$C6.

63436 $F7CC
Reads a byte value from memory.
Returns with the accumulator holding the value from the location specified by the address in 172-173/$AC-$AD (plus the offset specified in the Y register) in the bank specified in 198/$C6.

63440 $F7D0 INDFET
Retrieves a character from any bank.
(This routine has a jump table entry at 65396/$FF74.)
Stores the zero-page pointer address, in the accumulator upon entry, in the INDFET address pointer byte (682/$02AA); then converts the bank number for the target address, in the X register upon entry, into the corresponding MMU configuration register setting and calls the RAM-resident portion of the INDFET routine [$02A2]. Upon return, the accumulator will hold the value from the location at the address specified in the zero-page pointer (plus the offset specified in the Y register) in the specified bank.

63450 $F7DA INDSTA

Stores the accumulator contents in any bank.
{This routine has a jump table entry at 65399/$FF77.)

Converts the bank number for the target address, in the X register upon entry, into the corresponding MMU configuration register setting; then calls the RAM-resident portion of the INDSTA routine [$02AF]. This will place the value in the accumulator into the specified bank at the address specified in a zero-page pointer, plus the offset in the Y register. The address of the zero-page pointer must be stored in location 697/$02B9 before this routine is called.

63459 8F7E3 INDCMP

Compares the accumulator contents with a value from any bank.
{This routine has a jump table entry at 65402/$FF7A.)
Converts the bank number for the target address, in the X register upon entry, into the corresponding MMU configuration register setting; then calls the RAM-resident portion of the INDCMP routine [$02BE], This will compare the value in the accumulator with the value at the address specified in a zero-page pointer, plus the offset in the Y register, in the specified bank. The address of the zero-page pointer must be stored in location 712/$02C8 before this routine is called. Upon return, the status register N, Z, and C (carry) bits will reflect the result of the comparison.

63468 8F7EC GETCFG

Translates a bank number into an MMU register setting.
(This routine has a jump table entry at 65387/$FF6B.)
Returns with the accumulator holding the MMU register setting value corresponding to the bank number in the X register upon entry. See Chapter 8 for more information on the MMU.

63472 6F7F0

Table of MMU register settings for standard banks.
Each of the 16 values in this table corresponds to the MMU configuration register value that sets up one of the 16 standard banks.

r

Bank	MMU Configuration Setting
0/$00	63/$3F (%00111111)
1/$01	127/$7F (%0111111I)
2/$02	191/$BF (%10111111)
3/$03	255/$FF (%111I1111)
4/$04	22/$16 (%00010110)
5/$05	86/$56 (%01010110)
6/$06	150/$96 <%10010110)
7/$07	214/$D6 (%11010ll0)
8/$08	42/$2A (%00101010)
9/$09	106/$6A (%0I101010)
10/SOA	170/$AA (%10101010)
11/$OB	234/$EA (%11101010)
12/$0C	6/$06 (%00000110)
13/$0D	13/$0D (%00001110)
14/$0E	1/$01 (%00000001)
15/$0F	0/$00 {%00000000}

63488 8F800

Code for Kernal RAM-based subroutines.

This area of ROM contains the code for the RAM-resident portions of the INDFET, INDSTA, INDCMP, JSRFAR, JMPFAR, and DMA_CALL routines. The routines are copied to the appropriate areas of RAM by the routine at 57549/SEOCD, part of the reset sequence.

63591 $F867 PHOENIX

Initializes function ROMs and attempts to boot a disk in the default drive.

(This routine has a jump table entry at 65366/$FF56.)

Initializes any internal or external 128 function ROMs logged during the reset sequence (by the routine at 57963/$E26B). If a ROM is detected at one of the four possible address areas for function ROMs, the corresponding byte in the ROM ID table at 2753-2756/$0ACl-$0AC4 will contain a nonzero value. For logged ROMs, the JSRFAR routine is used to call the cold start vector of the ROM ($8000 or $C0O0 in bank 4, or $8000 or $C000 in bank 8). Depending on the ROMs, there may be no return from the JSR. However, if the routine does return from all the ROM initializations {or if no ROMs are present), the X register is loaded with the value 8, and the accumulator with 48/$30, the character code for 0. The routine then falls through into the following one to attempt to boot a disk in drive 0 of device 8.

63632 $F890 BOOT_CALL

Attempts to boot a disk.
(This routine has a jump table entry at 65363/SFF53.)

Stores the value in the accumulator upon entry as the character code for the current drive number and the value in the X register as the current device number; then closes any open files on the specified device. The sector number ($C2/194) is initialized to 0 and the track number is initialized to 1 (booting begins at sector 0 of track 1). The disk block read command is copied from the table at 64008/$FA08 into the disk command buffer at 256-268/$0100-$010C. Logical file 0 (the system file) is opened as a command channel and logical file 13 as a data channel. The first boot sector is read from the specified drive into the buffer at 2816-3071/$0B00-$0BFF. If the first three bytes in the sector (bytes 0-2) are the character codes for the letters *CBM*, this is a valid first boot sector. Otherwise, the drive will be reset and the routine will exit.

For a valid first boot sector, the message BOOTING is printed; then bytes 3-4 from the sector (the load address for data from any following boot sectors) are stored in locations 172-173/$AC-$AD, byte 5 from the sector (the bank number into which data is to be loaded) is stored in 174/$AE, and byte 6 (the number of additional boot sectors to load) is stored in 175/$AF. Subsequent bytes from the sector are printed to the screen as character codes until a byte with the value zero is encountered or until the end of the sector is reached. Following that message, three periods (...) will be printed.

The bank number for boot data is transferred into the working bank number location (198/$C6). If any more boot sectors are to be loaded (if the value in 175/$AF is nonzero), data from the sectors is loaded into the bank specified in 198/$C6, starting at the address in 172-173/$AC-$AD. Additional sectors are loaded sequentially starting with sector 1 of track 1. The high byte of the load address will not be allowed to roll over from 255/$FF to 0/$00, regardless of the number of boot sectors specified, (That is, the boot sectors should not attempt to load data to addresses above 65279/$FEFF.)

After all additional boot sectors are loaded (or if no additional sectors are to be loaded), the drive is reset. The routine then searches the buffer from the zero byte marking the end of the message until another byte containing a zero is found. The characters, if any, between the zero bytes are taken to be

the name of a file to be loaded, and the drive number and a colon are placed immediately before the name in the buffer. If a filename is found, the routine attempts to load a file with that name into bank 0. Because the Kernal LOAD routine is used, this file must be PRG (program) type. This file is always loaded into bank 0, regardless of the bank number specified for boot sectors.

After the file is loaded (or if no filename is specified), the JSRFAR address pointer (3-4/$03-$04) holds the address of the buffer location following the end-of-filename zero byte, and the JSRFAR bank (2/$02) is set for bank 15. The JSRFAR routine is then used to execute the machine language subroutine following the filename in the boot sector buffer. (Some machine language code must be present, even if it's only an RTS opcode.) Finally, the routine exits with the status register carry bit clear.

63883 SF98B

Resets the disk drive.

Preserves the status register and accumulator values on the stack for later retrieval, then restores the data channel for booting (logical file 13). The reset command (UI) is sent to the disk drive; then I/O channels are reset and the command channel (logical file 0) is closed as well. Finally, the status register and accumulator are restored to their original values before exiting.

63923 $F9B3

Loads additional boot sectors.

Begins by incrementing the sector number (194/SC2). If the count exceeds 20 (the maximum sector number for tracks 1-17), the sector number is reset to zero and the track number (193/$C1) is incremented. (Since a maximum of 255 additional sectors can be loaded, all boot sectors will be located on tracks 1-13.) The equivalent ASCII digits for the track and sector values are then added to the block read command in the buffer at 256-268/$0100-$010C, and the command is sent to the drive via the command channel (logical file 0). The 256 data bytes from the sector are then read via the data channel (logical file 13) and are stored starting at the address in 172-173/$AC-$AD in the bank specified in 198/SC6.

63995 $F9FB

Converts a byte value into two ASCII digits.
Returns two character codes representing digits for the decimal equivalent of the value in the accumulator upon entry. The left digit will be in the X register upon return, and the right digit will be in the accumulator. This routine works only for input values in the range 0-99/$00-$63.

64008 $FA08

Table of disk commands for booting.

This table holds the text for the disk block read command for booting (Ul:13 0 01 00), the initialize command (I), and the channel number command (#).

64023 $FA17 PRIMM

Handles PRIMM (print immediate) function.
(This routine has a jump table entry at 65405/$FF7D.)

Begins by stashing the accumulator and X and Y register values on the stack for later retrieval, then increments the return address on the stack and loads it into a working pointer at 206-207/$CE-$CF. The pointer thus contains the address of the location immediately following the JSR which called this routine. The byte at that location is retrieved and, unless its value is zero, is printed as a character. The routine then loops back to increment the address on the stack and retrieve another character, repeating until a zero byte is found. At that point, the original accumulator and X and Y register values are restored and the routine exits. Because the return address on the stack has been incremented, the routine will return to the address following the zero byte rather than to the address following the calling JSR.

It's very important always to call this routine (or its jump table entry) with JSR, not JMP, Only JSR puts a return address on the stack in the expected position; entering with JMP will cause the stack to be garbled, which will almost certainly result in a crash.

64064 $FA40 NMI

Handles NMI interrupts.
{This routine is the default target of the INMI indirect vector at 792/$0318.)

Begins by clearing the status register D (decimal) bit to insure that the system is not in decimal mode. Next, all CIA #2 interrupt sources are disabled, and the interrupt control register for that chip is checked to determine whether any CIA #2 source triggered the NMI interrupt. If so, the routine skips ahead to call the RS-232 handling routine. (CIA #2 interrupts are used to drive RS-232 output.) If the NMI was not triggered by a CIA #2 source, the routine checks whether the RUN/STOP key is pressed. If so, it's assumed that the NMI was triggered by pressing the RESTORE key, so the RUN/STOP-RESTORE sequence is performed. Otherwise, the RS-232 NMI handling routine [$E805] is called (if the NMI was not triggered by a CIA #2 source, this step will simply reenable any active RS-232 CIA #2 interrupt sources), and the routine exits via the common interrupt return [SFF33],

The RUN/STOP-RESTORE sequence consists of the following steps:

- A call to the RESTORE routine [$E056] to restore default Kernal indirect vectors.
- A call to the IOINIT routine [$E109] to reset all I/O chip registers to their default values (character definitions for 80-column video are normally not reinitialized).
- A call to the CINT routine [$C000] to restore screen editor variables to their default values (keyboard table pointers and programmable key definitions will normally be preserved).
- A jump back to BASIC through the restart vector (2560/$0A00), which normally points to the BASIC warm start entry vector, 16387/$4003.

You can modify NMI handling by redirecting the INMI vector (792-793/$0318-$0319) to a routine of your own. See the INMI entry and Appendix A for details.

64101 $FA65 IRQ

Handles IRQ interrupts.
(This routine is the default target of the IIRQ indirect vector at 788/$0314.)
Begins by clearing the status register D (decimal) bit to insure that the system is not in decimal mode. The screen editor IRQ routine [SC024] is called to handle screen mode settings, scan the keyboard, and blink the cursor. If the carry is clear upon return from that routine, indicating that the interrupt was triggered by a midscreen raster interrupt, the routine exits without

performing any further actions. Otherwise, the UDTIM routine [$F5F8] is called to update the jiffy timers; then the tape motor interlock handling routine [$EED0] is called. The interrupt control register for CIA #1 is read to clear any CIA #1 interrupts that might have occurred. Next, the initialization status flag (2564/$0A04) is checked. If the flag has bit 0 set to %1, indicating that BASIC has been initialized, the BASIC IRQ routine [$4006] is called to handle sprite movement and sound statements. Finally, the routine exits via the common interrupt return [$FF33].

You can modify IRQ handling by redirecting the IIRQ vector (788-789/$0314-$0315) to a routine of your own. See the IIRQ entry and Appendix A for details.

64128 $FA80

Standard keyboard decoding tables.

The following five 89-byte tables are used to translate the keyscan code generated by the SCNKEY routine [$C55D] into the corresponding character code. The appropriate table is selected according to the value in the shift key flag (211/$D3), and its address is loaded into the keyboard table pointer (204-205/$CC-$CD). Then the keyscan code is used as an offset into the table to retrieve the appropriate character code. The table starting addresses are as follows:

Address	Table
64128/$FA80	Standard (unshifted) and ALT
64217/$FAD9	SHIFT
64306/$FB32	Commodore
64395/$FB8B	CONTROL
64484/$FBE4	CAPS LOCK

Tables 9-1 through 9-5 show the character codes for each table. See 830-841/$033E-$0349 for information on how you can customize the tables. Values in the standard table (the one addressed in 830-831/$033E-$033F, normally 64128/$FA80), rather than the physical keyboard layout, determine which keys are treated as shift keys. Any key having an entry in that table with one of the following values is treated as the corresponding shift key:

l/$01	SHffT
2/$02	Commodore
4/$04	CONTROL
8/$08	ALT

This also means that these character codes cannot be returned by a key in the standard table. The special functions of the RUN/STOP key (halting a BASIC program, stopping a listing, aborting a save or load, working with RESTORE, and so forth) cannot be transferred to another key. That key has a character code of 3/$03 in the standard table, but for its special functions, the keyboard column containing RUN/STOP is scanned separately (see 63037/$F63D), and its character code is irrelevant.

Table 9-5 (the CAPS LOCK table) shows one of the more amusing bugs in the first version of 128 Kernal ROM. Note the entry for the Q key. That key is separated from the other alphabetic keys, so the programmer at Commodore who prepared this table overlooked it and neglected to change the entry when the other letter values were changed to their SHIFTed equivalents. The Q entry should be 2O9/$D1, which explains why the Q key is unaffected by CAPS LOCK.

64573-6S279 $FC3D-$FEFF Unused

All bytes in this unused area of Kernal ROM hold the value 255/$FF.

65280-65284 $FF00-$FF04 MMU Registers

The MMU configuration and load configuration registers appear here in all banks (refer to Chapter 8 for details).

65285 **$FF05** **JNMI**

Jump **to** NMI handler routine.
(This routine is the target of the processor NMI vector at 65530/$FFFA.)

Pushes the accumulator and *X* and Y register values onto the stack, then places the current MMU configuration register value onto the stack as well. (Thus, 128 interrupts place one more byte on the stack than do Commodore 64 interrupts.) The routine then configures the system for bank 15 and jumps through the INMI indirect vector at 792/$0318 to a routine to handle the NMI. The vector normally points to 64064/$FA40, but you can redirect it to a routine of your own for special handling. See the INMI entry for details. If normal processing is to continue following the interrupt handling, the handling routine should end with a jump to the routine at 65331/$FF33 to restore the processor registers and MMU configuration to their original values.

Table 9-1. Standard (Unshifted) and ALT Decoding Table 64128-64216/$FA80-$FAD8

Table index	Key	Character code	Table index	Key	Character code
0/$00	INST/DEL	20/$14	45/$2D		58/S3A
l/$01	RETURN	13/SOD	46/$2E	@	64/$40
2/S02	CRSR w	29/$lD	47/$2F		44/$2C
3/$03	F7	136/$88	48/530	E	92/$5C
4/$04	Fl	133/$85	49/$31	*	42/52A
5/$05	F3	134/586	50/$32		59/S3B
6/506	F5	135/$87	51/$33	CLR/HOME	19/$13
7/$07	CRSR I	17/$11	52/$34	right SHIFT	1/501
8/$08	3	51/$33	53/$35	—	61/$3D
9/509	W	87/$57	54/$36		94/$5E
10/$0A	A	65/541	55/$37	/	47/S2F
11/SOB	4	52/S34	56/$38	1	49/S31
12/50C	Z	90/$5A	57/$39	<	95/$5F
13/$0D	S	83/$53	58/S3A	CONTROL	4/$04
14/$0E	E	69/$45	59/$3B	2	50/$32
15/$0F	left SHIFT	l/$01	60/$3C	space bar	32/520
16/$10	5	53/535	61/$3D	Commodore	2/S02
17/S11	R	82/$52	62/$3E	Q	81/$51
18/$12	D	68/$44	63/$3F	RUN/STOP	3/$03
19/$13	6	54/S36	64/$40	HELP	132/$84
20/$14	C	67/$43	65/$41	8 (keypad)	56/S38
21/$15	F	70/$46	66/$42	5(keypad)	53/$35
22/$16	T	84/$54	67/$43	TAB	9/$09
23/$:7	X	88/$58	68/544	2 (keypad)	50/S32
24/$18	7	55/537	69/$45	4 (keypad)	52/$34
25/$19	Y	89/559	70/$46	7 (keypad)	55/$37
26/51A	G	71/$47	71/$47	1 (keypad)	49/$31
27/$IB	S	56/$38	72/S48	ESC	27/S1B
28/$lC	B	66/$42	73/$49	+ (keypad)	43/S2B
29/$ ID	ll	72/548	74/$4A	— (keypad)	45/$2D
30/S1E	U	85/$55	75/54B	LINE FEED	10/$0A
31/S1F	V	86/$56	76/$4C	ENTER	13/$0D
32/$20	9	57/S39	77/$4D	6(keypad)	54/S36
33/$21	I	73/549	78/$4E	9 (keypad)	57/539
34/S22	J	74/$4A	79/$4F	3 (keypad)	51/533
35/523	0	48/$30	80/$50	ALT	8/$08
36/$24	M	77/$4D	81/$51	0(keypad)	48/$30
37/$25	K	75/54B	82/$52	. (keypad)	46/S2E
38/$26	O	79/S4F	83/$53	t (cursor)	145/$91
39/527	N	78/S4E	84/$54	1 (cursor)	17/$11
40/528	+	43/52B	85/555	*- (cursor)	157/S9D
41/$29	P	80/$50	86/$56	- (cursor)	29/S1D
42/$2A	L	76/$4C	87/$57	NO SCROLL	255/SFF
43/$2B		45/$2D	88/$58	no key pressed	255/SFF
44/$2C		46/52E			

Table 9-2. SHIFT Decoding Table 64217-64305/$FAD9-$FB31

Table index	Key	Character code
0/S00	INST/DEL	148/S94
l/$01	RETURN	141/$8D
2/$0[2]	CRSR ~	157/S9D
3/$03	F7	140/$8C
4/S04	Fl	137/$89
5/$05	F3	138/$8A
6/S06	F5	139/$8B
7/$07	CRSR 1	145/S91
8/S08	3	35/$23
9/$09	W	215/SD7
10/SOA	A	193/$C1
ll/$0B	4	36/$24
12/$0C	Z	218/SDA
13/$0D	S	211/SD3
14/$0E	E	197/SC5
15/$0F	left SHIFT	l/$01
16/S10	5	37/$25
17/111	R	210/SD2
18/$12	D	196/SC4
19/$13	6	38/$26
20/$14	C	195/$C3
21/$15	F	198/SC6
22/$16	T	212/SD4
23/$17	X	216/SD8
24/$18	7	39/$Z7
Z5/S19	Y	217/$D9
26/$1A	G	199/$C7
27/$lB	8	40/$28
28/$1C	B	194/SC2
29/$ ID	H	200/$C8
30/$1E	U	213/$D5
31/$1F	V	214/$D6
32/$20	9	41/$29
33/$21	[	201/$C9
34/$22	J	202/$CA
35/123	0	48/530
36/$2*	M	205/$CD
37/$25	K	203/5CB
38/$26	0	207/5CF
39/$27	N	206/5CE
40/$28	+	219/$DB
41/$29	P	208/5D0
42/$2A	L	204/$CC
43/$2B	-	221/$DD
44/$2C		62/$3E
45/$2D		91/55B
46/$2E	@	186/5BA
47/$2F		60/53C
48/$30	£	169/$A9
49/$31	*	192/$C0
50/$32		93/$5D
51/$33	CLR/HOME	147/$93
52/534	right SHIFT	1/501
53/535		61/53D
54/$36		222/$DE
55/$37	/	63/$3F
56/$38	1	33/$21
57/539	4	95/$5F
58/53A	CONTROL	4/$04
59/S3B	2	34/$22
60/$3C	space bar	160/SA0
61/$3D	Commodore	2/$02
62/$3E	Q	209/5D1
63/53F	RUN/STOP	131/483
64/540	HELP	132/$84
65/541	8(keypad)	56/$38
66/$42	5 (keypad)	53/S35
67/543	TAB	24/$18
68/544	2 (keypad)	50/$32
69/545	4 (keypad)	52/$34
70/546	7(keypad)	55/$37
71/$47	1 (keypad)	49/$31
72/548	ESC	27/$lB
73/549	+ (keypad)	43/$2B
74/54A	- (keypad)	45/$2D
75/54B	LINE FEED	10/$0A
76/$4C	ENTER	141/58D
77/$4D	6(keypad)	54/$36
78/$4E	9(keypad)	57/$39
79/$4F	3(keypad)	51/533
80/$50	ALT	8/508
81/$51	0(keypad)	48/530
82/$52	. (keypad)	46/52E
83/$53	T (cursor)	145/$91
84/$54	1 (cursor)	17/511
85/$55	- (cursor)	157/$9D
86/$56	- (cursor)	29/51D
87/$57	NO SCROLL	255/$FF
88/$58	no key pressed	255/$FF

Table 9-3. Commodore Decoding Table 64306-64394/$FB32-$FB8A

Table index	Key	Character code
0/$00	1NST/DEL	148/$94
1/S01	RETURN	141/SSD
2/502	CRSR —	157/$9D
3/$03	F7	U0/$8C
4/$04	Fl	137/$89
5/$05	F3	138/$8A
6/$06	F5	139/$8B
7/$07	CRSR I	145/$91
8/$08	3	150/596
9/$09	W	179/$B3
10/50A	A	176/$B0
11/50B	4	151/S97
12/50C	Z	173/$AD
13/SOD	S	174/$AE
14/$0E	E	177/5B1
15/$0F	left SHIFT	**1/S01**
16/$10	5	152/$98
17/511	R	178/$B2
18/$12	D	172/$AC
19/$13	6	153/$99
20/$14	C	188/$BC
21/515	F	187/$BB
22/516	T	163/$A3
23/$17	X	189/5BD
24/$18	7	154/$9A
25/$19	Y	183/5B7
26/51A	G	165/SA5
27/51B	8	155/59B
28/51C	B	191/SBF
29/$ ID	H	180/$B4
30/$IE	U	184/$B8
31/$1F	V	190/$BE
32/$20	9	41/529
33/$21	I	162/$A2
34/$22	I	181/$B5
35/$23	0	48/$30
36/$24	M	167/$A7
37/$25	**K**	161/$A1
3S/$26	**0**	185/$B9
39/$27	**N**	170/$AA
40/$28	**+**	166/$A6
41/S29	**P**	175/SAF
42/$2A	**L**	182/5B6
43/$2B	**-**	220/5DC
44/S2C		62/$3E
45/$2D		91/$5B
46/S2E	@	164/SA4
47/52F		60/$3C
48/$30	E	168/5A8
49/$31		223/$DF
50/$32		93/55D
51/$33	CLR/HOME	147/S93
52/534	right SHIFT	l/$01
53/$35	=	61/S3D
54/$36	*	222/5DE
55/$37	/	63/53F
56/$38	1	129/$81
57/539	4	95/$5F
58/$3A	CONTROL	4/$04
59/$3B	2	149/595
60/$3C	space bar	160/$A0
61/S3D	Commodore	2/$02
62/53E	Q	I71/5AB
63/$3F	RUN/STOP	3/$03
64/$40	HELP	132/584
65/$41	8(keypad)	56/538
66/542	5 (keypad)	53/$35
67/543	TAB	24/$18
68/$44	2 (keypad)	50/S32
69/$45	4 (keypad)	52/534
70/S46	7 (keypad)	55/537
71/$47	1 (keypad)	49/$31
72/548	ESC	27/$IB
73/$49	+ (keypad)	43/$2B
74/$4A	- (keypad)	45/$2D
75/S4B	LINE FEED	10/$0A
76/S4C	ENTER	141/58D
77/54D	6 (keypad)	54/J36
78/$4E	9 (keypad)	57/539
79/$4F	3(keypad)	51/533
80/S50	ALT	8/508
81/$51	0(keypad)	48/530
82/$52	.(keypad)	46/$2E
83/$53	T (cursor)	145/591
84/$54	•I (cursor)	17/$11
85/S55	* (cursor)	157/$9D
86/$56	- (cursor)	29/$ ID
87/$57	NO SCROLL	255/$FF
88/558	no key pressed	255/5FF

Table 9-4. CONTROL Decoding Table

643 95-64483/$FB8B-$FBE3

Table index	Key	Character code
0/500	INST/DEL	255/5FF
l/$01	RETURN	255/$FF
2/502	CRSR —	255/SFF
3/503	F7	255/$FF
4/$04	Fl	255/$FF
5/$05	F3	255/$FF
6/$06	F5	255/SFF
7/S07	CRSR I	255/$FF
	3	28/51C
9/$09	W	23/517
10/$0A	A	l/$01
ll/$0B	4	159/$9f
12/$0C	Z	26/51A
13/$0D	S	19/513
14/$0E	E	5/$05
15/$0F	left SHIFT	255/$FF
16/S10	5	156/$9C
17/$11	R	18/512
1S/S12	D	4/$04
19/$13	6	30/S1E
20/$14	C	3/$03
21/S15	F	6/$06
22/$16	T	20/514
23/$17	X	24/518
24/$18	7	31/51F
25/$19	Y	25/519
26/$1A	G	7/$07
27/$lB	8	158/$9E
28/$lC	B	2/S02
29/$ ID	H	8/$08
30/$IE	U	21/$15
31/51F	V	22/516
32/$20	9	18/512
33/$21	1	9/$09
34/$22	r	10/50A
35/S23	0	146/$92
36/$24	M	13/50D
37/525	K	ll/$0B
38/526	O	15/50F
39/527	N	14/SOE
40/$28	+	255/$FF
41/529	P	16/$10
42/52A	L	12/SOC
43/S2B	-	255/5FF
44/S2C		255/5FF
45/$2D		
46/$2E		
47/S2F		
48/S30		
49/$31		
50/$32		
51/S33		
52/S34		
53/$35		
54/$36		
55/537		
56/$38		
57/$39		
58/$3A		
59/S3B		
60/$3C		
61/$3D		
62/$3E		
63/53F		
64/$40		
65/$41		
66/$42		
67/$43		
68/$44		
69/$45		
70/$46		
71/$47		
72/$48		
73/$49		
74/54A		
75/$4B		
76/$4C		
77/$4D		
78/$4E		
79/$4F		
80/550		
81/S51		
82/$52		
83/S53		
84/$54		
85/S55		
86/$56		
87/$57		
88/$58		

Key	Character code
b	27/51B
	0/$00
	255/$FF
£	28/$lC
	255/$FF
;	29/$lD
CLR/HOME	255/$FF
right SHIFT	255/$FF
=	31/51F
	30/51E
/	255/$FF
1	144/S90
4	6/$06
CONTROL	255/$FF
2	5/505
space bar	255/$FF
Commodore	255/$FF
Q	17/$11
SUN/STOP	255/$FF
HELP	132/$84
8(keypad)	56/$38
5 (keypad)	53/535
TAB	24/$18
2(keypad)	50/532
4 (keypad)	52/534
7(keypad)	55/$37
1(keypad)	49/$31
ESC	27/51B
+ (keypad)	43/52B
- (keypad)	45/52D
LINE FEED	10/50A
ENTER	141/$8D
6(keypad)	54/$36
9(keypad)	57/$39
3(keypad)	51/533
ALT	8/$08
0(keypad)	48/530
.(keypad)	46/$2E
f (cursor)	145/$91
i (cursor)	17/$11
- (cursor)	157/$9D
•• (cursor)	29/51D
NO SCROLL	255/$FF
no key pressed	255/$FF

Table 9-5. CAPS LOCK Decoding Table 64484-64572/$FBE4-$FC3C

Table index	Key	Character code
0/$00	INST/DEL	20/$14
l/$01	RETURN	13/$0D
2/E02	CRSR <-	29/$ ID
3/$03	F7	136/$88
4/$04	Fl	133/$85
5/$05	F3	134/S86
6/$06	F5	135/$87
7/S07	CRSR !	**17/S11**
8/$08	3	51/$33
9/$09	W	215/$D7
10/$0A	A	193/SCI
11/$OB	4	52/$34
12/SOC	Z	218/SDA
13/SOD	S	211/$D3
H/$0E	E	197/$C5
15/SOF	left SHIFT	l/$01
16/$10	5	53/135
17/$11	R	210/SD2
1S/S12	D	196/$C4
19/$13	6	54/$36
20/$14	C	195/$C3
21/$15	F	198/$C6
22/$16	T	212/$D4
23/$17	X	216/$D8
24/$18	7	55/$37
25/$19	Y	217/$D9
26/$lA	G	199/SC7
27/$lB	8	56/$38
28/$lC	B	194/$C2
29/$lD	H	200/$C8
30/$1E	U	213/$D5
31/$1F	V	2U/SD6
32/$20	9	57/S39
33/S21	I	201/SC9
34/$22	J	202/SCA
35/$23	0	48/S30
36/$24	M	205 /$CD
37/$25	K	203/$CB
38/$26	O	207/$CF
39/$27	N	206/$CE
40/$28	+	43/$2B
41/$29	P	208/SDO
42/$2A	L	204/$CC
43/$2B	-	45/S2D
44/$2C		46/S2E

Table index	Key	Character code
45/$ 2D	:	58/S3A
46/S2E	@	64/$40
47/$2F		44/$2C
48/$30	£	92/$5C
49/$31	•	42/$2A
50/$32		59/$3B
51/$33	CLR/HOME	19/313
52/$34	right SHIFT	l/$01
53/$35	—	61/$3D
54/$36	*	94/S5E
55/$37	/	47/$2F
56/$38	1	49/$31
57/$39	4	95/$5F
58/$3A	CONTROL	4/$04
59/$3B	2	50/$32
60/$3C	space bar	32/S20
61/$3D	Commodore	2/$02
62/$3F_	Q	81/$51
63/$3F	RUN/STOP	3/$03
64/$40	HELP	132/$84
65/$41	8 (keypad)	56/$3S
66/$42	5 (keypad)	53/$35
67/$43	TAB	9/$09
6S/$44	2 (keypad)	50/S32
69/$45	4 (keypad)	52/$34
70/$46	7 (keypad)	55/$37
71/$47	1(keypad)	49/$31
72/$48	ESC	27/$IB
73/$49	+ (keypad)	43/$2B
74/$4A	- (keypad)	45/$2D
75/$4B	LINEFEED	10/$0A
76/$4C	ENTER	13/$0D
77/$4D	6(keypad)	54/$36
78/S4E	9(keypad)	57/$39
79/$4F	3(keypad)	51/$33
80/$50	ALT	8/$08
81/S51	0(keypad)	48/$30
82/$52	. (keypad)	46/$2E
83/$53	! (cursor)	145/$91
84/S54	*I* (cursor)	17/$11
85/$55	*• (cursor)	157/$9D
86/$56	-• (cursor)	29/$lD
87/$57	NO SCROLL	255/$FF
88/$58	no key pressed	255/SFF

This routine is also copied into all RAM banks to handle interrupts which occur when the system is configured for a bank where Kernal ROM is not visible.

65303 $FF17 JIRQ

Jump to **IRQ** or **BRK** handler routine.
(This routine is the target of the processor IRQ/BRK vector at 65534/$FFFE.)
Pushes the accumulator and X and Y register values onto the stack, then places the current MMU configuration register value onto the stack as well. (Thus, 128 interrupts place one more byte on the stack than do Commodore 64 interrupts.) The routine then configures the system for bank 15 and reads the processor status register value on the stack (the interrupt automatically causes the status register value and return address to be placed on the stack before this routine is called). If the B bit (bit 4) of the status register is set to %1, indicating that the interrupt is the result of the execution of a BRK instruction (a software interrupt), the routine jumps through the IBRK indirect vector at 790/$0316 to a routine to handle the BRK. That vector normally points to 45059/$B003, the break entry point into the machine language monitor. If the B status bit is %0, a hardware interrupt has been triggered by an external source, so the routine jumps through the IIRQ indirect vector at 788/$0314. That vector normally points to the handling routine at 64101/$FA65. In either case, you can redirect the vector to a routine of your own for special handling. See the IBRK and IIRQ entries for details.

If normal processing is to continue following the interrupt handling, the handling routine should end with a jump to the routine at 65331/$FF33 to restore the processor registers and MMU configuration to their original values. For example, the normal IRQ service routine exits in this manner, so processing resumes unaffected after a standard IRQ interrupt. However, the BRK service routine does not return, since normal processing is halted when the monitor is entered.

This routine is also copied into all RAM banks to handle interrupts which occur when the system is configured for a bank where Kernal ROM is not visible.

65331 8FF33 CRTI

Common exit routine for all interrupt routines.
Retrieves the MMU configuration register value from the stack and restores the system to its original bank setting, then restores the Y and X register and accumulator values from the stack. Processing resumes at the instruction following the one during which the interrupt has occurred.

65341 $FF3D JRESET

Jump to reset handler routine.
(This routine is the target of the processor RESET vector at 65532/$FFFC.)
Sets the system for the bank 15 configuration, then jumps to the RESET routine [$E000].

This routine is also copied into all RAM banks to handle any reset which might occur when the system is configured for a bank where Kernal ROM is not visible.

65349-65350 $FF45-$FF46 Unused

Two unused bytes, filled with the value 255/SFF.

New 128 Kernal Jump Table

Locations 65351-65407/$FF47-$FF7F are a table of jump vectors to routines found in 128 ROM, but not in previous versions of the Kernal for earlier Commodore computers. As with the other jump tables, each table entry consists of a JMP opcode (76/$4C) followed by the address of the target routine.

65351 SFF47 JSPIN_SPOUT

Entry point for the Kernal SPIN-SPOUT routine at 58875/$E5FB, which sets up the serial bus for fast communications mode. Enter with the status register carry bit clear to establish fast serial input or with the bit set to establish fast serial output. Unless you are writing a custom data transfer routine, it's not necessary to call this routine explicitly. All the standard serial I/O routines already include this setup step.

65354 **$FF4A** **JCLOSE^ALL**

Entry point for the Kernal CLOSE_ALL routine at 62013/$F23D, which closes all files currently opened to a specified

device. (This is different from the Kernal CLALL routine f$FFE71, which merely resets the number of open files to zero without explicitly closing any open files.) Enter the routine with the accumulator holding the number of the device on which files are to be closed. If the specified device is the current input or output device, the channel will be reset to the default device {screen or keyboard). If all files to the device have been successfully closed, the status register carry bit will clear upon return. A set carry bit indicates that a device error has occurred.

65357 **$FF4D JC64_MODE**

Entry point for the Kernal C64_MODE routine at 57931/ $E24B, which switches the system immediately to 64 mode. To get back to 128 mode, it is necessary to reset the computer or to turn it off and back on.

65360 $FF50 JDMA_CALL

Entry point for the Kernal DMA_CALL routine at 63397/ $F7A5, which passes commands to a DMA (direct memory access) controller. The DMA device will take control of the system to perform the requested command. The only DMA peripherals currently available for the 128 are the 1700 and 1750 Memory Expansion Modules, controlled by a DMA chip known as the REC (see Chapter 8 for more information). Call the routine with the Y register holding the command for the DMA device and the X register holding the bank number for the operation. Additional setup steps may be required, depending on the command.

65363 SFF53 JBOOT_CALL

Entry point for the Kernal BOOT_CALL routine at 63632/$F890, which attempts to load and execute boot sectors from a specified disk drive. Call the routine with the X register holding the device number for the desired disk drive (usually 8) and the accumulator holding the character code corresponding to the desired drive number—not the actual drive number. (For example, the single drive in 1541 and 1571 units is designated drive 0, so you would use 48/$30, the character code for zero.) If the specified drive is not present or turned off, or tf the disk in the drive does not contain valid boot sectors, the

routine will return with the status register carry bit set. If boot sectors are found and executed, the boot code may or may not return to the calling routine.

65366 8FF56 JPHOENIX

Entry point for the Kernal PHOENIX routine at 63591/$F867, which initializes any installed-function ROMs and attempts to boot a disk from the default drive (drive 0 of device 8). The Kernal reset routine [$E000] records the presence of function ROMs in cartridges plugged into the expansion port or into the spare ROM socket on the system circuit board, but it does not initialize them unless they are autostarting (unless their cartridge ID byte holds the value 1). This routine initializes all recorded function ROMs by calling their cold start entry addresses (the first address of the memory slot for the ROM). If any ROMs are present, they may or may not return to this routine, depending on the initialization steps performed. If all ROMs return after initialization (or if no ROMs are present), the routine attempts to boot a disk in drive 0 of device 8.

65369 $FF59 JLKUPLA

Entry point for the Kernal LKUPLA routine at 63389/$F79D, which checks whether a specified logical file number is currently in use. Call the routine with the accumulator holding the desired logical file number. If that number is used for a currently open file, the status register carry bit will be clear upon return. The accumulator will still hold the logical file number; the X register will hold the associated device number, and the Y register the secondary address. However, if the logical file number is not currently used, the carry bit will instead be set upon return (the logical file number will still be in the accumulator).

65372 $FF5C JLKUPSA

Entry point for the Kernal LKUPSA routine 63366/$F786, which checks whether a specified secondary address is currently in use. Call the routine with the Y register holding the desired secondary address. If that number is used for a currently open file, the status register carry bit will be clear upon return. The Y register will still hold the secondary address; the accumulator will hold the associated logical file number, and

the X register the device number. However, if the secondary address is not currently used, the carry bit will instead be set upon return (the secondary address value will still be in the Y register).

65375 $FF5F JSWAPPER

Calls the Kernal SWAPPER routine's screen editor jump table entry at 49194/$C02A. The routine switches active screen displays by exchanging the active and inactive screen editor variable tables, tab stop bitmaps, and line link bitmaps, and by toggling bit 7 of the active screen flag (215/$D7). The routine doesn't physically turn either video chip on or off; both chips remain enabled. Instead, the routine determines which display will be the recipient of subsequent printing. (Only the active screen will have a "live" cursor.)

65378 $FF62 JDLCHR

Calls the Kernal INIT80 routine's screen editor jump table entry at 49191/$C027. The routine copies character shape data from ROM into the 8563 80-column video chip's private block of RAM (the 8563 has no character ROM of its own).

65381 $FF65 JPFKEY

Calls the Kernal KEYSET routine's screen editor jump table entry at 49185/$C021. The routine assigns a new definition to one of the ten programmable function keys (F1-F8, SHIFT-RUN/STOP, and HELP). Call the routine with the accumulator holding the address of a three-byte string descriptor in zero page, the X register containing the key number (1-10), and the Y register containing the length of the new definition string. The descriptor in zero page should consist of the two-byte address of the new definition string (in standard low-byte/high-byte order), followed by the bank number where the definition string is located. The key number is not checked for validity; if you specify a value outside the acceptable range, you may garble existing definitions. Upon return, the carry bit will be clear if the new definition has been successfully added, or it will be set if there is insufficient room in the definition table for the new definition.

65384 $FF68 JSETBNK

Entry point for the Kernal JSETBNK routine at 63295/$F73F, which establishes the current bank from which data will be read or to which data will be written during load/save operations, as well as the bank where the filename for the I/O operations can be found. Call the routine with the accumulator holding the bank number for data and the X register holding the bank for the filename. All register values are preserved during this routine.

65387 $FF6B JGETCFG

Entry point for the Kernal GETCFG routine at 63468/$F7EC, which translates a bank number into the MMU register setting which will configure the system for that bank. Call the routine with the X register containing the bank number (0-15). Upon return, the accumulator will hold the corresponding MMU configuration register setting value. The routine does not check the validity of the bank number input. If you specify a number outside the acceptable range, the routine will return a meaningless value.

65390 SFF6E JJSRFAR

Entry point for the Kernal JSRFAR routine at 717/$02CD, which jumps to a subroutine in a specified bank and returns to the calling routine in bank 15. Prior to calling this routine, you must load location 2/$02 with the bank number (0-15) of the target routine and locations 3-4/$03-$04 with the address of the target routine (in high-byte/low-byte order, the opposite of the normal arrangement). You should also load location 5/$05 with the value you want placed in the status register when the target routine is called. (The behavior of many routines is influenced by the status register setting, particularly the state of the carry bit. Load 5/$05 with the value 0/$00 if the target routine is to be called with carry clear, or with l/$01 if it is to be called with carry set,) If you wish to pass other register values to the routine you will be calling, store the desired accumulator value in location 6/$06, the X register value in 7/$07, and the Y register value in 8/$08. Upon return, location 5/$05 will hold the status register value at the time of exit, 6/$06 will hold the accumulator value, 7/$07 will hold the X register value, 8/$08 will hold the Y register

value, and 9/$ 09 will hold the stack pointer value. The system is always reconfigured for bank 15 upon exit.

65393 $FF71 JJMPFAR

Entry point for the Kernal JMPFAR routine at 739/S02E3, which jumps to a routine in a specified bank, with no return to the calling bank. Prior to calling this routine, you must load location 2/$02 with the bank number (0-15) of the target routine and locations 3-4/$03-$04 with the address of the target routine (in high-byte/low-byte order, the opposite of the normal arrangement). You should also load location 5/$05 with the value you want placed in the status register when the target routine is entered. (The behavior of many routines is influenced by the status register setting, particularly the state of the carry bit. Load 5/$05 with the value 0/$00 if the target routine is to be entered with carry clear or with l/$01 if it is to be entered with carry set.) If you wish to pass other register values to the routine you will be calling, store the desired accumulator value in location 6/$06, the X register value in 7/$07, and the Y register value in 8/$08.

65396 SFF74 JINDFET

Entry point for the Kernal INDFET calling routine at 63440/$F7DO, which retrieves a byte from a specified bank. Prior to calling this routine, you must load a two-byte zero-page pointer with the address of the location from which the byte is to be retrieved (or the base location if a series of bytes are to be retrieved). Call the routine with the accumulator holding the address of the zero-page pointer, the X register holding the bank number (0-15) for the target location, and the Y register holding an offset value which will be added to the address in the pointer to determine the location from which the byte is to be loaded. (Load Y with zero if no offset is desired.) Upon return, the accumulator will hold the byte from the specified address. The value in the Y register will be preserved during the routine. If you are retrieving data from a series of locations, it is necessary to reload the accumulator [a]nd X registers with the pointer address and bank number before every call to this routine, but you can read up to 256 sequential locations without changing the address in the zero-page pointer by simply incrementing the Y register between calls.

65399 $FF77 JINDSTA

Entry point for the Kernal INDSTA calling routine at 63450/$F7DA, which stores the accumulator contents at an address in a specified bank. Prior to calling this routine, you must load a two-byte zero-page pointer with the address of the location at which the byte is to be stored (or the base location if a series of bytes is to be stored); then store the address of this pointer in location 697/$02B9. Call the routine with the accumulator holding the byte to be stored, the X register holding the bank number (0-15) for the target location, and the Y register holding an offset value which will be added to the address in the pointer to determine the location in which the byte is to be stored. (Load Y with zero if no offset is desired.) Upon return, the accumulator will still hold the byte value. The value in the Y register will also be preserved. If you are writing data to a series of locations, it is necessary to reload the X register with the bank number before every call to this routine, but you can write to up to 256 sequential locations without changing the address in the zero-page pointer by simply incrementing the Y register between calls.

65402 $FF7A JINDCMP

Entry point for the Kemal INDCMP calling routine at 63459/$F7E3, which compares the accumulator contents against the contents of a location in a specified bank. Prior to calling this routine, you must load a two-byte zero-page pointer with the address of the location with which the byte is to be compared (or the base location if a series of bytes are to be compared); then store the address of this pointer in location 712/$02C8. Call the routine with the accumulator holding the byte to be compared, the X register holding the bank number (0-15) for the target location, and the Y register holding an offset value which will be added to the address in the pointer to determine the location with which the byte is to be compared. (Load Y with zero if no offset is desired.) Upon return, the accumulator will still hold the byte value, and the status register N, Z, and C (carry) bits will reflect the result of the comparison. The value in the Y register will also be preserved. If you are comparing a series of locations, it is necessary to reload the X register with the bank number before every call to this routine, but you can compare up to 256 sequential locations without

changing the address in the zero-page pointer by simply incrementing the Y register between calls.

65405 $FF7D JPRIMM

Entry point for the Kernal PRIMM routine 64023/$FA17, which prints the string of character codes immediately following the JSR to this routine. You must always call this routine with JSR, never with JMP. Only JSR places the required address information on the stack. The end of the character code string is indicated by a byte containing 0/$00 (the routine continues printing bytes as character codes until a zero byte is encountered). When the ending marker is found, the routine returns to the address immediately following the zero byte.

65408 $FF80

Some Commodore literature suggests that this location will be updated in future revisions of the Kernal ROM to indicate the revision number. For the original version of the 128 Kernal, this location contains the value 0/$00.

Standard Commodore Jump Table

Locations 65409-65525/$FF81-$FFF5 are a table of jump vectors to the standard Commodore Kernal routines, most of which have been part of previous versions of the Kernal. As with the other 128 jump tables, each table entry consists of either a direct JMP opcode (76/$4C) followed by the address of the target routine or an indirect JMP opcode (108/$6C) followed by the address of a Kernal indirect vector containing the address of the target routine.

65409 SFF81 JCINT

Calls the Kernal CINT routine's screen editor jump table entry at 49152/$C000. This routine initializes all the RAM locations used by the screen editor, clears both displays, and redirects printing to the display indicated by the position of the 40/80 DISPLAY key.

65412 $FF84 JIOINIT

Entry point for the Kemal IOINIT routine at 57609/$E109, which initializes the CIA, SID, and 40- and 80-column video chips, along with related RAM locations.

65415 SFF87 JRAMTAS

Entry point for the Kernal RAMTAS routine at 57491/$E093, which clears zero-page RAM (locations 2-255/$02-$FF) and initializes Kernal memory pointers. The routine also sets the BASIC restart vector (2560/$0A00) to point to BASIC'S cold start entry address, 16384/$4000.

65418 SFF8A JRESTOR

Entry point for the Kernal RESTOR routine at 57430/$E056, which restores the Kernal indirect vectors at 788-819/ $0314-$0333 to their default values.

65421 $FF8D JVECTOR

Entry point for the Kernal VECTOR routine at 57435/SE05B, which loads or stores the Kernal indirect vectors at 788-819/ $0314-$0333. When this routine is called, the X and Y registers should be loaded with the address of a 32-byte table (low byte in X, high byte in Y). If the status register carry bit is clear when the routine is called, the vectors will be loaded with the values from the table. If carry is set, the 16 two-byte address values currently in the vectors will be copied to the table.

65424 $FF90 JSETMSG

Entry point for the Kernal SETMSG routine at 63324/$F75C, which sets the value of the Kernal message flag (157/$9D). Call the routine with the accumulator holding the desired flag value. Refer to the entry for 157/$9D for information on flag values.

65427 $FF93 JSECOND

Entry point for the Kernal SECOND routine at 58578/$E4D2, which sends a secondary address to a serial device which has been commanded to listen. The value in the serial status flag (144/$90) upon return will indicate whether the operation is successful.

65430 $FF96 JTKSA

Entry point for the Kernal TKSA routine at 58592/$E4E0, which sends a secondary address to a serial device which has been commanded to talk. The value in the serial status flag

Cl44/$90) upon return will indicate whether the operation is successful.

65433 $FF99 JMEMTOP

Entry point for the Kernal MEMTOP routine at 63331/$F763, which reads or sets the value of the Kernal's top-of-memory pointer (2567-2568/$0A07-$0AO8). To read the pointer, call the routine with the carry flag set; the pointer value will be returned in the X and Y registers (low byte in X, high byte in Y). To set the pointer, call the routine with the carry flag clear and with the X and Y registers containing the low and high bytes, respectively, of the desired pointer value.

65436 $FF9C JMEMBOT

Entry point for the Kernal MEMBOT routine at 63346/$F772, which reads or sets the value of the Kernal's bottom-of-memory pointer (2565-2566/$0A05-$0A06). To read the pointer, call the routine with the carry flag set; the pointer value will be returned in the X and Y registers (low byte in X, high byte in Y). To set the pointer, call the routine with the carry flag clear and with the X and Y registers containing the low and high bytes, respectively, of the desired pointer value.

65439 $FF9F JKEY

Calls the Kernal SCNKEY routine's screen editor jump table entry at 49170/$C012. The routine scans the keyboard matrix to determine which keys, if any, are currently pressed. The character code for the key currently pressed is loaded into the keyboard buffer at 842/$034A, from which it can be retrieved using the Kernal GETIN routine [$FFE4], The matrix code of the keypress read during this routine can also be read in location 212/$D4, and the status of the shift keys can be read in 211/$D3. Since this routine is normally called as part of the standard IRQ service routine, it's not usually necessary to call it explicitly to read the keyboard.

65442 $FFA2 JSETTMO

Entry point for the Kernal SETTMO routine at 63327/$F75F, which sets the value of the IEEE timeout flag <2574/$0A0E). This routine is superfluous, since the flag isn't used by any 128 ROM routine. It is present merely to maintain consistency with previous versions of the Kernal.

65445 $FFA5 JACPTR

Entry point for the Kernal ACPTR at routine 58430/$E43E, which retrieves a byte from a serial device. The success of the operation will be indicated by the value in the serial status flag (144/$90) upon return. If the operation is successful, the accumulator will hold the byte received from the device. The value in the Y register will be preserved. For the routine to function properly, the serial device must currently be a talker on the serial bus, which requires a number of setup steps. Generally, it's preferable to use higher level Kernal calls such as BASIN [$FFCF] instead.

65448 $FFA8 JCIOUT

Entry point for the Kernal CIOUT routine at 58627/$E503, which sends a byte to a serial device. Call the routine with the accumulator holding the byte to be sent. All register values are preserved during this routine. The success of the operation will be indicated by the value in the serial status flag (144/$90) upon return. For the routine to function properly, the serial device must currently be a listener on the serial bus, which requires a number of setup steps. However, if you have already performed all the preparatory steps necessary for BSOUT [$FFD2] to a serial device (SETLFS, SETBNK, SETNAM, OPEN, and CKOUT), you can freely substitute CIOUT for BSOUT since for a serial device BSOUT simply jumps to the CIOUT routine.

65451 $FFAB JUNTLK

Entry point for the Kernal UNTLK routine at 58645/$E515, which sends an UNTALK command to all devices on the serial bus. Any devices which are currently talkers will cease sending data.

65454 $FFAE JUNLSN

Entry point for the Kernal UNLSN routine at 58662/$E526, which sends an UNLISTEN command to all devices on the serial bus. Any devices which are currently listeners will cease receiving data.

65457 **$FFB1** **JLISTN**

Entry point for the Kernal LISTN routine at 58174/$E33E, which sends a LISTEN command to a specified serial device. Call the routine with the accumulator holding the device number (4-31) of the desired serial device. The success of the operation will be indicated by the value in the serial status flag (144/S90) upon return.

65460 $FFB4 JTALK

Entry point for the Kernal TALK routine at 58171/$EE3B, which sends a TALK command to a specified serial device. Call the routine with the accumulator holding the device number (4-31) of the desired serial device. The success of the operation will be indicated by the value in the serial status flag (144/$90) upon return.

65463 8FFB7 JREADSS

Entry point for the Kernal READSS routine at 63300/$F744 (called READST in previous versions of the Kernal), which returns the status of the previous I/O operation. The status value will be in the accumulator upon return; the contents of the X and Y registers are unaffected. If the current device number is 2 (indicating an RS-232 operation), the status value is retrieved from the RS-232 status flag (2580/$0A14), and the flag is cleared. Otherwise, the status value is retrieved from the tape/serial status flag (144/$90). That flag is not cleared after being read.

65466 **$FFBA** JSETLFS

Entry point for the Kernal SETLFS routine at 63288/$F7E8, which assigns the logical file number, device number, and secondary address for an I/O operation. Call the routine with the accumulator holding the logical file number, the X register holding the device number, and the Y register holding the secondary address. All register values are preserved during the routine.

65469 SFFBD JSETNAM

Entry point for the Kernal SETNAM routine at 63281/SF731, which assigns the length and address of the filename for an I/O operation. Call the routine with the accumulator holding

the length of the filename and the X and Y registers holding the address of the first character of the name (low byte in X, high byte in Y). The Kernal SETBNK routine [$FF68] must be used to specify the appropriate bank for the filename address. If no name is used for the current operation, load the accumulator with the value 0/$00; the values in X and Y are then irrelevant. All register values are preserved during this routine.

65472 $FFCO JOPEN

Entry point for the Kernal OPEN routine, which opens a logical file to a specified device. The routine is entered via the IOPEN indirect vector (794/$031A), which normally points to the OPEN routine at 61403/$EFDB. You can modify the actions of the routine by changing the vector to point to a routine of your own. See the IOPEN entry for details.

At least one preparatory step is required before the standard OPEN routine is called: The SETLFS routine [$FFBA] must be called to establish the logical file number (184/$B8), the device number (186/SBA), and the secondary address (185/$B9). For tape (device 1), RS-232 (device 2), or serial (device 4 or higher), SETBNK [$FF68] and SETNAM [$FFBD] are also required to specify the length (183/$B7) and address (187-188/$BB-$BC) of the associated filename and the bank number (199/$C7) where the filename can be found.

The status register carry bit will be clear if the file is successfully opened, or set if it cannot be opened. When carry is set upon return, the accumulator will hold an error code indicating the problem. Possible error code values include 1 (10 files, the maximum allowed, are already open), 2 (a currently open file already uses the specified device number), and 5 (specified device did not respond). One exception is when RS-232 files are opened with x-line handshaking. Because of a bug in the RS-232 OPEN routine [$F040], carry will be set if the RS-232 device is present when x-line handshaking is used (if the DSR line is high), or it will be clear if the device is absent—the opposite of the proper setting. The RS-232 and tape/serial status flags (2580/$OA14 and 144/$90, respectively) also reflect the success of the operation.

65475 $FFC3 JCLOSE

Entry point for the Kernal CLOSE routine, which closes a specified logical file. The routine is entered via the ICLOSE in-

direct vector (796/$031C), which normally points to the CLOSE routine at 61832/$F188. You can modify the actions of the routine by changing the vector to point to a routine of your own. See the ICLOSE entry for details.

The standard CLOSE routine should be called with the accumulator holding the number of the logical file to be closed. If no file with the specified logical file number is currently open, the routine exits immediately. (No error will be indicated.) If a file with the specified number is open, its entry in the logical file number, device number, and secondary address tables will be removed. For RS-232 files, the driving CIA #2 interrupts will be disabled. For tape files, the final block of data will be written (followed by an end-of-tape marker, if one has been specified). For disk files, the EOI sequence will be performed. However, the 128 Kernal routine offers a special close function for disk files: If this routine is called with the status register carry bit set, and if the device number for the file is 8 or greater, and if the file has been opened with a secondary address of 15, the EOI sequence is skipped (the table entries for the file are deleted, but that's all). This provides a solution to a problem in earlier versions of the Kernal. A disk file opened with a secondary address of 15 is a command channel to the drive. Performing an EOI sequence to such a file closes all files currently open to the drive, not just the command channel file. This special mode allows the command channel file to be closed without disturbing other files that may be open to the drive.

65478 $FFC6 JCHKIN

Entry point for the Kernal CHKIN routine, which specifies a logical file as the source of BASIN and GETIN input. The routine is entered via the ICHKIN indirect vector (798/$031E), which normally points to the CHKIN routine at 61702/$F106. You can modify the actions of the routine by changing the vector to point to a routine of your own. See the ICHKIN entry for details.

For a logical file to receive input, it must first be opened (see the OPEN routine entry at 65472/$FFC0). The standard CHKIN routine should be called with the desired logical file number in the X register. The input channel (153/$99) is set to the device number for that file. If the device is RS-232 (device number 2), the routine also enables the CIA #2 interrupts

responsible for RS-232 reception. If a serial device (device number 4 or greater) has been specified, the device is also made a talker on the serial bus. If the file has successfully been set for input, the carry bit will be clear upon return. If carry is set, the operation is unsuccessful and the accumulator will contain the Kernal error code indicating which error has occurred. Possible error codes include 3 (file not open), 5 (device did not respond), and 6 (file not open for input). The RS-232 and serial status flags (2580/$0A14 and 144/$90, respectively) also reflect the success of the operation on one of those devices.

65481 SFFC9 JCKOUT

Entry point for the Kernal CKOUT routine, which specifies a logical file as the recipient of BSOUT output. The routine is entered via the ICKOUT indirect vector (800/$0320), which normally points to the CKOUT routine at 61772/$F14C. You can modify the actions of the routine by changing the vector to point to a routine of your own. See the ICKOUT entry for details.

For output to be sent to a logical file, it must first be opened (see the OPEN routine entry at 65472/$KFC0). The standard CKOUT routine should be called with the desired logical file number in the X register. The output channel (154/$9A) is set to the device number for that file. If the device is RS-232 (device number 2), the routine also enables the CIA #2 interrupts responsible for RS-232 transmission. If a serial device (device number 4 or greater) has been specified, the device is also made a listener on the serial bus. If the file has successfully been set for output, the carry bit will be clear upon return. If carry is set, the operation is unsuccessful and the accumulator will contain the Kernal error code indicating which error has occurred. Possible error codes include 3 (file not open), 5 (device did not respond), and 7 (file not open for output). The RS-232 and serial status flags (2580/$0A14 and 144/$90, respectively) also reflect the success of the operation on one of those devices.

65484 $FFCC JCLRCH

Entry point for the Kernal CLRCH routine, which resets the system for default I/O sources. The routine is entered via the ICLRCH indirect vector (802/$0322), which normally points to the CLRCH routine at 61990/$F226. You can modify the

actions of the routine by changing the vector to point to a routine of your own. See the ICLRCH entry for details.

The standard CLRCH routine resets the output channel (154/$9A) to device 3, the video display. (If the previous output channel is a serial device, it will be sent an UNLISTEN command.) The input channel (153/$99) is reset to device 0, the keyboard. (If the previous input channel is a serial device, it will be sent an UNTALK command.)

65487 $FFCF JBASIN

Entry point for the Kernal BASIN routine (called CHRIN in previous versions of the Kernal), which receives a byte from the logical file currently specified for input. The routine is entered via the IBASIN indirect vector (804/$0324), which normally points to the BASIN routine at 61190/$EF06. You can modify the actions of the routine by changing the vector to point to a routine of your own. See the IBASIN entry for details.

Except to retrieve input from the keyboard when the system is set for default I/O, you must open a logical file to the desired device and specify the file as the input source (see the entries for the OPEN and CHKIN routines, 65472/$FFC0 and 65478/$FFC6, respectively). For keyboard input (device 0), the standard BASIN routine accepts keypresses until RETURN is pressed; then it returns characters from the input string one at a time on each subsequent call. The character code for RETURN, 13/$0D, is returned when the end of an input string is reached. (Generally, the GETIN routine [FFE] is preferred for retrieving individual keypresses.) BASIN from tape (device 1) retrieves the next character from the cassette buffer. If all characters have been read from the buffer, the next data block is read from tape into the buffer. BASIN from RS-232 (device 2) returns the next available character from the RS-232 input buffer. If the buffer is empty, the routine waits until a character is received—unless the RS-232 status flag (2580/$0A14) indicates that the DSR signal from the external device is missing, in which case a RETURN character code, 13/$0D, is returned.

BASIN from the screen (device 3) is supposed to retrieve characters one at a time from the current screen line, ending with a RETURN character code when the last nonspace character on the logical line is reached. However, it does not work

properly in the current version of the 128 Kernal (see the entry for 49819/SC29B in Chapter 7 for details). BASIN from serial devices (device numbers 4 and higher) returns the next character available from the serial bus, unless the serial status flag (144/S90) contains a nonzero value. In that case, the RETURN character code is returned.

For all input devices, the received byte will be in the accumulator upon return. The contents of the X and Y registers will be preserved during input from keyboard, screen, or RS-232. For input from tape, only the X register contents are preserved. For input from serial devices, only the Y register contents are preserved. For input from screen, keyboard, or serial devices, the status register carry bit will always be clear upon return. For tape input, the carry bit will be clear unless the operation has been aborted by pressing the RUN/STOP key. For tape and serial input, the success of the operation will be indicated by the value in the tape/serial status flag (144/$90). Due to a bug in the RS-232 portion of BASIN, the carry bit will be set if a byte has been successfully received; it will be clear only if a RETURN character code is returned when the DSR signal is missing. The success of an RS-232 operation will be indicated by the value in the status flag (2580/$0A14).

65490 SFFD2 JBSOUT

Entry point for the Kernal BSOUT routine (called CHROUT in previous versions of the Kernal), which sends a byte to the logical file currently specified for output, The routine is entered via the IBSOUT indirect vector (806/$0326), which normally points to the BSOUT routine at 61305/$EF79. You can modify the actions of the routine by changing the vector to point to a routine of your own. See the IBSOUT entry for details.

Except to send output to the screen when the system is set for default I/O, you must open a logical file to the desired device and specify the file as the output target (see the entries for the OPEN and CKOUT routines, 65472/$FFC0 and 65481/$FFC9, respectively). For output to tape (device 1), the character is stored at the next available position in the cassette buffer. When the buffer is full, the data block is written to tape. For output to RS-232 (device 2), the character is stored in the next available position in the RS-232 output buffer. If the buffer is

full, the routine waits until a character is sent. BSOUT to the screen (device 3) [$C00C] prints the character on the active display at the current cursor position. The CIOUT routine r$FFA8] is used for output to serial devices (device numbers 4 and higher).

Regardless of the output device, the contents of the accumulator and X and Y registers will be preserved during this routine. The status register carry bit will always be clear upon return unless tape output has been aborted by pressing the RUN/STOP key (in that case, the accumulator will also be set to 0/$00 to set the Z bit as well). For tape, serial, or RS-232 output, the success of the operation will be indicated by the value in the status flag (144/$90 for tape or serial operations; 2580/$0A14 for RS-232).

65493 $FFD5 JLOAD

Entry point for the Kernal LOAD routine 62053/$F265, which loads or verifies a program file from tape or disk into a specified area of memory. Before calling this routine, the SETLFS routine [$FFBA] must be called to establish the device number (186/$BA) and the secondary address (185/SB9) for the operation. The logical file number (184/$B8) isn't significant for loading. The secondary address value determines whether the load/verify will be absolute or relocating. If bit 0 of the secondary address is %0 (if the value is 0, for example), the file will be loaded starting at the address specified in the X and Y registers when the LOAD routine is called (a relocating load). If the bit is %1 (if the value is 1, for example), the data will be loaded starting at the address specified in the file itself (an absolute load). For tape files, the secondary address specification can be overridden by the file type. Nonrelocatable tape program files always reload to their absolute address, even if the secondary address value specifies a relocating load. The SETNAM routine [$FFBD] must be called to specify the length (183/$B7) and address (187-188/$BB-$BC) of the associated filename. The SETBNK routine [$FF68] must be called to specify the bank number where the filename can be found (199/$C7) and the bank into which data is to be loaded (198/$C6).

When the routine is called, the accumulator should hold the operation type value (0/$00 for a load or any nonzero value for a verify). If the secondary address specifies a relocating load, the X and Y registers should hold the starring ad-

^

dress at which data is to be loaded (low byte in X, high byte in Y). The load will be aborted if it extends beyond address 65279/$FEFF. This is to prevent corruption of the MMU configuration register, which appears at 65280/$FF00 in every bank.

The status register carry bit will be clear upon return if the file has successfully been loaded, or it will be set if an error has occurred or if the RUN/STOP key has been pressed to abort the load. When carry is set upon return, the accumulator will hold the Kernal error code indicating the problem. Possible error code values include 4 (file was not found), 5 (device was not present), 8 (no name was specified for a serial load), 9 (an illegal device number was specified), and 16 (the load extended beyond address $FEFF). The success of the operation will also be indicated by the value in the tape/serial status flag **(144/$90).**

65496 $FFD8 JSAVE

Entry point for the Kernal SAVE routine at 62782/$F53E, which saves the contents of a block of memory to disk or tape. Before this routine is called, the SETLFS routine [$FFBA] must be called to establish the device number (186/$BA) and secondary address (185/$B9) for the operation. The logical file number (184/$B8) isn't significant for saving. The secondary address is irrelevant for saves to serial devices, but for tape it specifies the header type. If bit 0 of the secondary address value is %1 (if the value is 1, for example), the data will be stored in a nonrelocatable file—one that will always load to the same memory address from which it has been saved. Otherwise, the data will be stored in a relocatable file. If bit 1 of the secondary address is %1 (if the value is 2 or 3, for example), the file will be followed by an end-of-tape marker. The SETNAM routine [$FFBD] must be called to specify the length (183/$B7) and address (187-188/$BB-$BC) of the associated filename. The SETBNK routine [SFF68] must be called to specify the bank number where the filename can be found (199/$C7) and the bank from which data is to be saved (198/SC6).

A two-byte zero-page pointer should be loaded with the starting address of the block of memory to be saved. The routine should be called with the accumulator holding the address of the zero-page pointer and the X and Y registers holding the

ending address plus 1 for the save (low byte in X, high byte in Y). To save all bytes in the desired area, it's important to remember that X and Y must hold an address that is one location beyond the desired ending address.

The status register carry bit will be clear if the file has successfully been saved, or it will be set if an error has occurred or if the RUN/STOP key has been pressed to abort the save. When carry is set upon return, the accumulator will hold the Kernal error code indicating the problem. Possible error code values include 5 (serial device was not present), 8 (no name was specified for a serial save), and 9 (an illegal device number was specified). The success of the operation will also be indicated by the value in the tape/serial status flag (144/S90) upon return.

65499 6FFDB JSETTIM

Entry point for the Kernal SETTIM routine at 63077/$F665, which sets the value in the software jiffy clock (160-162/ $A0-$A2). The clock is normally incremented 60 times per second by the UDTIM routine [$FFEA] as part of the IRQ sequence. The value in the accumulator is transferred to the low byte (162/$A2), the value in the X register to the middle byte (161/$A1), and the value in the Y register to the high byte (160/$A0). The specified value should be less than $4F1AO1, the count corresponding to 24:00:00. Whenever the clock reaches that value (which corresponds to the number of jiffies in 24 hours), the UDTIM routine resets all three bytes to zero.

65502 $FFDE JRDTIM

Entry point for the Kernal RDTIM routine at 63070/$F65E, which returns the current value in the jiffy clock (160-162/ »A0-$A2). The clock is normally incremented 60 times per second by the UDTIM routine [$FFEA] as part of the IRQ sequence. The clock value corresponds to the number of jiffies U/60 second intervals) that have elapsed since the system J!astunied on or reset, or the number of jiffies since midnight UW:00:00) if the clock value has been set. The low byte of the oock value (162/SA2) is returned in the accumulator, the mid-, . e 5 te (161/SA1) in the X register, and the high byte (160/ $A0) in the Y register.

65505 SFFE1 JSTOP

Entry point for the Kernal STOP routine, which checks whether the RUN/STOP key is pressed. The routine is entered via the ISTOP indirect vector (808/$0328), which normally points to the STOP routine at 63086/$F66E. You can modify the actions of the routine by changing the vector to point to a routine of your own. See the ISTOP entry **for** details.

The standard STOP routine returns with the status register Z bit clear if the key is not pressed or with the bit set if it is. Additionally, if RUN/STOP is pressed, the CLRCH routine [$FFCC] is called to reset default I/O, and the count of keys in the keyboard buffer (208/$D0) is reset to zero.

65508 8FFE4 JGETIN

Entry point for the Kernal GETIN routine, which retrieves a character from the current input device. The routine is entered via the IGETIN indirect vector (810/S032A), which normally points to the GETIN routine at 61163/$EEEB. You can modify the actions of the routine by changing the vector to point to a routine of your own. See the IGETIN entry for details.

The standard GETIN routine checks to see whether the current input device is 0 (keyboard) or 2 (RS-232). If it's not one of these, the BASIN routine [$EF06] is used instead. (See the BASIN entry for information on GETIN's behavior for other devices.) For keyboard or RS-232, the retrieved character will be in the accumulator upon return, and the status register carry bit will be clear. The value 0/$00 will be returned if no character is available from the device. The contents of the Y register are unaffected by this routine. For RS-232, bit 3 of the status flag (2580/$0A14) will be set if no characters are available.

65511 $FFE7 JCLALL

Entry point for the Kernal CLALL routine, The routine is entered via the ICLALL indirect vector (812/$032C), which normally points to the CLALL routine at 61986/SF222. You can modify the actions of the routine by changing the vector to point to a routine of your own. See the ICLALL entry for details.

The standard CLALL routine resets the number of open files (152/$98) to zero, then falls through into the CLRCH

routine to reset default I/O. Note that, despite its name, the routine doesn't actually close any files that may be open to tape, disk, or RS-232 devices. Unclosed files may cause problems' particularly on disks, so this routine is of limited usefulness' see the Kernal CLOSE_ALL entry [$FF4A] for a routine that properly closes all files open to a serial device.

65514 $FFEA JUDTIM

Entry point for the Kernal UDTIM routine at $62968/$F5F8, which increments the software jiffy clock, decrements the countdown timer, and scans the keyboard column containing the RUN/STOP key.

65517 $FFED JSCRORG

Calls the Kernal SCRORG routine's screen editor jump table entry at 49167/$C00F. (This routine was called SCREEN in previous versions of the Kernal.) The routine returns information on the size of the screen's current output window. Upon return, the X register will contain the number of columns minus 1 in the current window, and the Y register will contain the number of rows minus 1. The accumulator will hold the maximum column number for the display currently active (39 for the 40-column screen or 79 for the 80-column screen).

65520 $FFF0 JPLOT

Calls the Kernal PLOT routine's screen editor jump table entry at 49176/$C018. The routine reads or sets the cursor position on the active display. If it is called with the status register carry bit clear, the value in the X register will specify the new cursor row (vertical position), and the value in the Y register will specify the column (horizontal position). The position will be relative to the home position of the current output window rather than to the upper left corner of the screen. (Of course, in the case of a full-screen output window, the default condition, the upper left comer of the screen is the home position of the window.) The carry bit will be set upon return if the specified column or row value is beyond the right or bottom margin of the current output window. If the routine is called with the carry bit set, the row number for the current cursor position is returned in the X register, and the current column

number is returned in the Y register. Again, the position is relative to the home position of the output window rather than the absolute home position of the screen,

65523 8FFF3 JIOBASE

Entry point for the Kernal IOBASE routine at 63361/$F781, which returns the base address of the system I/O block (53248/$D000). The low byte, 0/$00, is returned in the X register, and the high byte, 208/$D0, is returned in Y.

65526-65527 $FFF6-$FFF7 Unused

Two unused bytes filled with the value 255/$FF.

65528 $FFF8 System Vector

These locations in bank 1 are the soft reset vector, an address through which the reset routine [$E000] takes an indirect jump. The ROM vector (in bank 15) is not normally used, but it still contains the default address (57892/$E224).

8502 CPU Vectors

One of the hardware features of the 8502 microprocessor (and its predecessors, the 6502 and 6510) is that certain events cause the processor to jump to an address held in one of the following two-byte vectors. This implies that any system using one of these processors must have ROM at these addresses, or, as in the case of the RAM banks of the 128, must have some method of initializing the RAM at these addresses with the appropriate values.

An NMI (nonmaskable interrupt), triggered by a high-to-low transition on the processor's NMI input line, causes a jump to the address at 65530/$FFFA. A reset, triggered when the RESET input line is pulled low, causes a jump to the address in 65532/$FFFC An IRQ (interrupt request), triggered when the IRQ input line is pulled low, causes a jump to the address in 65534/$FFFE, as does the execution of a BRK opcode (0/$00),

65530 $FFFA NMI

When an NMI signal is detected, the processor completes execution of the current instruction, then stores the contents of the status register and the address of the next instruction on

r

the stack. The address in this vector is then loaded into the program counter, and execution continues from that address. In the 128, this vector contains 65285/SFF05, the address of a routine that saves the accumulator, X and Y register, and MMU configuration register values on the stack; it then configures the system for bank 15 and jumps through the INMI vector at 792/$0318.

65532 $FFFC RESET

When a reset signal is detected, the processor immediately terminates the current operation and loads the program counter with the address in this vector, Execution then resumes with the routine at that address. In the 128, this vector contains 65341/$FF3D, the address of a routine which configures the system for bank 15, then jumps to the reset routine at 57344/$E000.

65534 SFFFE IRQ/BRK

When a BRK opcode is executed or an IRQ signal is detected, the processor completes execution of the current instruction, then stores the contents of the status register and the address of the next instruction on the stack. The address in this vector is then loaded into the program counter, and execution continues from that address. In the 128, this vector contains 65303/ $FF17, the address of a routine that saves the accumulator, X and Y register, and MMU configuration register values on the stack; it then configures the system for bank 15, determines whether the interrupt was the result of a BRK or IRQ, and jumps accordingly through either the 1BRK vector at 790/S0316 or the IIRQ vector at 788/$0314.

Interrupts

Todd Heimarck

The 8502 microprocessor that is the brains of the 128 is highly methodical and single-minded. When executing a program, it fetches an operation code from memory, fetches additional operand bytes (depending on the opcode it gets), and executes the instruction. Then it goes back for the next one, the next one, and so on. If you left the 8502 to itself, it would execute lots of instructions, but it would never communicate with the outside world. For example, the 8502 has no way of remembering by itself to check the keyboard. Without the keyboard, you'd have an awfully tough time writing a program.

The situation is analogous to the absent-minded professor who would forget to eat dinner or go to sleep if it weren't for the housekeeper's help. The 8502 needs an assistant to give it a nudge occasionally to remind it to attend to other tasks. In computer parlance, this nudge is called an interrupt. When an interrupt request occurs, the 8502 saves its position in the current program and goes off to run another program elsewhere in memory. When it's finished with the interrupt routine, the 8502 returns to the main program that was running before the interrupt occurred.

Interrupts for the 8502 fall into two broad classes: maskable interrupts (called IRQs) and nonmaskable interrupts (NMIs). As the names imply, the processor can be told to ignore IRQ interrupts (using the SEI instruction), but NMIs demand immediate attention. The 128 uses both types to manage a variety of housekeeping chores.

The Housekeeping Chores

The main housekeeping interrupt is an IRQ generated 60 times per second. It's called the system hardware interrupt, or sometimes the jiffy interrupt—a 1/60-second unit of time is called a j^fy. (Actually, if you have a European 128 using the PAL video system, your jiffies last 1/50 second, and your system IRQ interrupts occur 50 times per second.) The interrupt

causes several things to happen. First, the 8502 automatically pushes the current address in the program counter onto the stack, as if it were executing a JSR, Next other sources of interrupts are turned off, so an interrupt doesn't happen in the middle of servicing the IRQ. The accumulator and X and Y registers are pushed on the stack, and the current state of the MMU configuration register at 65280/$FF00 is also put on the stack (because the computer has to return to the proper bank setup when it's finished handling the interrupt).

The IRQ handling routine has several distinct segments. The first sets up the current screen display mode. For example, if you've created a split graphics screen with either GRAPHIC 2 or GRAPHIC 4, the VIC chip has to be set up for the proper screen and color memory areas. Next, the keyboard is scanned. If a key has been pressed, the appropriate character code is determined and stored in the keyboard buffer at 842/$034A. If it's time for the 40-column display's cursor to blink, that's taken care of. The software jiffy clock at 160-162/$A0-$A2 is clicked up a notch. The last big segment of the IRQ sequence is the BASIC IRQ routine, which handles a variety of sprite and sound support functions. For example, if you've used the MOVSPR to put a sprite into motion, the new locations have to be calculated and the sprite position updated. COLLISION is also an interrupt-driven command, handled during the IRQ routine. And the sound durations for the SOUND and PLAY statements are manipulated during this routine.

Then, the interrupt ends. The registers are restored to their former values and the bank configuration is stored back to 65280/$FF00. Interrupts are reenabled and the RTI instruction brings the 8502 back to the program it was working on.

In 128 mode, the timing of the system IRQ interrupt is tied directly to the position of the raster beam that traces the video screen. When the VIC chip gets to the point where it's drawing a certain line on the screen, the interrupt occurs. This is a convenient way to tie in the split-screen modes. In 64 mode, the interrupt request is generated by a hardware timer in one of the CIA chips; this timer is not necessarily in sync with the current position of the raster beam.

Nonmaskable Interrupts (NMIs)

The other type of interrupt, the NMI, is normally caused by one of two things: pressing the RESTORE key, or performing an RS-232 operation. When an NMI occurs, the 128 jumps to the address held at 65530/SFFFA, which points to the Kernal area at 65285/$FF05. This is a very short program, which ends up bouncing off the RAM vector at 792-793/ $0318-$0319.

If the interrupt was triggered by the RESTORE key, the system immediately checks to see if the RUN/STOP key was held down at the time RESTORE was pressed. If not, the 8502 merely returns to the program it was running. If both RUN/ STOP and RESTORE were pressed, the RUN/STOP-RESTORE initialization sequence is performed, ending with a warm start of BASIC (see the entry for 64064/$FA40 in Chapter 9).

If something's happening at the RS-232 port, the incoming character is processed and stored in the buffer at 3072/ $0C00, or the outgoing character is sent from the buffer at 3328/$0D00.

Other Sources of IRQs

The hardware interrupt is not the only way to have an IRQ happen. You can force the equivalent of an IRQ by including the BRK instruction (the opcode is $00) in a machine language program. Any of the the interrupt sources in the VIC chip or CIA #1 can generate an IRQ request. These include sprite collisions, raster interrupts, the light pen input, or any of the CIA timers. Of these, only the raster interrupt is used by the system. As mentioned, a raster interrupt occurs when the screen redraw routine reaches a certain line on the screen. The line to trigger the standard jiffy interrupt is off the bottom of the visible area of the screen. However, the system also supports mid-screen raster interrupts to manage the split-screen displays of the GRAPHIC 2 and GRAPHIC 4 modes. The IRQ routine checks for mid-screen raster interrupts, and performs only the screen-setup portion of the interrupt routine in that case. For any other type of IRQ, you must write your own handling routine.

The first thing the 128 does at the start of an IRQ is jump off the processor vector at 65534/$FFFE, which takes it to the short routine at 65303/$FF17. At this point, it decides whether the interrupt was caused by software (a machine language BRK instruction) or hardware (the raster, the timer, the light pen, a sprite collision, and so on).

A BRK instruction causes the system to proceed to a RAM vector at 790-791/$0316-$0317. The usual response to a BRK is to start up the built-in monitor, which is also called by the BASIC MONITOR command. The advantage to this is plain: When you're writing a machine language program, you can insert $00s here and there in the program (breakpoints) and monitor the progress of the program.

Any type of hardware interrupt sends the computer to the vector at 788-789/$0314-$0315, where the interrupt is handled. It's at this point that the keyboard is polled, the jiffy clock is updated, and the various other housekeeping chores are done.

Writing Your Own Interrupt Handler

Commodore has inserted a deliberately vulnerable point in the process of handling interrupts: the RAM vectors at 788-793/$0314-$0319. You can change these pointers to turn off interrupts or to set up your own interrupt processing routine.

The following machine language program provides a short example of how to wedge into the IRQ routine. Before jumping to the normal IRQ handler, it checks the status of the 40/80 DISPLAY key and switches screens if the key setting doesn't match the currently active screen.

First, the address currently in the IRQ vector (788-789/$0314-$0315) is stored as the target address of the JMP that ends our own routine. Then the address of our MAIN routine is stored into the IRQ vector. And that's the end of the installer routine.

From then on, whenever an IRQ interrupt occurs, the 128 jumps to our custom routine, because the vector has been changed. Within our routine, the 40/80 switch is checked and if it has been changed, an ESCape (CHR$(27)) and X are placed in the keyboard buffer. Whether or not the 40/80 DISPLAY key has been pressed, we finish the routine by executing a JMP to the normal IRQ routine.

```
10  ; This program sets up an interrupt that checks the 40/80 key
20  ; and switches to the appropriate screen if a change has been made
30  !oRG $0C00          ; Put the program at $0C0O
40  FORTY = $D7         ; 0 if set to 40 columns, 128 if 80
50  KEYNDX = $D0        ; Index to how many keys are waiting in the
                          buffer
60  IRQVEC = $0314      ; Vector to the IRQ routine
70  KEYBUF = $034A      ; The keyboard buffer
80  SWITCH = $D505      ; Bit 7 tells us if the 40/80 switch is up or down
90  ; %0 is down, %1 is up
170 ;
180 ; The first step is to save the current address in the IRQ vector
190 ;
200 SEI                 ; Disable interrupts while the vector is being
                          changed
210 LDA IRQVEC          ; Get the low byte of the vector
220 CMP JUMP+1          ; See if we've done this before
230 BEQ LEAVE           ; If ifs equal, jump ahead to exit the routine
240 STA JUMP +1         ; Else store the current IRQ vector address as the
250 LDA 1RQVEC+1        ; target of the JMP to exit from the MAIN routine
260 STA JUMP+ 2
265 ;
270 ; Now reset the vector to point to our routine at MAIN
275;
280 LDA #<MAIN          ; Put the address of the MAIN routine
290 STA IRQVEC          ; into IRQVEC
300 LDA #>MAIN
310 STAIRQVEC+1
320;
330 CLI                 ; We're done resetting the vector, so reenable
340 LEAVE RTS           ; interrupts and exit from the setup routine
350;
400 MAIN = •
420 ; First, check the status of the 40/80 switch
430 ;
440 LDA FORTY           ; Value is either 0 (40 columns) or $80 (80 columns)
450 EOR SWITCH          ; Exclusive-OR with the 40/80 switch flag (bit 7)
460                     ; if (he status register N flag is %0,
470 BPL CHANGE          ; go ahead and switch displays
480 JUMP JMP $FFFF      ; Else jump to the normal IRQ routine
490                     ; Note: We never really jump to address $FFFF.
495                     ; The installation routine changes the $FFFF to the
500                     ; address of the normal IRQ routine
510                     ;
520 CHANGE LDA #27; Put the code for the ESCape key
530 STA KEYBUF          ; into the keyboard buffer, along with
540 LDA #"X"            ; the letter X (ESC X means switch

580 LDA KEYBUF-1        ; Watch the keyboard buffer and branch if no IRQ routine
```

Bugs and Quirks in 128 ROM

Like all new computer systems, the 128 has a few bugs in its system ROMs. Neither BASIC nor the Kernal is seriously flawed, but both include many new routines that weren't part of earlier versions of the operating system. With all the new code, it was inevitable that some mistakes would be made. Some of the items described below are more idiosyncrasies than true bugs. Others aren't really errors in programming, but rather errors in documentation—instances where the routine works, but not exactly as described in the *128 System Guide* or *128 Programmer's Reference Guide.* It's entirely possible that some of these situations will be corrected in future versions of either the ROMs or the manuals.

BASIC

1. Perhaps the most significant bug in BASIC 7.0 is that, contrary to what the manuals claim, you can't use negative relative parameters in graphics statements. For example, according to the *System Guide,* DRAWTO -5,-5 should be a valid statement to draw a line five pixels up and five pixels left of the current pixel cursor position. Instead, it causes an ILLEGAL QUANTITY error. The graphics routines themselves are set up to handle such coordinates, but the routine which evaluates parameters uses a subroutine that checks the sign of the value and causes an error if the value is negative.

2. An attempt to OPEN a logical file to an RS-232 device such as a modem will result in a DEVICE NOT PRESENT error if x-line handshaking is specified. This is the result of the Kernal RS-232 OPEN bug described below.

3. An attempt to use INPUT* with a logical file specified for device 3 (the screen) will not work because of the Kernal screen BASIN bug mentioned below.

4. In strings for the PLAY statement, any number of digits can follow a U (volume) command, but only the last one counts. Also, a digit alone, with no other command, is the

same as U followed by that digit. All four of the following statements are equivalent:

```
PLAY "VI U7 ABCDE"
PLAY "VI U1234567 ABCDE"
PLAY "VI U7777 ABCDE"
PLAY "VI 7 ABCDE"
```

5. The RSPRITE function accepts sprite number parameters up to 16, even though only 1-8 are valid.

6. The RWINDOW function doesn't return values stated in the *System Guide.* The manual states that RWINDOW(O) returns the number of lines in the current output window and RWINDOW(l) returns the number of rows. Actually, RWINDOW(O) returns the number of rows minus 1 in the output window, and RWINDOW(l) returns the number of columns minus 1.

7. Commodore literature claims that the default scaled size for a standard bitmapped screen (if you use SCALE 1 without additional parameters) is 1023 X 1023. Actually, it's 1024 X 1024, but that's relatively trivial. A more serious problem with scaling is that it doesn't work as claimed for multicolor (GRAPHIC 3 or GRAPHIC 4) screens. The default scaled multicolor screen is also 1024 X 1024, not 511 X 511 as claimed in the *System Guide.* When you supply scaling factors for a multicolor screen, you must use twice the desired horizontal scaling value. For example, to scale the multicolor display to 256 X 256, you must use SCALE 1,512,256. The horizontal factor must be greater than 320 to prevent an ILLEGAL QUANTITY error.

8. The SCNCLR routine doesn't properly fill color memory for a GRAPHIC 4 (split multicolor bitmapped and text) screen. That is, the routine fails to fill color memory with the current color source 3 value. Since that routine is also used for the clear option of the GRAPHIC statement, GRAPHIC 4,1 won't properly initialize color memory either. For multicolor bitmapped mode, color memory determines the color of pixels with %11 bit patterns. The solution is to follow any GRAPHIC 4 statement with a SCNCLR 3, which does properly fill the color memory area. As part of the same bug, color memory is unnecessarily filled for SCNCLR 2 or GRAPHIC 2,1, but that has no visible effect because color memory is not used in standard bitmapped mode.

9. All BASIC disk commands automatically add the drive 0 specification if no other drive number is specified, except for CATALOG and DIRECTORY. Although Commodore drives will supply the directory of drive 0 by default, failing to specify a drive number sets up conditions for the infamous 1541 drive Save-with-Replace bug. Thus, it's safest to add a DO after each CATALOG or DIRECTORY.

10. In the BASIC disk commands, the ON U parameter can be specified any number of times; only the last occurrence counts. CATALOG ON U6 ON U7 ON U8 is equivalent to CATALOG ON U8.

11. Commodore literature fails to describe the RREG statement included in BASIC 7.0. The proper format is RREG *variable 1, variable 2, variable* 3, *variable 4.* The specified variables will be set to the values in locations 6, 7, 8, and 5, respectively. These locations hold the accumulator, X register, Y register, and status register values from the last time the JSRFAR routine was used, such as upon return from the most recent SYS statement. Any of the variables may be omitted, so RREG AC,,,SP and RREG „YR are valid statements.

12. BASIC 7.0 allows LIST to be used as a statement within a program, but the RENUMBER statement will not renumber any line numbers that may follow LIST in a program being renumbered. This is a trivial oversight, since occasions to use LIST within a program are quite rare,

Kernal

1. The BASIN routine will not accept input from the screen because it fails to properly mark the end of the input string. Refer to the discussion of the routine at 49819/$C29B in Chapter 7 for more information and a solution to the problem.

2. Although it isn't mentioned in the manuals, ESC ESC is accepted as a synonym for ESC O (cancel quote mode).

3. The screen editor CINT and SWAPPER routines copy one too many bytes when initializing or exchanging the contents of the screen editor variable table at 224-249/$E0-$F9. As a result, the contents of the otherwise unused locations 250/$FA and 256O/$0A5A are overwritten whenever the screen is initialized or switched.

4. The keyboard decoding table for CAPS LOCK has the incorrect value for the Q key, so that key will appear to be unaffected by CAPS LOCK,

5. If x-line handshaking was specified in the OPEN routine for RS-232, the routine returns with the status register carry bit set if the external device responded, or clear otherwise. This is the opposite of the carry bit setting when any other device is successfully opened.

Character, Screen, and Keyboard Codes

The Commodore 128 represents characters in several different manners: as characters, as screen codes, and as keyboard codes. This appendix covers each of these possible representations.

Character Codes

The 128 has two separate sets of 256 characters. The set that is normally activated when the computer is turned on is called the uppercase/graphics set. It has only uppercase (capital) letters, but includes many graphics characters. The alternative lowercase/ uppercase set has both lowercase and uppercase letters, but includes substantially fewer graphics characters. It is useful for creating more attractive text displays, and is essential for tasks like word processing. You can switch manually between sets by pressing the SHIFT and Commodore keys simultaneously. You can also switch to the lowercase/uppercase set within a program by printing character 14—PRINT CHR$(14). To switch back to the normal uppercase/graphics set, print character 142,

In the 40-column display mode, switching character sets affects all characters currently on the screen. For example, uppercase letters printed from the uppercase /graphics set will change to lowercase characters after CHR$(14) is printed. However, any character set switching in 80-column mode affects only those characters printed after the switch. In this case, uppercase letters printed from the uppercase/graphics set will remain in uppercase after a CHR$(14) is printed.

The lowercase/uppercase character set has two identical groups of uppercase letters, characters 97-122 and characters 193-218. When this set is being used, PRINT CHR$(97) and PRINT CHR$(193) both cause an A to be displayed on the screen. However, you should be aware that the ASC function always returns values from the higher group—in lowercase/ uppercase, PRINT ASC("A") always gives 193 instead of 97.

In machine language, the Kernal GETIN routine also returns values 193-218 for shifted letters in the lowercase/uppercase set.

The following table lists the codes for the 128 character

25, 26, 128, 131, and 132.

Dec	Hex	Uppercase/Graphics Set	Lowercase/Uppercase Set
2	02	underline on[3]	
5	05	white	
7	07	bell tone[2]	
8	08	disable SHIFT-Commodore[3]	
9	09	tab'	
		enable SHIFT-Commodore[3]	
10	0A	linefeed[2]	
11	0B	disable SHIFT-Commodore*	
12	OC	enable SHIFT-Commodore[2]	
13	0D	RETURN	
14	0E	switch to lowercase	
15	OF	flash on[1]	
17	11	cursor down	
18	12	reverse on	
19	13	home	
20	14	delete	
24	18	tab set/clear[2]	
27	IB	ESCape	
28	1C	red	
29	ID	cursor right	
30	IE	green	
31	IF	blue	
32	20	space	
33	21	!	!
34	22		
35	23	#	#
36	24	$	$
37	25	%	%

pec	Hex	Uppercase/Graphics Set	Lowercase/Uppercase Set
38	26	&	&
39	27		
40	28	(	(
41	29	)	)
42	2A	*	*
43	2B	+	+
44	2C		,
45	2D	-	-
46	2E		.
47	2F	/	/
48	30	0	0
49	31	1	1
50	32	2	2
51	33	3	3
52	34	4	4
53	35	5	5
54	36	6	6
55	37	7	7
56	38	8	8
57	39	9	9
58	3A		*
59	3B		
60	3C		<
61	3D		=
62	3E		>
63	3F	?	?
64	40	@	@
65	41	A	a
66	42	B	b
67	43	C	c
68	44	D	d
69	45	E	e
70	46	F	f

Dec	Hex	Uppercase/Graphics Set	Lowercase/Uppercase Set
71	47	G	g
72	48	H	h
73	49	I	i
74	4A	J	j
75	4B	K	k
76	4C	L	l
77	4D	M	m
78	4£	N	n
79	4F	O	o
80	50	P	p
81	51	Q	q
82	52	R	r
83	53	S	t
84	54	T	u
85	55	U	v
86	56	V	w
87	57	W	x
88	58	X	y
89	59	Y	z
90	5A	Z	[
91	5B	[	E
92	5C	£	1
93	5D	]	r
94	5E	I	
95	5F	4-	
96	60	B	&
97	61	*	A
98	62	DD	B
99	63		C
100	64	a	D
101	65	H	E
102	66	S	F
103	67	DD	G

Dec	Hex	Uppercase/Graphics Set	Lowercase/Uppercase Set
104	68	LL	H
105	69	E	I
106	6A	ffl	J
107	6B	E	K
108	6C	LJ	L
109	6D	\	M
110	6E	z	N
111	6F	r	O
112	70	n	P
113	71		Q
114	72	y	R
115	73		S
116	74	I]	T
117	75	r^	u
118	76		V
119	77	ip	w
120	78		x
121	79	r	Y
122	7A		z
	7B	S	+
124	7C	B	
125	7D	[E	•D
	7E		SB
127	7F		S3
129	81	orange[4]	
		dark purple[1]	
130	82	underline off	
133	85	F1	
134	86	F3	
	87	F5	
	88	F7	
137	89	F2	
	8A	F4	

Dec	Hex	Uppercase/Graphics Set	Lowercase/Uppercase Set
139	8B	F6	
140	8C	F8	
141	8D	SHIFT-RETURN	
142	8E	switch to uppercase	
143	8F	flash off[1]	
144	90	black	
145	91	cursor up	
146	92	reverse off	
147	93	clear screen	
148	94	insert	
149	95	brown[4]	
		dark yellow[1]	
150	96	light red	
151	97	dark gTay[4]	
		dark cyan[1]	
152	98	medium gray	
153	99	light green	
154	9A	light blue	
155	9B	light gray	
156	9C	purple	
157	9D	cursor left	
158	9E	yellow	
159	9F	cyan	
160	A0	SHIFT-space	
161	A1		
162	A2		
163	A3		
164	A4		
165	A5		
166	A6		
167	A7		
168	A8		
169	A9		

Dec	Hex	Uppercase/Graphics Set	Lowercase/Uppercase Set
170	AA	3	[ ]
171	AB	[B	IB
172	AC	Ld	Q
173		AD	ffl ffl
174	AE	ffl	T
175	AF	U	•
176		B0	ffl ffl
177		Bl	ffl ffl
178	B2		&H ffl
179	B3	*m*	*m*
180	B4	CJ	
181	B5	C	ID
182	B6	LI	Ll
183	B7		
184	B8	H	n
185	B9	U	y
186	BA	•	~*V*\
187	BB	H	B
188	BC	L3	1
189		BD	ffl ffl
190	BE	H	H
191	BF	fij	BJ
192	CO	H	H
193	Cl	H	A
194	C2	J |	B
195	C3	H	C
196	C4	S	D
197	C5	H	E
198	C6	H	F
199	C7	l i	G
200	C8	U	H
201	C9	B	I
202	CA	f?	J

Dec	Hex	Uppercase/Graphics Set	Lowercase/Uppercase Set
203	CB	ffl	K
204	CC	t]	L
205	CD	k |	M
206	CE	g]	N
207	CF	|-j	0
208	DO	3	P
209	D I	\m\	Q
210	D2	H	R
211	D3	m	S
212	D4	11	T
213	D5	H	U
214	D6	X	V
215	D7	[g	w
216	D8	t	X
217	D9	[J	Y
218	DA	3	Z
219	DB	+	+
220	DC	BC	B
221	DD	f l	I f
222	DE	i i l	g |
223	DF	^	j g
224	E0		
225	El	| J	| j
226	E2	y	y
227	E3	^	•
228	E4	•	
229	E5.	O	Hi
231	E7	1	[ ]
232	E8	i i	^
233	E9	B	5g
234	EA	El	•
235	EB	DB	Lfe

Dec	Hex	Uppercase/Graphics Set	Lowercase/Uppercase Set
236	EC	H	~~*m*
237	ED	t	^
238	EE	ffl	SI
239	EF	U	U
240	F0	_T	L^
241	FI	a	*m*
242	F2		H ffl
243	F3	tfj	J !
244	F4	G	I !
245	F5	[I	I I
246	F6	|	| |
247	F7	H	
248	F8	H	H
249	F9	y	M
250	FA	Ul	^
251	FB	kJ	B
252	FC	"	H
253	FD	B	?J
254	FE	Fl	
255	FF	ifl	S?

Notes

1. For *80-column display only*
2. *For 128 mode only*
3. *For 64 mode only*
4. *For 40-colunm display only*

Screen Codes

There are 256 screen codes for each character set; codes 128-255 are the reverse images of codes 0-127. To display any character in reverse video, simply add 128 to its screen code value. Thus, POKE 1024,1:POKE 1025,1 + 128 displays an A and a reverse A.

The character ROM has a total of 153 different characters. In the uppercase/graphics set, the character patterns for codes 32 and 96 are identical, as are those for 64 and 67, 66 and 93, 101 and 116, and 103 and 106.

Dec	Hex	Uppercase/Graphics Set	Lowercase/Uppercase Set
0	00	@	@
1	01	A	a
2	02	B	b
3	03	C	c
4	04	D	d
5	05	E	e
6	06	F	*i*
7	07	G	g
8	08	H	h
9	09	I	i
10	0A	J	j
11	0B	K	k
12	0C	L	1
13	0D	M	m
14	0E	N	n
15	OF	O	o
16	10	P	p
17	11	Q	q
18	12	R	r
19	13	S	s
20	14	T	t
21	15	U	u
22	16	V	v
23	17	W	w

Dec	Hex	Uppercase/Graphics Set	Lowercase/Uppercase Set
24	18	X	x
25	19	Y	y
26	1A	Z	z
27	IB	[	[
28	1C	E	E
29	ID	]	]
30	IE	t	T
31	IF	*-	«
32	20	space	
33	21	!	!
34	22		ft
35	23	#	#
36	24	$	$
37	25	%	%
38	26	&	&
39	27		t
40	28	(	(
41	29	)	)
42	2A		*
43	2B	+	+
44	2C	t	t
45	2D	-	-
46	2E		*
47	2F	/	/
48	30	0	0
49	31	1	1
50	32	2	2
51	33	3	3
52	34	4	4
53	35	5	5
54	36	6	6
55	37	7	7
56	38	8	8

Dec	Hex	Uppercase/Graphics Set	Lowercase/Uppercase Set
57	39	9	9
58	3A	:	
59	3B	;	;
60	3C	<	<
61	3D	=	=
62	3E	>	>
63	3F	?	?
64	40	g	B
65	41	•	A
66	42	I]]	B
67	43	H	C
68	44	g	D
69	45	—'	E
70	46	—	F
71	47	|]	G
72	48	[11	H
73	49	E	I
74	4A	V	j
75	4B	ffi	K
76	4C	C	L
77	4D	SJ	M
78	4E	2	N
79	4F	C	0
80	50	^	P
81	51	IB]	Q
82	52	Q	R
83	53	W	S
84	54	Hj	T
85	55	ffl	U
86	56	J£	V
87	57	O	W
88	58	|5|	*X*
89	59	d	Y

Dec	Hex	Uppercase/Graphics Set	Lowercase/Uppercase Set
90	5A	ffl	Z
91	5B	*	+
92	5C	£1	£J
93	5D	[B	T
94	5E	ff[1]	^
95	5F	15	^
96	60	SHIFT-space	
97	61	H	|]
98	62	y	H
99	63	n	
100	64	•	•
ioi	65	r	r
103	67	fl	•
104	68	ki	kaJ
105	69	Fl	^
106	6A	•	]
107	6B	[B	|_£
108	6C	;H	.
109	6D	[5	^
HO	6E	*h*	ffl
in	6F	^.	y
112	70	ffl	ffl
113	71	ffl	^
H4	72	ffl	T
H5	73	3]	ffi
116	74	Cl	C
H7	75	IJ	I
US	76	I	d
ii9	77		n
121	79	U	Q
122	7A	•	|Z

Dec	Hex	Uppercase/Graphics Set	Lowercase/Uppercase Set
123	7B		•
124	7C		
125	7D		ffl
126	?E	F]	
127	7F	n	

Keycodes

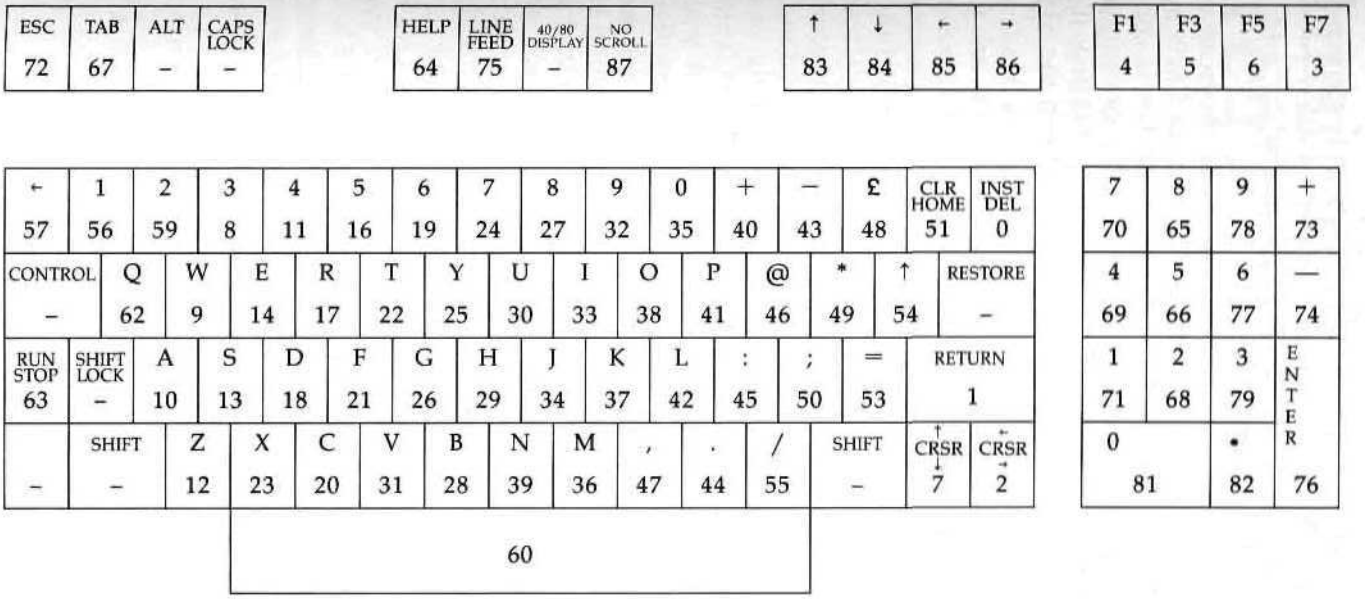
The 128 keyboard is arranged electrically as a matrix of 11 columns X 8 rows of keys. Every 1/60 second, the computer scans the keyboard to see if a key is pressed. If a key is pressed, the keyscan routine generates a value that corresponds to the key's position in the matrix. (The formula is keycode = *column* * 8 + *row,* where *column* has a value from 0 to 10, and *row* has a value from 0 to 7.) If no key is pressed, a value of 88 is generated. This value is stored in location 212. Location 213 will also hold the same value. The contents of this location can be used to determine which key is currently being pressed, as an alternative to using GET (or the Kernal GETIN routine inmachine language). For example, these two lines have the the same effect—to pause the program until any key is pressed:

```
100 GET K$:IF K$="" THEN 100
100 IF PEEK(212)=88 THEN 100
```

Figure C-l gives the keyscan codes for the 128 keyboard. Notice that the figure shows no values for the left and' right SHIFT keys, CONTROL, the Commodore key, and ALT. They are the keys which use the missing codes 15, 52, 58, 61, and 80, respectively. These keys are detected later in the keyscan routine, and location 211 is used to record their status. The values found in that location are as follows:

No shift key pressed	0
SHIFT	1
Commodore	2
CONTROL	4
ALT	**8**
CAPS LOCK	16

Figure C-1. Commodore 128 Keycodes



For this location the values are cumulative. If you press both SHIFT and CONTROL, the location will contain 1 + 4 = 5. Holding down SHIFT, CONTROL, Commodore, and ALT together results in a value of 15.

The other "missing" keys are not part of the keyscan matrix. RESTORE is connected to the CIA #2 chip, and acts by generating an NMI interrupt. The 40/80 column key is connected to the MMU chip and is read and acted upon only at power-on or reset. The SHIFT LOCK key is not scanned; it's merely a switch that has the effect of holding down the SHIFT key.

Musical Note Frequencies

The following table lists the frequencies for the standard musical notes which can be produced by the 128's SID sound chip, along with the SID frequency register values required to produce these sounds. The register values shown are for 128s using the NTSC (North American) video system. Refer to the discussion of the SID chip in Chapter 8 for more information on sound programming.

Although music is often considered an art rather than a science, there is a precise mathematical relationship between the notes in the scale. Any two adjacent notes differ in frequency by a factor of 2 t (1 / 12), or about 1.05946. That is, for any given note the frequency of the next higher note will be equal to the current note frequency multiplied by 1.05946, and the frequency of the next lower note will be equal to the current note frequency divided by 1.05946. Furthermore, the frequency of a note in one octave and that of a note with the same letter designation in a higher or lower octave will differ by a factor of 2 times the number of octaves between the notes. For example, by international convention the base frequency in this standard system of musical notation is the A at 440 hertz (octave 3 in the table below). You'll notice that the A in octave 2 has a frequency of 220 hertz (440 / 2), and the A in octave 4 has a frequency of 880 hertz (440 * 2); the A in octave 1 has a frequency of 110 hertz (440 / 4), and the A in octave 5 has a frequency of 1760 hertz (440 * 4). The octave designations shown (0-6) correspond to those used with the O parameter in the BASIC PLAY statement.

Note	**Frequency (hertz)**	**Frequency** register value **(low byte)**	**(high byte)**
Octave 0:			
C	32.70	24/$18	2/$02
C/D	34.65	56/$38	2/$02
D	36.71	90/$5A	2/$02
D/E	38.89	126/$7E	2/$02
E	41.20	164/$A4	2/$02

Note	Frequency (hertz)	Frequency register value (low byte)	(high byte)	
F	43.65	204/$CC	2/$02	
F/G	46.25	247/$F7	2/$02	
G	49.00	36/$24	3/$03	
G/A	51.91	84/$54	3/$03	
A	55.00	134/$86	3/$03	
A/B	58.27	188/$BC	3/$03	
B	61.74	245/$F5	3/$03	
Octave 1:				
C	65.41	49/$31	4/$04	
C/D	69.30	113/$71	4/$04	
D	73.42	180/$B4	4/$04	
D/E	77.78	252/$FC	4/$04	
E	82.41	72/$48	5/$05	
F	87.31	152/$98	5/$05	
F/G	92.50	237/$ED	5/$05	
G	98.00	72/$48	6/$06	
G/A	103.83	167/$ A 7	6/$06	
A	110.00	12/$0C	7/$07	
A/B	116.54	120/$78	7/$07	
B	123.47	233/$E9	7/$ 07	
Octave 2:				
C	130.81	98/$62	8/$08	
C/D	138.59	225/$El	8/$08	
D	146,83	105/$69	9/$09	
D/E	155.56	248/$F8	9/$09	
E	164.81	144/$90	10/$0A	
F	174.61	48/$30	ll/$0B	
F/G	185.00	219/$DB	11/$OB	
G	196.00	143/$8F	12/$0C	
G/A	207.65	78/$4E	13/$0D	
A	220.00	25/$19	14/$0E	
A/B	233.08	240/$F0	14/$0E	
B	246.94	211/$D3	15/$0F	
Octave 3:				
C	261.63	196/$C4	16/$10	(middle C)
C/D	277.18	195/$C3	17/$11	
D	293.66	209/$D1	18/$12	
D/E	311.13	240/$F0	19/$13	
E	329.63	31/$1F	21/$15	
F	349.23	97/$61	22/$16	
F/G	369.99	182/$B6	23/$17	
G	392.00	30/$1E	25/$19	

Note	Frequency (hertz)	Frequency register value (low byte)	(high byte)
G/A	**415.30**	157/$9D	26/$lA
A	440.00	50/$32	28/$lC
A/B	466.16	223/$DF	29/$lD
B	493.88	166/$A6	31/$1F
Octave 4:			
C	523.25	136/$88	33/$21
C/D	554.37	134/S86	35/$23
D	587.33	163/$A3	37/$25
D/E	622.25	224/SEO	39/$27
E	659.25	63/$3F	42/$2A
F	698.46	194/$C2	44/$2C
F/G	739.99	107/$6B	47/S2F
G	783.99	61/$3D	50/$32
G/A	830.61	58/$3A	53/$35
A	880.00	100/S64	56/$38
A/B	932.33	190/SBE	59/$3B
B	987.77	76/$4C	63/$3F
Octave S:			
C	1046.50	15/$0F	67/$43
C/D	1108.73	12/$0C	71/$47
D	1174.66	70/$46	75/$4B
D/E	1244.51	191/SBF	79/$4F
E	1318.51	125/$7D	84/$54
F	1396.91	131/S83	89/$59
F/G	1479.98	214/$D6	94/$5E
G	1567.98	122/$7A	100/$64
G/A	1661.22	115/$73	106/$6A
A	1760.00	200/$C8	112/$70
A/B	1864.65	124/$7C	119/$77
B	1975.53	151/S97	126/$7E
Octave 6:			
C	2093.00	30/$1E	134/S86
C/D	2217.46	24/$18	142/$8E
D	2349.32	139/$8B	150/$96
D/E	2489.01	127/$7F	159/$9F
E	2637.02	251/$FB	168/$A8
F	2793.82	7/$07	179/$B3
F/G	2959.95	172/$ AC	189/$BD
G	3135.96	243/$F3	200/SC8
G/A	3322.44	230/$E6	212/$D4
A	3520.00	143/$8F	225/$El
A/B	3729.31	249/SF9	238/$EE
B	3951.07	47/$2F	253/$FD

64/128 Memory Map Cross Reference

The following list provides a cross reference of the Commodore 64 and Commodore 128 memory maps. In addition to the RAM locations shown, all I/O chips appear in the 128 at the same addresses as in the Commodore 64. (When the 128 is used in Commodore 64 mode, the two extra VIC chip registers and the VDC 80-column chip are still available, although they must be programmed directly, since there are no routines to support them.) Also, the Kernal jump table at 65409-65525/ $FF81-$FFF5 is common to both the Commodore 64 and 128.

S

Commodore 64 address	Commodore 128 address	Function
0/$00	0/$00	Processor on-chip I/O port data direction register
l/$01	l/$01	Processor on-chip I/O port data register
3-4/$03-$04	4474-4475/$117A-$117B	Vector: Routine to convert a number from floating point to signed integer
5-6/$05-$06	4476-4477/$117C-$117C	Vector: Routine to convert a number from integer to floating point
7/$07	9/$09	Search character for scanning BASIC text input
8/$08	10/$OA	Search character for statement terminator or quote
9/$09	11/$OB	Column position of the cursor before the last TAB or SPC
1O/SOA	12/$0C	BASIC LOAD/VERIFY flag
11/$OB	13/$0D	[ndex into the text input buffer / number of array subscripts
12/$0C	H/$0E	Flag for routines that locate or build an array
13/$0D	15/$0F	Flag: Type of data {string OT numeric)
14/$0E	16/$10	Flag: Type of numeric data (integer or floating point)
15/$0F	17/$11	Flag for LIST, garbage collection, and program tokenizatiort
16/$10	I8/$12	Flag: Subscript reference to an array or user-defined function call (FN)
17/$11	19/$13	Input source flag (GET, READ, or INPUT)
18/$12	20/$14	Flag: Sign of the result of the TAN or SIN function
19/$13	2I/$15	Current I/O channel {CMD logical file) number
20-21/$14-$15	22-23/$16-$17	Integer line number value
22/$16	24/$18	Pointer to the next available space in the temporary string stack
23-24/$17-$18	25-26/$19-$1A	Pointer to the address of the last string in the temporary string stack
25-33/$19-$21	27-35/$lB-$23	Descriptor stack for temporary strings
34-37/$22-$25	36-39/$24-$27	Miscellaneous temporary pointers and save area
38-42/$26-$2A	40-44/$28-$2C	Floating point multiplication work area
43-44/$2B-$2C	45-46/$2D-$2E	Pointer to the start of BASIC program text
45-46/$2D-$2E	47-48/$2F-$30	Pointer to the start of the variable storage area
47-48/$2F-$30	49-50/$31-$32	Pointer to the start of the array storage area
49-50/$31-$32	51-52/$33-$34	Pointer to the start of the area available for string storage

Commodore 64 address	Commodore 128 address	Function
51-52/$33-$34	53-54/$35-$36	Pointer to the bottom of the string pool
53-54/$35-$36	55-56/$37-$38	Temporary pointer for strings
55-56/$37-$38	57-58/$39-$3A	Pointer to the highest address used by BASIC
57-58/$39-$3A	59-60/$3B-$3C	Current BASIC line number
59-60/$3B-$3C	4608-4609/$1200-$12ul	Previous BASIC line number
61-62/$3D-$3E	4610-4611/$1202-$1203	Pointer to the address of the current BASIC statement
63-64/$3F-$40	65-66/$41-$42	Current DATA line number
65-66/$41-$42	67-68/$43-$44	Pointer to the address of the current DATA item
67-68/$43-$44	69-70/$45-$46	Pointer to the source of GET, READ, or INPUT
69-70/$45-$46	71-72/$47-$48	Current BASIC variable name
71-72/$47-$48	73-74/$49-$4A	Pointer to the current BASIC variable value
73-74/$49-$4A	75-76/$4B-$4C	Temporary pointer to the index variable used by FOR
75-76/$4B-$4C	77-78/$4D-$4E	Math operator table displacement
77/$4D	79/$4F	Mask for comparison operation
78-79/$4E-$4F	80-81/$50-$5I	Pointer to the current FN descriptor
80-82/$50-$52	82-84/$52-$54	Temporary pointer to the current string descriptor
83/$53	n.a.	Constant for garbage collection
84-86/$54-$56	86-88/$56-$58	Jump to function instruction
87-96/$57-$60	89-98/$59-$62	BASIC numeric work area
97-101/$61-$65	99-103/$63-$67	Floating point accumulator #1
102/$66	104/$68	Floating point accumulator #1 sign
103/S67	105/$69	Number of terms in a series evaluation
104/$68	991/$03DF	Floating point accumulator #1 overflow digit
105-109/$69-$6D	106-110/$6A-$6E	Floating point accumulator #2
110/S6E	111/$6F	Floating point accumulator #2 sign
111/$6F	112/$70	Result of a sign comparison of FAC1 to FAC2
112/$70	113/$71	Low byte of floating point accumulator #1 (for rounding)
113-114/$71-$72	114-115/$72-$73	Series evaluation pointer
115-138/$73-$8A	896-926/$0380-$039E	Subroutine to get next BASIC text character (CHRGET)
139-143/$8B-$8F	4635-4639/$121B-$121F	RND function seed value

	Commodore 64 address	Commodore 123 address	Function
to	144/$90	144/$90	Kernal I/O status word (ST)
	145/S91	145/$91	Flag: Was STOP key pressed?
	H6/$92	146/$92	Timing constant for tape reads
	147/$93	147/$93	Flag for Kerna! LOAD routine (0 = LOAD, 1 = VERIFY)
	148/$94	148/$94	Serial bus buffered character flag
	149/$95	149/$95	Buffered character for serial bus
	150/$96	150/$96	Cassette block synchronization number
	151/$97	**151/$97**	Temporary register save area
	152/$98	152/$98	Number of open files / index into the file tables
	153/$99	153/$99	Default input device (set to 0 for the keyboard)
	154/$9A	154/$9A	Default output (CMD) device (set to 3 for the screen)
	155/S9B	155/$9B	Tape character parity
	156/$9C	156/$9C	Flag: Tape byte received
	157/$9D	157/$9D	Kernal message control flag
	158/$9E	158/$9E	Tape pass 1 error log index
	159/$9F	159/$9F	Tape pass 2 error log correction index
	160-162/$ A0-$A2	160-162/$ A0-$A2	Software jiffy clock
	163-164/$A3-$A4	163-164/$A3-$A4	Cassette/serial work bytes
	165/$A5	165/$A5	Cassette synchronization character countdown
	166/$A6	166/$A6	Count of characters in tape I/O buffer
	167/$A7	167/SA7	RS-232 input bits / cassette temporary storage area
	168/$A8	168/$A8	RS-232 input bit count / cassette temporary storage
	169/$A9	169/$A9	RS-232 start bit received flag
	170/$AA	170/$AA	RS-232 input byte buffer / cassette temporary storage
	171/$AB	171/$AB	RS-232 input parity / cassette leader counter
	172-173/$AC-$AD	172-173/$AC-$AD	Pointer to the starting address of a load / screen scrolling
	174-175/$AE-$AF	174-175/$AE-$AF	Pointer to the ending address of the load (end of program)
	176-177/$BO-$B1	176-177/$B0-$BI	Tape timing
	178-179/$B2-$B3	178-179/$B2-SB3	Pointer to start of tape buffer
	180/$B4	180/$B4	RS-232 output bit count / cassette temporary storage

Commodore 64 address	Commodore 12S address	Function
181/SB3	131/$B5	RS-232 next bit to send / tape EOT flag
182/SB6	182/$B6	RS-232 output byte buffer
183/$B7	183/$B7	Length of current filename
184/$B8	184/$B8	Current logical file number
185/$B9	185/$B9	Current secondary address
186/$BA	186/$BA	Current device number
187-188/$BB-$BC	187-188/$BB-$BC	Pointer to current filename
189/$BD	189/$BD	RS-232 output parity / cassette temporary storage
190/$BE	190/$BE	Cassette read/write block count
191/$BF	191/$BF	Tape input byte buffer
192/$C0	192/$C0	Tape motor interlock
193-I94/$C1-$C2	193-194/$C1-$C2	I/O start address
195-196/$C3-$C4	195-196/$C3-$C4	Tape load temporary addresses
197/$C5	213/$D5	Matrix coordinate of last key pressed
198/$C6	208/$D0	Number of characters in keyboard buffer (queue)
199/$C7	243/$F3	Reverse character flag
200/$C8	2608/$OA30	Pointer to end of logical line for input
201-202/$C9-$CA	232-233/$E8-$E9	Cursor column and row position at start of input
203/$CB	212/$D4	Matrix coordinate of current key pressed
204/$CC	2599/$0A27	Cursor blink enable flag (0 = flashing cursor)
205/$CD	2600/$0A28	Timer: Countdown to blink cursor
206/$CE	2601/$0A29	Character under cursor
207/$CF	2598/$0A26	Flag: Was last cursor blink on or off?
208/$D0	2]4/$D6	Flag: Input from keyboard or screen
209-210/$Dl-$D2	224-225/$E0-$El	Pointer to the address of the current screen line
211/$D3	236/$EC	Cursor column on current line
212/$D4	244/$F4	Quote mode flag (0 — off)
213/$D5	238/$EE	Maximum length of physical screen line
2H/$D6	235/$EB	Current cursor physical line number
215/$D7	240/$FO	Temporary storage area for last character printed

ON

g;	Commodore 64 address	Commodore 128 address	Function
^	216/$D8	245/$F5	Flag: Insert mode (any number greater than 0 is the number of inserts)
	217-242/$D9-$F2	n.a.	Screen line link table / editor temporary storage
	243-244/$F3-$F4	226-227/$E2-$E3	Pointer to the address of the current screen color RAM location
	245-246/$F5-$F6	204-205/$CC-$CD	Pointer to current keyboard decode table
	247-248/$F7-$F8	200-201 /$C8-$C9	Pointer to RS-232 input buffer
	249-250/$F9-$FA	202-203/$CA-$CB	Pointer to RS-232 output buffer
	251-254/$FB-$FE	251-254/$FB-$FE	Free zero-page locations for user programs
	255-266/$FF-$010A	255-266/$FF-$010A	Work area for floating point to ASCII conversion
	256-317/$0100-$013D	256-217/$0100-$013D	Tape input error log
	256-511/$0100-$01FF	256-511/$0100-$01FF	Microprocessor stack area
	512-600/$0200-$0258	512-671/$0200-$02AO	BASIC input buffer
	601-610/$0259-$0262	866-875/$0362-$036E	Table of active logical file numbers
	6Jl-620/$0263-$026C	876-885/$036C-$0375	Table of device numbers for each logical file
	621-630/$026D-$0276	886-895/$0376-$037F	Table of secondary addresses for each logical file
	631-640/$0277-$0280	842-851/$034A-$0353	Keyboard buffer
	641-642/$0281-$0282	2565-2566/$OA05-$OA06	Pointer to start of memory
	643-644/$0283-$0284	2567-2568/$OA07-$OA08	Pointer to end of memory
	645/$0285	2574/$OA0E	IEEE time-out flag
	646/$0286	241/$F1	Current foreground color for text
	647/$0287	2602/$OA2A	Color of character undeT cursor
	648/S0288	2619/$0A3B	Top page of screen memory
	649/$0289	2592/$0A20	Maximum keyboard buffer size
	650/$028A	2594/$0A22	Flag: Which keys will repeat?
	651/$028B	2595/S0A23	Counter for timing the delay between key repeats
	652/$028C	2596/$0A24	Counter for timing the delay until the first key repeat begins
	653/$028D	211/$D3	Current SHI FT/CTRL/Commodore keypress
	654/$028E	n.a.	Last pattern of SHIFT/CTRL/Commodore keypress
	655-656/$028F-$0290	826-827/$033A-$033B	Vector to keyboard table setup routine
	657/S0291	247/$F7	Flag: Enable/disable character set switching with SHIFT-Commodore

Commodore 64 address	Commodore 128 address	Function
658/$0292	248/$F8	Flag: Screen scroll enabled
659/$0293	2576/$OA10	RS-232 control register
660/S0294	2577/$0All	RS-232 command register
661-662/$0295-$0296	2578-2579/$OA12-$0A13	RS-232 Nonstandard bit timing
663/S0297	2780/$0A14	RS-232 status register
664/$0298	2581/$0A15	Number of bits left to be sent/received
665-666/$0299-$029A	2582-2583/$OA16-$0A17	Time required to send a bit
667/$029B	2584/$0A18	Index to end of RS-232 input buffer
668/$029C	2585/S0A19	Index to start of RS-232 input buffer
669/$029D	2586/$0AlA	Index to start of RS-232 output buffer
670/$029E	2587/S0A1B	Index to end of RS-232 output buffer
671-672/$029F-$02A0	2569-2570/$OA09-$OAOA	Save area for IRQ vector during cassette operations
673/$02Al	2575/$0AOF	RS-232 interrupts enabled
674/$02A2	2571/S0A0B	CIA #1 control register B storage during cassette operations
675/$02A3	2572/S0A0C	Save area for CIA #1 interrupt control register during cassette read
676/$02A4	2573/S0A0D	Save area for CIA #1 control register A during cassette read
677/S02A5	223/$DF	Temporary index to the next 40-column line for screen scrolling
678/S02A6	2563/$0A03	PAL/NTSC Hag
768-769/$0300-$0301	768-769/$0300-$03ul	Vector to print BASIC error message routine
770-771/$0302-$0303	770-771/$0302-$0303	Vector to main BASIC program loop
772-773/$0304-$0305	772-773/$0304-$0305	Vector to routine that crunches the ASCII keywords into tokens
774-775/$0306-$0307	774-775/$0306-$0307	Vector to routine that lists BASIC program tokens
776-777/$0308-$0309	776-777/$0308-$0309	Vector to routine that executes next BASIC program token
778-779/$030A-$030B	778-779/$030A-$030B	Vector to routine that evaluates a single-term arithmetic expression
780/$030C	6/$06	Storage for A register (accumulator)
781/$030D	7/$07	Storage for X index register
782/$030H	8/$08	Storage for Y index register
783/$030F	5/$05	Storage for status register

Commodore 64 address	Commodore 128 address	Function
784–786/$0310–$0312	4632–4634/$1218–$121A	USR routine vector
788–789/$0314–$0315	788–789/$0314–$0315	Indirect vector: IRQ interrupt routine
790–791/$0316–$0317	790–791/$0316–$0317	Indirect vector: BRK instruction interrupt
792–793/$0318–$0319	792–793/$0318–$0319	Indirect vector: Nonmaskable interrupt
794–795/$031A–$031B	794–795/$031A–$031B	Indirect vector: Kernal OPEN routine
796–797/ $031C–$031D	796–797/$031C–$031D	Indirect vector: Kernal CLOSE routine
798–799/$031E–$031F	798–799/$031E–$031F	Indirect vector: Kernal CHKIN routine
800–801/$0320–$0321	800–801/$0320–$0321	Indirect vector: Kernal CKOUT routine
802–803/$0322–$0323	802–803/$0322–$0323	Indirect vector: Kernal CLRCH routine
804–805/$0324–$0325	804–805/$0324–$0325	Indirect vector: Kernal CHRIN routine
806–807/$0326–$0327	806–807/$0326–$0327	Indirect vector: Kernal CHROUT routine
808–809/$0328–$0329	808–809/$0328–$0329	Indirect vector: Kernal STOP routine
810–811/$032A–$032B	810–811/$032A–$032B	Indirect vector: Kernal GETIN routine
812–813/$032C–$032D	812–813/$032C–$032D	Indirect vector: Kernal CLALL routine
814–815/$032E–$032F	786–787/$0312–$0313	Indirect vector: User-defined command
816–817/$0330–$0331	816–817/$0330–$0331	Indirect vector: Kernal LOAD routine
818–819/$0332–$0333	818–819/$0332–$0333	Indirect vector: Kernal SAVE routine
828–1019/$033C–$03FB	2816–3071/$0B00–$0BFF	Cassette buffer

n.a. = not applicable

BASIC Keyword Index

The following table lists all of the BASIC 7.0 keywords in alphabetical order, along with their corresponding tokens and the addresses of the routines called to perform each token's operation.

Keyword	Token	Address
ABS	182/$B6	35972/$8C84
AND	175/$AF	19593/$4C89
APPEND	254/$FE + 14/$0E	41268/$A134
ASC	198/$C6	34423/$8677
ATN	193/$C1	38067/$94B3
AUTO	220/SDC	22901/$5975
BACKUP	246/$F6	41852/$A37C
BANK	254/$FE + 2/$02	27593/$6BC9
BEGIN	254/$FE + 24/$18	[1]
BEND	254/$FE + 25/$19	21135/S528F
BLOAD	254/$FE + 17/$ll	41496/$A218
BOOT	254/$FE + 27/$lB	29493/$7335
BOX	225/$El	25271/$62B7
BSAVE	254/$FE + l6/$10	41416/$A1C8
BUMF	206/$CE + 3/$03	33660/S837C
CATALOG	254/$FE+12/$0C	41086/$A07E
CHAR	224/$E0	26583/$67D7
CHR$	199/$C7	34239/$85BF
CIRCLE	226/$E2	26254/$668E
CLOSE	160/$AO	37274/$919A
CLR	156/$9C	20984/$51F8
CMD	157/$9D	21824/$5540
COLLECT	243/$F3	41775/SA32F
COLLISION	254/$FE + 23/$17	29028/$7164
COLOR	231/SE7	27106/$69E2
CONCAT	254/$FE+19/$13	41826/$A362
CONT	154/$9A	23136/$5A60
COPY	244/$F4	41798/$A346
COS	190/$BE	37897/$9409
DATA	131/S83	21135/I528F
DCLEAR	254/$FE + 21/$15	41762/$A322

Keyword	Token	Address
DCLOSE	254/$FE+15/$0F	41327/$A16F
DEC	2O9/$D1	32886/$8076
DEF	150/$96	34042/$84FA
DELETE	247/$F7	24199/$5E87
DIM	134/$86	22651/$587B
DIRECTORY	238/$EE	41086/$A07E
DLOAD	240/$F0	41383/$A1A7
DO	235/$EB	24544/$5FE0
DOPEN	254/$FE+13/$0D	41245/$A11D
DRAW	229/$E5	26519/$6797
D5AVE	239/$EF	41356/$A18C
DVERIFY	254/$FE + 20/$14	41380/$A1A4
ELSE	213/$D5	21393/$5391
END	128/$80	19405/$4BCD
ENVELOPE	254/$FE + 10/$OA	28865/$70Cl
ERR$	211/$D3	33014/$80F6
EXIT	237/$ED	24633/$6039
EXP	189/IBD	36915/$9033
FAST	254/$FE + 37/$25	30643/$77B3
FETCH	254/$FE + 33/$21	43556/$AA24
FILTER	254/$FE + 3/$03	28742/$7046
FN	165/$A5	34107/$853B
FOR	129/$81	24057/$5DF9
FRE	184/$B8	32768/$8000
GET	161/$A1	22034/$5612[2]
GO	203/$CB	23101/$5A3D
GOSUB	141/$8D	22991/$59CF
GOTO	137/$89	23003/$59DB
GRAPHIC	222/$DE	27482/$6B5A
GSHAPE	227/$E3	25997/$658D
HEADER	241/$F1	4I575/$A267
HELP	234/$EA	22918/$5986
HEX$	210/$D2	33090/S8142
IF	I39/S8B	21189/$52C5
INPUT	133/$85	22H4/$5662
INPUT#	132/$84	22088/$5648
INSTR	212/SD4	39361/S99C1
INT	181/$B5	36091/$8CFB
JOY	207/$CF	33283/$8203
KEY	249/$F9	24842/$610A
LEFTS	200/$C8	34262/$85D6
LEN	195/$C3	34408/$8668
LET	136/$88	21446/$53C6
LIST	155/$9B	20706/$50E2
LOAD	147/$93	37164/$912C

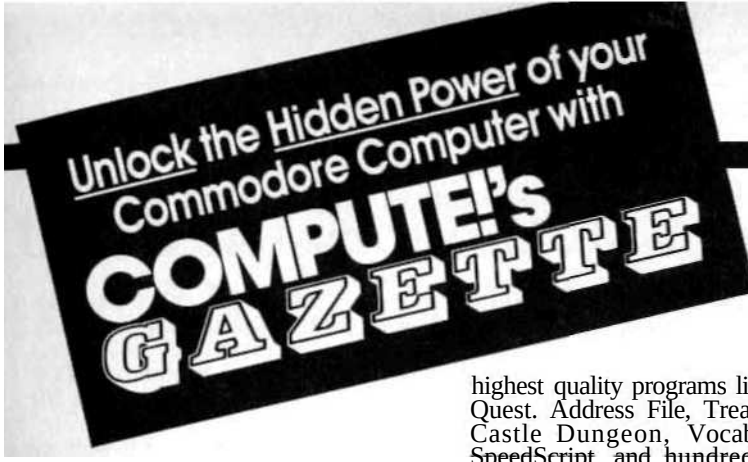
Keyword	Token	Address
LOCATE	23O/SE6	26965/$6955
LOG	188/$BC	35274/$89CA
LOOP	236/$EC	24714/$608A
MID$	202/$CA	[3]
MONITOR	250/$FA	45056/$B000
MOVSPR	254/$FE + 6/$06	27846/$6CC6
NEW	162/$A2	20950/$51D6
NEXT	130/$82	22516/$57F4 [4]
NOT	168/$A8	31024/$7930
OFF	254/$FE + 36/$24	18502/$4846 [5]
ON	145/$91	21411/$53A3 [6]
OPEN	159/$9F	37261/$918D
OR	176/$B0	19590/$4C86
PAINT	223/SDF	25000/$61A8
PEEK	194/$C2	32965/$80C5
PEN	206/$CE+4/$04	33454/$82AE
PLAY	254/$FE + 4/$04	28129/S6DE1
POINTER	206/$CE + 10/$0A	33530/$82FA
POKE	151/S97	32997/$80E5
POS	185/$B9	34000/$84DO
POT	206/$CE + 2/$02	33357/$824D
PRINT	153/$99	21850/$555A
PRINT*	152/$98	21818/$553A
PUDEF	221/$DD	24372/$5F34
QUIT	254/$FE + 30/$lE	18502/$4846 [5]
RCLR	205/$CD	33179/$819B
RDOT	208/$D0	39692/$9B0C
READ	135/$87	22185/S56A9
RECORD	254/$FE + 18/$12	41687/$A2D7
REM	143/$8F	21149/$529D
RENAME	245/$F5	41838/$A36E
RENUMBER	248/$F8	23288/$5AF8
RESTORE	140/$8C	23242/$5ACA
RESUME	214/$D6	24418/$5F62
RETURN	142/$8E	21090/$5262
RGR	204/$CC	33154/$8182
RIGHTS	201/$C9	34314/$860A
RND	187/$BB	33844/$8434
RREG	254/$FE + 9/$09	22717/$58BD
RSPCOLOR	206/$CE + 7/$07	33633/$8361
RSPPOS	206/$CE + 5/$05	33687/$8397
RSPRITE	206/$CE + 6/$06	33566/$831E
RUN	138/$8A	23195/$5A9B
RWINDOW	206/$CE + 9/$09	33799/$8407
SAVE	148/$94	37138/$9112

Keyword	Token	Address
SCALE	233/$E9	26976/$6960
SCNCLR	232/$E8	2 725 7/$ 6A 79
SCRATCH	242/$F2	41633/$A2A1
SGN	180/$B4	35941/$8C65
SrN	191/$BF	37904/$9410
SLEEP	254/$FE + ll/$0B	27607/$6BD7
SLOW	254/$FE + 38/$26	30660/$77C4
SOUND	218/$DA	29164/$71EC
SPC(	166/$A6	[7]
SPRCOLOR	254/$FE + 8/$08	29072/$7190
SPRDEF	254/$FEH-29/$lD	29554/$7372
SPRITE	254/$FE + 7/$07	27727/$6C4F
SPRSAV	254/$FE + 22/$16	30444/$76EC
SQR	186/$BA	36791/$8FB7
SSHAPE	228/$E4	25643/$642B
STASH	254/$FE + 31/$lF	43551/$AA1F
STEP	169/$A9	[8]
STOP	144/$90	19403/$4BCB
STR$	196/$C4	34222/$85AE
SWAP	254/$FE + 35/$23	43561/$AA29
SYS	158/$9E	22661/$5885
TAB(	163/$A3	m
TAN	192/$C0	37977/$9459
TEMPO	254/$FE + 5/$05	28631/$6FD7
THEN	167/$A7	[9]
TO	164/$A4	[10]
TRAP	215/$D7	24397/$5F4D
TROFF	217/$D9	22711/$58B7
TRON	216/$D8	22708/$58B4
UNTIL	252/$FC	
USING	251/$FB	m
USR	I83/$B7	4632/$1218
VAL	197/$C5	32842/$804A
VERIFY	149/$95	37161/$9129
VOL	219/$DB	29125/$71C5
WAIT	146/$92	27693/$6C2D
WHILE	253/$FD	[11]
WIDTH	254/$FE + 28/$lC	29110/$71B6
WINDOW	254/$FE + 26/$lA	29388/$ 72CC
XOR	206/$CE + 8/$08	33761/$83E1
+	170/$AA	34888/$8848
*	171/$AB	34865/$8831
/	172/$AC	35367/$8A27
	173/$ AD	35660/$8B4C
	174/$AE	36801/S8FC1

Keyword	Token	Address
<	179/$B3	19638/$4CB6
	178/$B2	19638/$4CB6
>	177/$B1	19638/$4CB6
K	255/$FF	[12]

Notes:

1. Normally handled during IF execution. When encountered outside IF processing, the target routine merely prints a SYNTAX ERROR message.
2. There are no separate tokens for the keywords GET* and GETKEY; both are handled as special cases during GET execution. GET* is represented by a GET token followed by a # character (code 35/$23). GETKEY is represented by a GET token followed by a KEY token (249/SF9).
3. Can be used as either a statement or a function. The routine at 22785/$5901 handles processing as a statement; when a function, the routine at 34332/S861C is used.
4. Can also be used in conjunction with the RESUME statement.
5. Defined but not implemented. The target routine merely prints an UNIMPLEMENTED COMMAND ERROR message.
6. Can also be used in conjunction with most disk commands to specify the device (unit) number.
7. Processed during PRINT execution.
8. Processed during FOR execution.
9. Processed during IF execution.
10. Processed during the execution of several other keywords which accept TO as part of a valid statement: DRAW, FOR, GO, and assorted disk commands (BACKUP, BSAVE, CONCAT, COPY, and RENAME).
11. Processed during DO or LOOP execution.
12. Processed during the operand evaluation routine (EVAL), 30935/$78D7.



More fun...More challenge...
Mote all new programs
each and every month.

Subscribe to *COMPUTEVs Gazette* through this special money-saving introductory offer—and start unleashing the full power of your Commodore computer.

Month after month, look to *COMPUTERS Gazette* to deliver the latest inside word on everything from short programming tips to the best new software. Our expert analysis and insights mean you have more fun...get more enjoyment...more of what you bought your computer for.

As a subscriber, you also receive up to 20 all-new, action-packed programs each month. Every big issue of *COMPUTEVs Gazette* comes complete with a steady supply of the most useful, the most entertaining, the highest quality programs like Number Quest. Address File, Treasure Hunt. Castle Dungeon, Vocab Builder, SpeedScript, and hundreds of other educational, home finance, and game programs.

So subscribe today—and unleash the hidden power of your Commodore computer, Return the card below—or call 1-800-247-5470 (in Iowa. 1-800-532-1272). Do it now.



Subscription Savings Card

i

YES!

I know a great deal when 1 see one. Sign me up for 12 big issues of *COMPUTEVs Gazette* for |ust $18.1 save 50% off the newsstand price.

n Payment enclosed • Bill me • Charge my VISA/MasterCard

Credit Card #_ _Exp. Date_

Signature-

Name_

Address.

City_ -State. _Zip_

Outside US A please aJJ S6-.U.S (per year (or postage.

Say YES now to COMPUTE'S GAZETTE

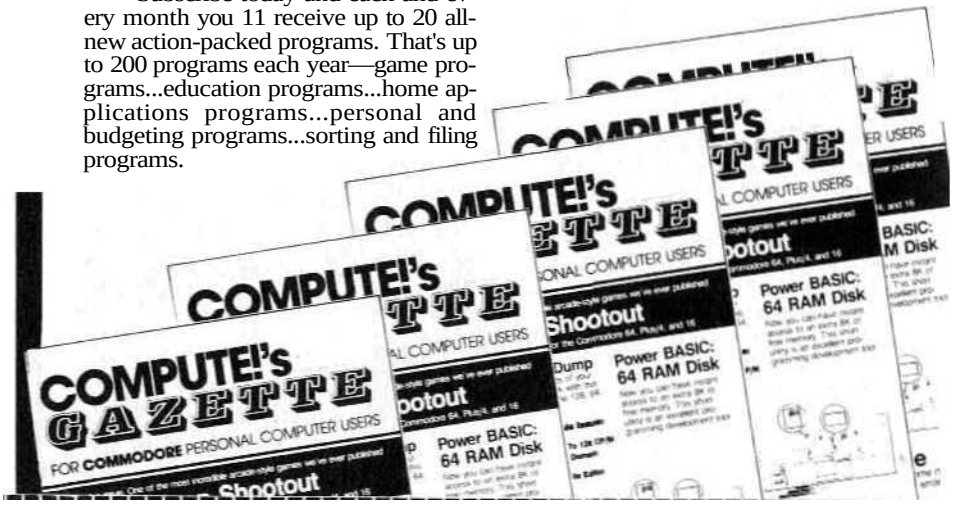
You can search far and wide and you simply *want* find a better magazine...a better source of insightful, stimulating, usable information for your Commodore computer than *COMPUTERS Gazette*.

COMPUTE'S Gazette works harder...digs deeper...researches further—all to help guarantee that you get the absolute most from your Commodore.

Subscribe today and each and every month you 11 receive up to 20 all-new action-packed programs. That's up to 200 programs each year—game programs...education programs...home applications programs...personal and budgeting programs...sorting and filing programs.

Add it up for yourself. Where else can you get exciting programs each month...expert advice...insightful analysis...up-to-the-minute software reviews and so much more—all for just $18.

So why wait. Subscribe now to *COMPUTEfs Gazette—and* get the most from your Commodore computer. Return card below—or call 1-SO0-247-5470 (in Iowa. 1-800-532-1272).



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 7551 DES MOINES. IA

POSTAGE: WILL BE PAID BY ADDRESSEE

COMPUTERS Gazette

PO Box 10775

Des Moines, IA 50347-0775

IJ.I.IL...ll..J..M...IM.,j...n.<.i.i.i...r.ii

Index of Locations and Routines

Chapter 2: Common Working Storage

	SO	Data direction register for processor's on-chip I/O port
1	$01	Data register for processor's on-chip I/O port
2	$02	Target bank for JMPFAR and JSRFAR
3-4	$03-$04	Target address for JMPFAR and JSRFAR
5	$05	Status register storage for JMFFAR and JSRFAR
6	$06	Accumulator storage for JMPFAR and JSRFAR
7	$07	X register storage for JMPFAR and JSRFAR
g	$08	Y register storage for JMPFAR and JSRFAR
9	$09	Stack pointer storage for JSRFAR and monitor
9	$09	Working storage for various routines
10	$0A	Working storage for various routines
11	$0B	Current screen column for TAB and SPC calculations
12	$0C	BASIC LOAD/VERIFY flag
13	$0D	Working storage for various routines
14	$0E	Array dimension flag
15	$0F	Variable type flag
16	$10	Numeric type flag
17	$11	Working storage for various routines
18	$12	Integer/subscript prohibit flag
19	$13	Input source flag
20	$14	Comparison type flag/Tangent sign flag
21	$15	Logical file number for BASIC input and output
22-23	$16-$17	Integer value of ASCII digit string
24	$18	Pointer into temporary string descriptor stack
25-26	$19-$1A	Pointer to most recent descriptor stack entry
27-35	$lB-$23	Temporary string descriptor stack
36-37	$24-$25	Multipurpose address pointer
38-39	$26-$27	Multipurpose address pointer
40-44	$28-$2C	Temporary storage area for multiplication and division
45-46	$2D-$2E	Start-of-BASIC-program pointer
47-48	$2F-$30	Start-of-variables pointer
49-50	$31-$32	Start-of-arrays pointer
51-52	$33-$34	Start-of-free-memory pointer
53-54	$35-$36	Bottom-of-string-space pointer
55-56	$37-$38	Temporary pointer into the string pool
57-58	$39-$3A	Top-of-memory pointer
59-60	$3B-$3C	Current BASIC line number
61-62	$3D-$3E	Pointer for main BASIC character retrieval routine
63-64	$3F-$40	Working pointer for varoious routines
65-66	$41-542	line number of current DATA statement
67-68	$43-$44	Pointer to next DATA item
69-70	$45-$46	Text pointer for input
71-72	$47-$48	Current variable name

73-74	$49-$4A	Pointer to variable descriptor
75-76	$4B-$4C	Variable descriptor pointer and working storage
77-78	$4D-$4E	Temporary storage for text pointer
79	$4F	Relational operator flag
80-81	$50-$51	Defined function pointer and working pointer
80-84	$50-$54	Temporary storage for floating point value
82-83	$52-$53	Variable address storage and working pointer
85	$55	HELP flag
86-88	$56-$58	BASIC function execution vector
89-93	$59-$5D	Floating point work area
90-91	$5A-$5B	Multipurpose working pointer
92-93	$5C-$5D	Multipurpose address pointer
93-95	$5D-$5F	String length and pointer for MID$
94-98	$5E-$62	Temporary storage for floating point value
94-95	$5E-$5F	Working pointer for garbage collection
95	$5F	Decimal point position
96-98	$60-$62	Substring length and pointer for MID$
96-104	$60-$68	Monitor zero-page pointers and working storage
97-98	$61-$62	Multipurpose address pointer
99-103	$63-$67	Floating point accumulator 1
304	$68	Sign of FAC1
105	$69	Sign flag during conversion/Count of terms in series evaluation
106-110	$6A-$6E	Floating point accumulator 2
111	$6F	Sign of FAC2
112	$70	Sign comparison flag
112-113	$70-$71	Multipurpose address pointer
113	$71	Rounding flag for FAC1
114-115	$72-$73	Multipurpose address pointer and working storage
116-117	$74-$75	Step value for autoincrement
118	$76	Graphics area flag
119	$77	General purpose working storage
120	$78	String offset pointer
121	$79	Multipurpose temporary storage
122	$7A	Index into input buffer for monitor
122-124	$7A-$7C	Descriptor for disk error string DS$
125-126	$7D-$7E	BASIC runtime stack pointer
127	$7F	RUN mode flag
128	$80	Decimal point position
128-129	$80-$81	Parameter flags for DOS support commands
130	$82	Storage for processor stack pointer
131	$83	Color source for current graphics command
132	$84	Color source 2 storage
133	$85	Color source 3 storage
134	$86	Current foreground color (source 1) storage
135-136	$87-$88	Horizontal scaling factor
137-138	$89-$8A	Vertical scaling factor
139	$8B	PAINT mode flag
140-141	$SC-$8D	Address pointer for graphics routines
142-143	$8E-$8F	Temporary storage for graphics routines
144	$90	Status flag for tape and serial bus operations
145	$91	Scan value of STOP key column
146	$92	Tape timing adjustment factor
147	$93	Kernal load/verify flag/Monitor operation flag
148	$94	Serial deferred character flag
K 9	$95	Serial character buffer

150	$96	Cassette block synchronization count
151	$97	Temporary register storage
152	$98	Number of files currently open
153	$99	Current input device
154	$9A	Current output device
155	$9B	Tape character parity
156	$9C	Tape dipole received flag
157	$9D	Kernal message control flag
158	$9E	Tape pass 1 error-log pointer
159	$9F	Tape pass 2 error-log pointer
160-162	$A0-$A2	Software jiffy clock
163	$A3	Tape: Count of bits to be read or written Serial: EOI flag
164	$A4	Tape: Half-cycle indicator Serial: Byte received
165	$A5	Tape: Leader synchronization countdown Serial: Count of bits to send
166	$A6	Pointer into cassette buffer
167	$A7	Tape: Leader clipole count/block indicator RS-232: Current bit received
168	$A8	Tape: Half-cycle indicator for writing/error flag for reading RS-232: Count of bits remaining to be received
169	$A9	Tape: Word marker flag/half-cycle flag RS-232: Start bit received flag
170	$AA	Tape: Read phase flag RS-232: Assembly byte for received bits
171	$AB	Tape: Leader dipole counter / checksum work byte RS-232: Received byte parity
172-173	$AC-$AD	Kernal working address pointer
172-175	$AC-$AF	Work area for disk booting
174-175	$AE-$AF	Kernal working storage: Used to hold the ending address for SAVE [$F53E]
176	$B0	Kernal address pointer
177	$B1	TEMP
178-179	$B2-$B3	Pointer to cassette buffer
180	$B4	Tape: leader/data flag RS-232: Count of bits transmitted
181	$B5	Taper Leader completed flag RS-232: Next bit to send
182	$B6	Tape: Error flag / end of block flag RS-232: Character being sent
183	$B7	Length of current filename
184	$B8	Logical file number
185	$B9	Current secondary address
186	$BA	Current device number
187-188	$BB-$BC	Pointer to start of filename
189	$BD	Tape: Byte read from tape / byte to be written to tape RS-232: Parity calculation working storage Serial: Current byte during burst mode load
190	(BE	Block count
191	$BF	Drive number (ASCII) for PHOENIX
192	$C0	Tape motor interlock
193-194	$C1-$C2	Kernal work pointer: Used to hold starting address for save
195-196	$C3-$C4	Kernal work pointer (starting address for load)

197	$C5	Bit read from tape / checksum of block written to tape
198	$C6	Bank where data for save, load, or verify is found
199	$C7	Bank where filename for open, save, load, or verify is found
200-201	$C8-$C9	Pointer to RS-232 input buffer
202-203	$CA-$CB	Pointer to RS-232 output buffer
204-205	$CC-$CD	Pointer to current keyboard decode table
206-207	$CE-$CF	Pointer for Kern a 1 PR1MM routine
208	$D0	Number of characters in the keyboard buffer
209	$D1	Number of characters pending from programmable key string
210	$D2	Pointer into the programmable key definition area
211	$D3	Shift key status flag
212	$D4	Current key pressed (matrix value)
213	$D5	Last key pressed
214	$D6	Input source flag
215	$D7	Active screen flag
216	$D8	Mode flag for 40-column screen
217	$D9	CHAREN bit shadow
218-223	SDA-$DF	5creen editor zero page work area
224-225	$EO-$E1	Pointer to first screen memory location for current line
226-227	$E2-$E3	Pointer to first attribute memory location for current line
228	$E4	Bottom margin of current window
229	$E5	Top margin of current window
230	$E6	Left margin of current window
231	$E7	Right margin of current window
232	$E8	Cursor row for start of input
233	$E9	Cursor column for start of input
234	$EA	Column of last nonspace character on logical line
235	$EB	Cursor row
236	$EC	Position of cursor within current logical line
237	$ED	Maximum number of rows allowed in output window
238	SEE	Maximum number of columns allowed per row
239	SEF	Character to print
240	$F0	Last character printed
241	$F1	Attribute of current character
242	$F2	Temporary storage for attribute byte
243	$F3	Reverse mode flag
244	$F4	Quote mode flag
245	$F5	Number of pending inserts
246	$F6	Autoinsert mode flag
247	$F7	Case switching/scroll pause control flag
248	$F8	Scroll/link control flag
249	$F9	Bell enable flag
250	$FA	Unused
251-254	tFB-SFE	Unused
255-266	$OOFF-$010A	Assembly area for numeric strings
256-511	$0100-$0IFF	Processor stack area
256-268	$0100-$010C	Assembly area for disk boot command
256-317	$0100-$013D	Tape error log
272-290	$0110-$0122	DOS command work area
291-310	S0I23-S0136	PRINT USING work area
294	$0126	Command type indicator for PLAY processing

311-507	$0137-$01FB	Stack space used by BASIC
512-672	J0200-S02AO	BASIC and monitor input buffer
673	S02A1	Unused
674-686	$02A2-$02AE	Retrieves a value from any bank
687-701	$02AF-$02BD	Stores a value in any bank
702-716	$02BE-$02CC	Compares the accumulator contents against a value from any bank
717-738	$02CD-$02E2	Calls a subroutine in any bank
739-763	$02E3-$02FB	Jumps to a routine in any bank
764-765	$02FC-$02FD	Indirect vector for extended functions
766-767	$02FE-$02FF	Unused indirect vector
768-769	$0300-$0301	Indirect vector in BASIC ERROR routine
770-771	$0302-$0303	Indirect vector in BASIC MAIN routine
772-773	$0304-$0305	Indirect vector in BASIC CRUNCH routine
774-775	$0306-$030;	Indirect vector in BASIC QPLOP routine
776-777	$0308-$0309	Indirect vector in BASIC GONE routine
778-779	$030A-$030B	Indirect vector in BASIC EVAL routine
780-781	$030C-$030D	Indirect vector for tokenizing new keywords
782-783	$030E-$030F	Indirect vector for listing new keywords
784-785	$0310-$0311	Indirect vector for executing new statements
786-787	$0312-$0313	Unused
788-789	$0314-$0315	Indirect vector to IRQ handling routine
790-791	$0316-$0317	Indirect vector to BRK handling routine
792-793	$0318-$0319	Indirect vector to NMI handling routine
794-795	$031A-$031B	Indirect vector in Kernal OPEN routine
796-797	$031C-$031D	Indirect vector in Kemal CLOSE routine
798-799	S031E-S031F	Indirect vector in Kemal CHKIN routine
800-801	$0320-$0321	Indirect vector in Kernal CKOUT routine
802-803	$0322-$0323	Indirect vector in Kernal CLRCH routine
804-S05	$0324-$0325	Indirect vector in Kemal BASIN routine
806-807	$0326-$0327	Indirect vector in Kemal BSOUT routine
808-809	$0328-$0329	Indirect vector in Kernal STOP routine
810-811	$032A-$032B	Indirect vector in Kemal GETIN routine
812-813	$032C-$032D	Indirect vector in Kemal CLALL routine
814-815	$032E-$032F	Indirect vector in monitor EXMON routine
816-817	$0330-$0331	Indirect vector in Kernal LOAD routine
818-819	$0332-$0333	Indirect vector in Kemal SAVE routine
820-821	$0334-$0335	Screen editor indirect vector
822-823	$0336-$O337	Screen editor indirect vector
824-825	$0338-$0339	Screen editor indirect vector
826-827	$033A-$033B	Screen editor indirect vector
828-829	$033C-$033D	Screen editor indirect vector
830-841	$033E-$0349	Screen editor indirect vector
842-851	$034A-$0353	Keyboard buffer
852-861	$0354-$035D	Tab stop bitmap
862-865	$035E-$0361	Line link bitmap
866-875	$0362-$036B	Logical file number table for currently open files
876-885	$036C-$0375	Device number table for currently open files
886-895	$0376-$037F	Secondary address table for currently open files
896-926	$0380-$039E	CHRGET (main BASIC character retrieval routine)
927-977	$O39F-$O3D1	Alternate BASIC character retrieval subroutines
978-980	$03D2-$03D4	Null descriptor
981	$03D5	Bank number for PEEK and POKE
982-985	$03D6-$3D9	Pointers for INSTR evaluation
986	$03DA	String block flag
987-990	$03DB-$03DE	Temporary storage for SHAPE data

991	$03DF	Floating-point overflow byte
992-993	$03E0-$03El	Temporary pointer storage
994	$03E2	Standard bitmap color fill value
995	$03E3	Multicolor bitmap color fill value
996-1007	$03E4-$03EF	Unused
1008-1020	$03FO-$03FC	DMA—CALL execution routine
1021-1023	$03FD-$03FF	Unused

Chapter 3: Bank 0 Working Storage

1024-2023	$0400-07FF	Default VIC screen memory
2024-2559	$0800-$09FF	BASIC runtime stack
2560-2561	$OA00-$0AOl	BASIC restart vector
2562	$OA02	Memory initialization status flag
2563	$0A03	PAL/NTSC flag
2564	$0A04	System initialization status flag
2565-2566	$0A05-$OA06	Kernal MEMBOT pointer
2567-2568	$0AO7-$0A08	Kemal MEMTOP pointer
2569-2570	$OA09-$OAOA	Temporary storage for ILRQ vector during tape operations
2571	$OA0B	CIA #1 control register A log
2572	$OA0C	CIA #1 interrupt control register log
2573	$OA0D	CIA #1 timer A status log
2574	$0A0E	IEEE timeout flag
2575	$0A0F	RS-232 activity flag
2576	$OA10	RS-232 control register
2577	S0A11	RS-232 command register
2578-2579	$0A12-$0A13	RS-232 baud rate factor
2580	S0A14	RS-232 status register
2581	$0A15	RS-232 bit count
2582-2583	$OA16-$0A17	R5-232 baud rate timing constant
2584	$0A18	Index to firs! character in RS-232 input buffer
2585	$0A19	Index to last character in RS-232 input buffer
2586	$OA1A	Index to first character in RS-232 output buffer
2587	$OA1B	Index to last character in RS-232 output buffer
2588	J0A1C	Fast serial mode flag
2589-2591	$OA1D-$OA1F	Software jiffy timer
2592	$0A20	Maximum number of keys in the keyboard buffer
2593	$0A21	Scroll pause flag
2594	$0A22	Key repeat flag
2595	$0A23	Countdown between key repeats
2596	$0A24	Countdown until key repeating begins
2597	$0A25	Delay between case switching repeats
2598	$0A26	Cursor blink flag
2599	$0A27	Cursor enable flag
2600	$0A28	Cursor blink countdown
2601	$0A29	Character under cursor
2602	$0A2A	Color under cursor
2603	$0A2B	VDC cursor mode
2604	S0A2C	VIC text screen and character base
2605	$0A2D	VIC bitmap and video matrix base
2606	$0A2E	Starting page for VDC screen memory
2607	$0A2F	Starting page for VDC attribute memory
2608	$0A3O	Ending row for screen input
2609-2610	$0A31-$OA32	Temporary storage for 80-column memory manipulation

2611	S0A33	Attribute of current cursor position
2612	$0A34	Scan line for screen split
2613	$0A35	Temporary storage for X register
2614	$0A36	Jiffy clock compensation flag
2615	$0A37	Temporary storage for clock rate register
2616	$0A38	Temporary storage for sprite enable register
2617	$0A39	Temporary storage for VIC control register
2618	$0A3A	Custom mode flag
2619	$0A38	Starting page for 40-coiumn screen memory
2620-2621	$0A3C-$0A3D	Working pointer into 80-column memory
2622-2623	$0A3E-$0A3F	Unused
2624-2650	$0A40-$0A5A	Screen editor variable storage for the inactive screen
2651-2655	$0A5B-$0A5F	Unused
2656-2665	$0A60-$0A69	Storage for inactive tab stop bitmap
2666-2669	$0A6A-$0A6D	Storage for inactive line link bitmap
2670-2687	$0A6E-$0A7F	Unused
2688-2703	$OA80-$OA8F	Filename buffer for load, save, or verify
2688-2719	$0A80-$0A9F	Search pattern buffer
2720-2727	$0AA0-$0AA7	Working storage for base conversion
2720-2729	$0AA0-$0AA9	Instruction assembly buffer
2730	$0AAA	Instruction format flag
2731	$0AAB	Instruction length
2732-2734	$OAAC-$0AAE	Three-character mnemonic pattern
2735	$0AAF	Temporary storage for X register
2736	$OAB0	Unused
2737	$0ABl	Calculated opcode
2738	$0AB2	Temporary storage for X register
2739	$0AB3	Transfer direction flag
2740	$0AB4	Digit counter
2741	$0AB5	Temporary storage for parameter conversion
2742	$0AB6	Number of bits per digit for base
2743-2745	$0AB7-$0A39	Monitor temporary storage
2746-2751	$OABA-$0ABF	Unused
2752	$0AC0	Counter for function ROM testing
2753-2756	$0ACl-$0AC4	Table of identifiers for function ROMs
2757	$0AC5	DK_FLAG
2758-2815	S0AC6-$0AFF	Unused
2816-307]	$OBO0-$0BFF	Cassette buffer/disk boot buffer
3072-3327	S0C00-SOCFF	RS-232 input buffer
3328-3583	$0DO0-$0DFF	RS-232 output buffer
3584-4095	$0E00-$OFFF	Sprite pattern storage area
4096-4351	$1000-$lFFF	Programmable key definition storage area
4352-4400	$1100-$1130	DOS command assembly area
4401-4402	$1131-$1132	Bitmapped screen pixel cursor horizontal position
4403-4404	$1133-$1134	Bitmapped screen pixel cursor vertical position
4405-4406	$1135-$U36	Final horizontal pixel position for graphics operations
4407-4408	$1137-$1138	Final vertical pixel position for graphics operations
4409-4455	$1139-$1177	Working storage for assorted graphics routines
4456	$1168	Starting page for character pattern definitions
4457	$1169	Bit counter for shape retrieval
4458	$116A	Scaling flag
4459	$116B	Line width for bitmapped graphics routines
4460	$116C	BOX fill flag
4461	$116D	Bit mask value
4462	$116E	Temporary storage for assorted routines
4463	$116F	Trace mode flag

4464-4467	$1170-$1173	Working storage for RENUMBER
4468	$1174	Loop counter for reading directory entries
4469-4470	$1175-$1176	Block count for directory entry
4471	$1177	Working storage for graphics parameter scaling
4472	$1178	Working storage for graphics parameter evaluation
4473	$1179	Working storage for graphics parameter evaluation
4474-4475	$117A-$117B	Pointer to floating point-to-integer conversion routine
4476-4477	$117C-$117D	Pointer to integer-to-floating poinl conversion routine
4478-4565	$117E-S11D5	Sprite movement control data
4566-4582	$11D6-$1IE6	Shadows for VIC sprite position registers
4583-4584	$11E7-$1IE8	Shadows for VIC sprite collision registers
4585-4586	$11E9-$11EA	Shadow for VIC light pen registers
4587	$11EB	Starting page for alternate character set during CHAK
4588	$11EC	Starting page for default character set during CHAR
4589	$11ED	Channel number for BASIC relative file operations
4590-4607	$11EE-$11FT	Unused
4608-4609	$1200-$1201	line number where program stopped
4610-4611	$1202-$1203	Pointer to the start of current line
4612-4615	$1204-$!207	Character definitions for PRINT USING
4616	$1208	Number of most recent error
4617-4618	$1209-$120A	Line number where most recent error occurred
4619-4620	$120B-$120C	Target line number for TRAP statement
4621	$120D	Temporary storage for high byte of TRAP line number
4622-4623	$120E-$120F	Pointer to start of statement where last error occurred
4624-4625	$1210-$1211	End-of-program pointer
4626-4627	$1212-$1213	Top-of-BASIC pointer
4628-4629	$1214-$1215	Temporary text pointer storage for DO
4630-4631	$1216-$1217	Temporary line number storage for DO
4632-4634	$1218-$121A	USR function jump vector
4635-4639	$121B-$121F	Seed value for random number generation
4640	$1220	Degrees between segments for CIRCLE routine
4641	$1221	Unused
4642	$1222	Tempo setting for PLAY statement
4643-4648	S1223-$1228	Durations for currently active notes
4649-4650	$1229-$122A	Duration of current note
4651	S122B	Octave for current note
4652	$122C	Sharp/flat flag
4653-4654	$122D-$122E	Frequency for current note
4655	$122F	Voice number for current note
4656-4658	$1230-$1232	Waveforms for current notes
4659	$1233	Dotted note flag
4660-4661	$1234-$1235	Current filter cutoff frequency
4662	$1236	Current resonance setting
4663	$1237	Current filter type
4664	$1238	Filter type index
4665	$1239	Temporary storage
4666	$123A	Current instrument number
4667-4669	$123B-$123D	Envelope parameters for current instrument
4670	$123E	Index into instrument table for current instrument
4671-4720	$123F-$1270	Instrument parameter tables
4721-4722	$1271-$1272	Current filter cutoff frequency
4723	$1273	Current filter control and resonance setting
4724	$1274	Current filter type selection
4725	$1275	Current SID chip volume setting
4726-4728	$1276-$!278	Collision flags

4729-4734	$1279-$127D	Target line numbers for COLLISION
4735	$127F	Collision enable flag
4736	$1280	Collision type index
4737	$1281	Voice for current SOUND statement
4738-4770	S1282-S12A2	Table of SOUND statement settings
4771-4776	$12A3-$12A8	Parameters for most recent SOUND statement
4785	$12B1	Temporary storage for POT and PEN routines
4786	$12B2	Temporary storage for POT routine
4787-4790	$12B3-$12B6	Temporary parameter storage for WINDOW statement
4791-4806	$12B7-$12C6	Filename buffer for DOS support commands
4791-4854	$12B7~$12F6	Sprite pattern storage
4854-4857	$12F6-$12F9	Sprite pattern suffix
4858	S12FA	Sprite mode indicator for SPRDEF
4859	$12FB	Sprite pattern line count for SPRDEF
4860	$12FC	Sprite number for SPRDEF
4861	$12PD	BASIC IRQ activity flag
4862-4863	$12FE-$12FF	Unused

Chaptei 5: BASIC

16384	$4000	BASIC cold-start entry point
16387	$4003	BASIC warm-start entry point
16390	$4006	BASIC IRQ entry point
16393	$4009	Performs a warm start of BASIC
16416	$4020	Unused
16419	$4023	Performs a cold start of BASIC
16453	$4045	Initializes BASIC pointers and constants
16658	$4112	Initializes SID registers and sound routine locations
16762	$417A	Initializes MMU preconfiguration registers
16781	$418D	Initializes sprite speed and direction table
16795	$419B	Displays the power-on message
16827	$41BB	Text for power-on message
16977	$4251	Initializes BASIC indirect vectors
16999	$4267	Table of default vector values
17017	$4279	Text for character retrieval routines
17102	$42CE	Assorted character retrieval subroutines
17162	$430A	Tokenizes keywords in lines of BASIC program text
17328	S43B0	Handles extended tokens
17356	$43CC	Deletes a character in the input buffer
17378	$43E2	Searches keyword tables for match
17431	$4417	BASIC keyword tables
17929	$4609	Table of extended token statements
18121	$46C9	Table of extended token functions
18172	$46FC	Table of statement dispatch addresses
18242	$4742	Table of statement dispatch addresses
18172	$46FC	Table of statement dispatch addresses
18317	$478D	Table of function dispatch addresses
18454	$4816	Table of function dispatch addresses
18472	$4828	Table of operator priorities and dispatch addresses
18502	$4846	Prints unimplemented command message
18507	$484B	Table of BASIC error messages
19074	$4A82	Sets pointer to error message
19103	$4A9F	Main BASIC statement execution routine
19190	$4AF6	Executes the next BA5IC statement
19381	S4BB5	Tests for RUN/STOP keypress

19403	$4BCB	Handles the STOP and END statements
19447	$4BF7	Handles the execution of function keywords
19587	$4C83	Displays the SYNTAX ERROR message
19590	$4C86	Handles the OR logical operator
19593	$4C89	Handles the AND logical operator
19638	$4CB6	Handles relational operators (<, —, >)
19754	$4D2A	Prints the READY prompt
19767	$4D37	Enters MAIN with a READY prompt
19770	$4D3A	Displays an OUT OF MEMORY error message
19772	$4D3C	Handles BASIC errors
19836	$4D7C	Prints a specified error message
19895	$4DB7	Handles immediate mode and program line entry
19938	$4DE2	Adds or deletes BASIC program lines
20303	$4F4F	Relinks BASIC program lines
20371	S4F93	Reads a line of input into the buffer
20394	$4FAA	Searches for a particular token in the runtime stack
20478	$4FFE	Decrements the runtime stack pointer
20503	$5017	Checks for available string space
20569	$5059	Increments runtime stack pointer
20580	$5064	Searches program text for a specified line number
20640	$50AO	Creates integer value from a character string
20706	$50E2	Handles the LIST statement
20771	$5123	Lists a single BASIC program line
20950	$51D6	Handles the NEW statement
20984	$51F8	Handles the CLR statement
21076	$5254	Resets the CHRGET text pointer
21090	$5262	Handles the RETURN statement
21135	$528F	Handles the BEND and DATA statements
21149	$529D	Handles the REM statement
21189	$52C5	Handles the IF statement
21280	$5320	Skips a BEGIN-BEND block
21393	$5391	Handles the ELSE statement
21411	$53A3	Handles the ON statement
21446	$53C6	Handles variable value assignments
21818	$553A	Handles the PRINT* statement
21824	$5540	Handles the CMD statement
21844	$5554	Handles the PRINT statement
22034	$5612	Handles the GET statement (also GET# and GETKEY)
22088	$5648	Handles the ENPUT# statement
22114	$5662	Handles the INPUT statement
22185	$56A9	Handles the READ statement
22474	$57CA	Moves the CHRGET text pointer to the next DATA statement
22516	$57F4	Handles the NEXT statement
22648	$5878	Handles the DIM statement
22661	$5885	Handles the SYS statement
22708	$58B4	Handles the TRON and TROFF statements
22717	$58BD	Handles the RREG statement
22785	$5901	Handles MID$ when used as a statement
22901	$5975	Handles the AUTO statement
22918	$5986	Handles the HELP statement
22956	$59AC	Highlights the portion of a listed line containing an error
2299]	$59CF	Handles the GOSUB statement
23003	$59DB	Handles the GOTO statement

23069	$5A1D	Places RETURN parameters in the runtime stack
23101	S5A3D	Handles the GO statement
23136	$5A60	Handles the CONT statement
23169	$5A81	Sets flags for running a program
23195	$5A9B	Handles the RUN statement
23242	$5ACA	Handles the RESTORE statement
23280	$5AF0	Table of tokens for RENUMBER
23288	$5AF8	Handles the RENUMBER statement
24057	$5DF9	Handles the FOR statement
24199	S5ES7	Handles the DELETE statement
24372	$5F34	Handles the PUDEF statement
24397	$5F4D	Handles the TRAP statement
24418	$5F62	Handles the RESUME statement
24544	$5FE0	Handles the DO statement
24633	$6039	Handles the EXIT statement
24714	$608A	Handles the LOOP statement
24801	$60E1	Assigns a definition string to a programmable key
24842	$610A	Handles the KEY statement
24989	$619D	Table of characters for KEY
25000	$61A8	Handles the PAINT statement
25271	$62B7	Handles the BOX statement
25643	$642B	Handles the SSHAPE statement
25997	$65 8D	Handles the GSHAPE statement
26254	$668E	Handles the CIRCLE statement
26443	$6750	Bitmapped graphics circle drawing subroutine
26519	$6797	Handles the DRAW statement
26583	$67D7	Handles the CHAE statement
26965	$6955	Handles the LOCATE statement
26976	$6960	Handles the SCALE statement
27096	$69D8	Table of scaling factors
27106	$69E2	Handles the COLOR statement
27212	$6A4C	Table for translating VIC color values to VDC color values
27228	$6A5C	Calculates color fill values
27257	$6A79	Handles the SCNCLR statement
27482	$6B5A	Handles the GRAPHIC statement
27593	$6BC9	Handles the BANK statement
27607	$6BD7	Handles the SLEEP statement
27693	$6C2D	Handles the WAIT statement
27727	$6C4F	Handles the SPRITE statement
27846	$6CC6	Handles the MOVSPR statement
28129	$6DE1	Handles the PLAY statement
28631	$6FD7	Handles the TEMPO statement
28644	$6FE4	Data tables for PLAY string processing
28689	$7011	Default values for ENVELOPE instrument tables
28742	$7046	Handles the FILTER statement
28865	$70Cl	Handles the ENVELOPE statement
29028	$7164	Handles the COLLISION statement
29072	$7190	Handles the SPRCOLOR statement
29110	$71B6	Handles the WIDTH statement
29125	$71C5	Handles the VOL statement
29164	$71EC	Handles the SOUND statement
29388	$72CC	Handles the WINDOW statement
29493	$7335	Handles the BOOT statement
29554	$7372	Handles the SPRDEF statement
30444	$76EC	Handles the SPRSAV statement

30643	$77B3	Handles the FAST statement
30660	$77C4	Handles the SLOW statement
30679	$77D7	Evaluates an expression with a test for type mismatch
30703	$77EF	Evaluates an expression
30935	$78D7	Evaluates a single term of a numeric expression
31084	$796C	Displays a SYNTAX ERROR message
31096	$7978	Evaluates a variable value
31407	$7AAF	Finds or creates a variable
31632	$7B90	Creates an entry in the variable table for a new scalar variable
31846	$7C66	Moves arrays upward in bank 1 to make room for a new scalar variable
31915	$7CAB	Finds or creates an array variable
32768	$8000	Handles the FRE function
32800	$8020	Prints designers message
32842	$804A	Handles the VAL function
32886	$8076	Handles the DEC function
32965	$80C5	Handles the PEEK function
32997	$80E5	Handles the POKE statement
33014	$80F6	Handles the ERR$ function
33090	$8142	Handles the HEX$ function
33154	$8182	Handles the RGR function
33179	$819B	Handles the RCLR function
332S3	$8203	Handles the JOY function
33357	$824D	Handles the POT function
33454	$82AE	Handles the PEN function
33530	$82FA	Handles the POINTER function
33566	$831E	Handles the RSPR1TE function
33633	$8361	Handles the RSPCOLOR function
33660	$837C	Handles the BUMP function
33687	$8397	Handles the RSPPOS function
33761	$83E1	Handles the XOR function
33799	$8407	Handles the RW1ND0W function
33844	$8434	Handles the RND function
33936	$8490	Table of floating point constants for RND calculation
34000	$84D0	Handles the POS function
34009	$84D9	Checks that BASIC is in run mode
34032	$84F0	Checks that BASIC is in immediate mode
34042	$84FA	Handles the DEF statement
34107	$853B	Handles user-defined functions using FN
34222	$85AE	Handles the STR$ fucntion
34239	$85BF	Handles the CHR$ function
34262	$85D6	Handles the LEFT$ function
34314	$860A	Handles the RIGHT$ function
34332	$861C	Handles the MID$ function
34408	$8668	Handles the LEN function
34423	$8677	Handles the ASC function
34437	$8685	Displays the ILLEGAL QUANTITY error message
34440	$8688	Creates space for a string in the string pool
34458	$869A	Stores a string in the string pool
34573	$870D	Performs string concatenation
34683	$877B	Evaluates a string parameter
34801	$87F1	Evaluates a numeric expression
34819	$8803	Evaluates parameters for POKE or WAIT
34831	$880F	Checks that the next character is a comma
34837	$8815	Evaluates a numeric parameter

34862	$882E	Subtracts value in memory from FAC1
34865	$8831	Subtracts FAC1 from FAC2
34885	$8845	Adds value in memory to FAC1
34888	$8848	Adds FAC1 to FAC2
34993	$88B1	Normalizes FAC1
35110	$8926	Forms twos complement of FAC1
35165	$895D	Displays OVERFLOW error message
35170	$8962	Performs byte alignment of FAC1
35274	$89CA	Handles the LOG function
35342	$8A0E	Adds 0.5 to FAC1
35364	$8A24	Multiplies value in memory by FAC1
35367	$8A27	Multiplies value in memory by FAC1
35465	$8A89	Loads FAC2 with value from the current bank
35508	$8AB4	Loads FAC2 with value from bank 1
35607	$8B17	Multiplies FAC1 by 10
35640	$8B38	Divides FAC1 by 10
35657	$8B49	Divides value in memory by FAC1
35660	$8B4C	Divides FAC2 by FAC1
35796	$8BD4	Loads FAC1 from memory
35840	$8C00	Copies FAC1 value into memory
35880	$8C28	Copies FAC2 into FAC1
35896	$8C38	Copies FAC1 into FAC2
35911	$8C47	Rounds FAC1
35927	$8C57	Determines the sign of the value in FAC1
35941	$8C65	Handles the SGN function
35972	$8C84	Handles the AB5 function
35975	$8C87	Compares FAC1 against FAC2
36039	$8CC7	Converts FAC1 to a four-byte integer
36091	$8CFB	Handles the INT function
36120	$8D18	Fills FAC1 with the value in the accumulator
36130	$8D22	Generates floating point value representing character string
36390	$8E26	Prints IN and a line number
36398	$8E2E	Prints a line number
36418	$8E42	Generates a character string representing the value in FAC1
36791	$8FB7	Handles the SQR function
36801	$8FC1	Handles the exponentiation (T) operator
36869	$9005	Table of floating point constants for EXP evaluation
36915	$9033	Handles the EXP function
36998	$9086	Performs series evaluation
370B0	$90D8	Calls the Kernal OPEN routine
37087	$90DF	Calls the Kernal BSOUT routine
37093	$90E5	Calls the Kernal BASIN routine
37117	$90FD	Calls the Kemal CHKIN routine
37129	$9109	Calls the Kemal GETIN routine
37138	$9112	Handles the SAVE statement
37161	$9129	Handles the VERIFY statement
37164	$912C	Handles the LOAD statement
37261	$918D	Handles the OPEN statement
37274	$919A	Handles the CLOSE statement
37294	$91AE	Evaluates parameters for SAVE, LOAD, and VERIFY
37366	$91F6	Evaluates parameters for OPEN and CLOSE
37433	$9239	Clears DS$ after disk operations
37457	$9251	BASIC calls to Kemal routines
37457	$9251	BASIC call to Kernal's READSS routine

37463	$9257	BASIC call to Kemal's 5ETLFS routine
37469	$925D	BASIC call to Kernal's SETNAM routine
37475	$9263	BASIC call to Kemal's BASIN routine
37481	$9269	BASIC call to Kemal's BSOUT routine
37487	$926F	3ASIC call to Kemal's CLRCH routine
37493	$9275	BASIC call to Kemal's CLOSE routine
37499	$927B	BASIC call to Kemal's CLALL routine
37505	$9281	BASIC call to Kemal's PRIMM routine
37511	$9287	BASIC call to Kemal's SETBANK routine
37517	$928D	BASIC call to Kemal's PLOT routine
37523	$9293	BASIC call to Kemal's STOP routine
37529	$9299	Creates space in the string pool for a temporary string
37610	$92EA	Performs garbage collection on string pool
37897	$9409	Handles the COS function
37904	$9410	Handles the SIN function
37977	$9459	Handles the TAN function
38021	$9485	Table of constants for trig function evaluation
38067	$94B3	Handles the ATN function
38115	$94E3	Table of constants for trig function evaluation
38176	$9520	Handles the PRINT USING statement
39361	$99C1	Handles the INSTR function
39692	$9B0C	Handles the RDOT function
39728	$9B30	Bitmapped graphics line drawing routine
39931	$9BFB	Bitmapped point plotting routine
40010	$9C4A	Scales graphics parameter
40366	$9DAE	Applies scaling factor to a specified parameter
40557	$9E6D	Evaluates graphics parameters
40712	$9F08	Handles relative graphics parameters
40783	$9F4F	Allocates the bitmapped graphics area
40903	$9FC7	Adjusts BASIC program pointers for graphics area allocation or de-allocation
40994	$A022	De-allocates the bitmapped graphics area
41076	$A074	Confirms that the graphics area has been allocated
41086	$A07E	Handles the CATALOG and DIRECTORY statements
41245	$A11D	Handles the DOPEN statement
41268	$A134	Handles the APPEND statement
41303	$A157	Finds an available secondary address
43327	$A16F	Handles the DCLOSE statement
41347	$A183	Closes all open files for a specified device
41356	$A18C	Handles the DSAVE statement
41380	$A1A4	Handles the DVERIFY statement
41383	$A1A7	Handles the DLOAD statement
41416	$A1C8	Handles the BSAVE statement
41496	$A218	Handles the BLOAD statement
41575	$A267	Handles the HEADER statement
41633	$A2A1	Handles the SCRATCH statement
41687	$A2D7	Handles the RECORD statement
41762	$A322	Handles the DCLEAR statement
41775	SA32F	Handles the COLLECT statement
41798	$A346	Handles the COPY statement
41826	$A362	Handles the CONCAT statement
41838	$A36E	Handles the RENAME statement
41852	$A37C	Handles the BACKUP statement
41923	$A3C3	Evaluates parameters for disk commands
42535	$A627	Table of disk command templates
42599	$A667	Sets up disk command buffer

42872	$A778	Reads disk status string (DS$)
42977	$A7E1	Provides AKE YOU SURF, query
43021	$A80D	Clears disk status string
43077	$A845	Switches to bank 15 configuration
43085	$A84D	BASIC IRQ service routine
43504	$A9F0	Common exit point from BASIC IRQ routine
43551	$AA1F	Handles the STASH statement
43556	$AA24	Handles the FETCH statement
43561	$AA29	Handles the SWAP statement
43630-44642	$AA6E-$AE62	Unused
44643-44799	$AE63-$AEFF	Encoded message from the designers of the 128
44800	$AF00	Entry point for the AYINT routine
44803	$AF03	Entry point for the GIVAYF routine
44806	$AF06	Entry point for the FOUT routine
44809	$AF09	Entry point for the VA1_1 routine
44812	$AF0C	Entry point for the GETADR routine
44815	$AF0F	Entry point for the FLOATC routine
44818	$AF12	Entry point for the FSUB routine
44821	$AF15	Entry point for the FSUBT routine
44824	$AF18	Entry point for the FADD routine
44827	$AF1B	Entry point for the FADDT routine
44830	$AF1E	Entry point for the FMULT routine
44833	$AF21	Entry point for the FMULTT routine
44836	$AF24	Entry point for the FDIV routine
44839	$AF27	Entry point for the FDIVT routine
44842	$AF2A	Entry point for the LOG routine
44845	$AF2D	Entry point for the INT routine
44848	$AF30	Entry point for the SQR routine
44851	$AF33	Entry point for the NEGOP routine
44854	$AF36	Entry point for the FPVVR routine
44857	$AF39	Entry point for the FPWRT routine
44860	$AF3C	Entry point for the EXP routine
44863	$AF3F	Entry point for the COS routine
44866	$AF42	Entry point for the SIN routine
44869	$AF45	Entry point for the TAN routine
44872	$AF48	Entry point for the ATN routine
44875	$AF4B	Entry point for the ROUND routine
44878	$AF4E	Entry point for the ABS routine
44881	$AF51	Entry point for the SIGN routine
44884	$AF54	Entry point for the FCOMP routine
44887	$AF57	Entry point for the RND_0 routine
44890	$AF5A	Entry point for the CONUPK routine
44893	JAF5D	Entry point for the ROMUPK routine
44896	$AF60	Entry point for the MOVFRM routine
44899	$AF63	Entry point for the MOVFM routine
44902	$AF66	Entry point for the MOVMF routine
44905	$AF69	Entry point for the MOVFA routine
44908	$AF6C	Entry point for the MOVAF routine
44914	$AF72	Entry point for the DRAWLN rotuine
44917	SAF75	Entry point for the GPLOT routine
44920	$AF78	Entry point for the CIRSUB routine
44923	$AF7B	Entry point for the RUN routine
44926	$AF7E	Entry point for the RUNC routine
44929	$AF81	Entry point for the CLR routine
44932	$AF84	Entry point for the NEW routine
44935	$AF87	Entry point for the LNKFRG routine

44938	SAFSA	Entry point for the CRUNCH routine
44941	$AF8D	Entry point for the FNDLN routine
44944	$AP90	Entry point for the NEWSTT routine
44947	$AF93	Entry point for the EVAL routine
44950	$AF96	Entry point for the FRMEVL routine
44953	$AF99	Entry poinl for the RUN routine
44956	$AF9C	Entry point for the SETEXC routine
44959	$AF9F	Entry point for the LINGET routine
44962	$AFAZ	Entry point for the GARBA2 routine
44965	$AFA5	Entry point for the MAIN routine

Chapter 6: Machine Language Monitor ROM

45056	$B0OO	Monitor cold start entry point
45059	$B003	Monitor break entry point
45062	$B006	Reentry point from the IMON indirect vector
45065	$B009	Monitor entry routine when BRK instruction encountered
45089	$B021	Cold start routine for monitor
45136	$B050	Handles R (register display) command
45195	$B08B	Main command execution loop for the monitor
45285	$B0E3	Handles X (exit to BASIC) command
45288	$B0E6	Table of monitor commands
45308	SB0FC	Table of execution addresses for the monitor commands
45338	$B11A	INDFET call for the monitor
45354	$B12A	INDSTA call for the monitor
45373	SB13D	INDCMP call for the monitor
45394	$B152	Handles M (memory display) command
45460	$B194	Handles ; (change register) command
45483	$B1AB	Handles > (change memory) command
45526	$B1D6	Handles G (go to routine) command
45535	$B1DF	Handles] (jump to subroutine) command
45544	$B1E8	Displays a line of memory as hex bytes and ASCII characters
45617	$B23I	Compares or transfers blocks of memory
45774	$B2CE	Searches memory for byte pattern
45879	$B337	Prepares for save, load, or verify
45983	$B39F	Handles save for monitor
45995	$B3AB	Handles load and verify
46033	$B3D1	Prepares for relocating load or verify
46043	$B3DB	Fills memory with specified byte value
46086	$B406	Handles A (assemble) command, or its equivalent
46489	$B599	Handles D (disassemble) command
46548	SB5D4	Disassembles a single instruction
46681	$B659	Calculates mnemonic and addressing mode
46753	$B6A1	Prints mnemonic for opcode
46787	$B6C3	Opcode decoding table
46855	$B707	Table of addressing mode indicators
46869	$B715	Table of mode identification characters
468S1	$B721	Table of mnemonics in packed form
47013	$B7A5	Evaluates a parameter in the input buffer
47054	SB7CE	Transforms numeric parameter into byte value
47242	$BS8A	Table of bases and bits-per-digit
47250	$B892	Prints a hexadecimal value
47277	$B8AD	Moves cursor to start of current line

47284	$B8B4	Moves cursor to start of next line
47289	$B8B9	Clears a screen line
47298	SB8C2	Prints two ASCII characters for a byte value
47314	JB8D2	Converts a byte value into two ASCII characters
47335	$B8E7	Tests next character in the input buffer
47361	$B901	Transfers address and bank values to working pointer
47374	$B90E	Calculates number of bytes and banks to display or move
47394	$B922	Decrements address or line count
47420	$B93C	Decrements byte count
47440	$B950	Increments address pointer
47456	SB960	Decrements address pointer
47476	$B974	Changes bank and address
47491	$B983	Prepares pointers for dual-address operations
47537	SB9B1	Performs number base conversion
47623	$BA07	Converts a hexadecimal value to decimal
47687	$BA47	Prints octal, binary, or decimal values
47760	$BA90	Handles @ (disk) commands
47875	$BB03	Displays disk directory
47986-49151	$BB72-$BFFF	Unused

Chapter 7: Screei Editor ROM

49152	**$C000**	Screen editor jump table
49263	$C06F	Table of default keyboard decoding table pointers
49275	$C07B	Initializes screen editor constants, variables, tables, and vectors
49474	$C142	Clears the current window and homes the cursor
49488	$C150	Moves the cursor to the home position of the current window
49500	$C15C	Sets starting address pointers for the current line
49502	$C15E	Sets starting address pointers for a specified line
49556	$C194	Performs screen and keyboard portion of IRQ functions
49716	$C234	Performs GETIN from keyboard
49752	$C258	Accepts a line of keyboard input and returns the first character
49819	$C29B	Performs BASIN from screen or keyboard
49919	$C2FF	Handles quote mode flag
49932	$C30C	Provides common exit for screen BSOUT subroutines
49952	$C320	Handles character printing for screen BSOUT
49982	SC33E	Updates the cursor position
50019	$C363	Moves the cursor down one line
50044	$C37C	Inserts a new line linked to the one above
50086	$C3A6	Scrolls the window up one line
50140	$C3DC	Copies lines up one row and clears bottom line
50189	$C40D	Copies a line
50341	$C4A5	Clears a line
50492	$C53C	Fills or copies a block of 8563 RAM
50525	$C55D	Scans keyboard matrix for keypress
50769	SC651	Decodes key matrix value into character value and handles key repeating
50909	$C6DD	Table of programmable key character values
50919	$C6E7	Handles cursor blinking for 40-column screen
50989	$C72D	Handles BSOUT to the screen
51055	$C76F	Handles RETURN and SHIFT-RETURN characters, CHR$(13) and CHR$(141)

51069	SC77D	Cancel quote arid reverse modes and dear pending inserts (ESC O and ESC ESC)
51084	$C78C	Tables of screen control codes and dispatch addresses
51126	$C7B6	Interprets character codes less than 32
51162	$C7DA	Handles color change characters
51190	$C7F6	Calls control code execution routines
51202	$C802	Interprets character codes greater than 127
51220	$C814	Handles character codes 128-159
51284	$C854	Handles cursor right character, CHR$(29)
51290	$C85A	Handles cursor down character, CHR$(17)
51293	$C85D	Checks whether cursor moved onto a new logical line
51303	$C867	Handles cursor up character, CHR$(145)
51317	$C875	Handles cursor left character, CHR$(157)
51328	$C880	Handles switch-to-lowercase character, CHR$(14)
51346	$C892	Handles switch-to-uppercase character, CHR${142)
51366	$C8A6	Handles case switching disable character, CHR$(11)
51372	$C8AC	Handles case switching enable character, CHR$(12)
51379	$C8B3	Handles cursor home character, CHR$(19)
51391	$C8BF	Handles reverse off character, CHE${146)
51394	$C8C2	Handles reverse on character, CHR$(18)
51399	$C8C7	Handles underline on character, CHR$(2)
51406	$C8CE	Handles underline off character, CHR$(130)
51413	$C8D5	Handles flash on character, CHR$(15)
51420	$C8DC	Handles flash off character, CHR$(143)
51427	$C8E3	Handles insert character, CHR$(148)
51483	$C91B	Handles delete character, CHR$(20}
51506	$C932	Restores the cursor row and column positions
51517	SC93D	Deletes a character in a logical line
51535	$C94F	Handles tab character, CHR$(9)
51553	$C961	Handles clear/set tab stop character, CHR$(24)
51564	$C96C	Tests tab stop bit for current cursor position
51584	$C980	Clears all tab stops (ESC Z)
51587	$C983	Sets default tab stops (ESC Y)
51598	$C98E	Handles bell character, CHR$(7)
51633	$C9B1	Handles line feed character, CHR$(10)
51646	$C9BE	Handles ESC sequences
51678	$C9DE	Table of ESC key dispatch addresses
51732	$CA14	Defines the upper left corner of the window (ESC T)
51734	$CA16	Defines the lower right comer of the window (ESC B)
51739	$CA1B	Sets window boundaries
51748	$CA24	Resets the window to full screen size
51773	$CA3D	Inserts a blank line (ESC I)
51794	$CA52	Deletes the current logical line (ESC D)
51830	$CA76	Erases to the end of the current logical line (ESC Q)
51851	$CA8B	Erases to the start of the current logical line (ESC P)
51871	$CA9F	Erases to the end of the window (ESC (a)}
51900	$CABC	Scrolls the display up one line (ESC V)
51914	$CACA	Scrolls the display down one line (ESC W)
51938	$CAE2	Enables screen scrolling (ESC L)
51941	$CAE5	Disables screen scrolling (ESC M)
51946	$CAEA	Cancels autoinsert mode (ESC C)
51949	$CAED	Enables autoinsert mode (ESC A)
51954	$CAF2	Changes 80-column cursor to solid block (ESC S)
51966	$CAFE	Changes 80-column cursor to underline (ESC U)
51979	$CB0B	Disables cursor blinking (ESC E)
52001	SCB21	Enables cursor blinking (ESC F)

52023	$CB37	Enables tone for bell character (ESC G)
52026	$CB3A	Disables tone for bell character (ESC H)
52031	$CB3F	Switches 80-column screen to reverse mode (ESC R}
52040	SCB48	Switches 80-column screen to normal mode (ESC N)
52050	$CB52	Moves the cursor past the last character on the current logical line (ESC K)
52056	$CB58	Reads character and attribute at current cursor position
52084	$CB74	Tests whether a line is linked
52097	$CB81	Links or unlinks the current screen line
52101	$CB85	Unlinks a screen line
52115	$CB93	Links a screen line
52127	$CB9F	Calculates offsets into the line link bitmap
52145	$CBB1	Moves the cursor to the start of logical line (ESC J)
52163	$CBC3	Finds the position of the last character in a line
52205	SCBED	Moves the cursor one position to the right
52224	$CC00	Moves the cursor one position to the left
52254	$CC1E	Stores the cursor position for later restoration
52263	$CC27	Prints a space
52271	$CC2F	Displays a character using the current attribute
52274	$CC32	Displays a character using the previous attribute
52276	SCC34	Displays a character at the current cursor position
52315	$CC5B	Returns height and width of current screen window
52330	$CC6A	Reads or sets the current cursor position
52386	$CCA2	Defines a programmable function key
52512	SCD20	Calculates the offset to the start of key definition string
52524	$CD2C	Changes screen displays (ESC X)
52526	$CD2E	Switches active screen displays
52567	$CD57	Sets cursor position on 80-column screen
52591	$CD6F	Turns cursor on
52639	$CD9F	Turns cursor off
52682	$CDCA	Writes a byte value to SO-column chip memory
52684	$CDCC	Writes to an 80-column chip register
52696	$CDD8	Reads a byte value from 80-column chip memory
52698	$CDDA	Reads from an 80-column chip register
52710	$CDE6	Sets the current address in 80-column screen memory
52729	$CDF9	Sets the current address in 80-column attribute memory
52748	$CE0	Initializes character definitions for BO-column screen
52812	$CE4C	Table of color character translation values
52828	SCE5C	Table of 8563 color code translation values
52844	SCE6C	Table of bit mask values
52852	SCE74	Tables of default screen editor variables
52904	$CEA8	Table of standard function key definitions
52981-53247	$CEF5-$CFFF	Unused

Chapter 8: Hardware Chip Registers, Color RAM, and Character ROM

VIC (40-Column) Video Chip Registers

53248	$D000	Sprite 0 horizontal position register
53249	$D001	Sprite 0 vertical position register
53250	SD002	Sprite 1 horizontal position register
53251	$D003	Sprite 1 vertical position register

53252	$D004	Sprite 2 horizontal position register
53253	$D005	Sprite 2 vertical position register
53254	$D006	Sprite 3 horizontal position register
53255	$D007	Sprite 3 vertical position register
53256	$D008	Sprite 4 horizontal position register
53257	$D009	Sprite 4 vertical position register
53258	$D00A	Sprite 5 horizontal position register
53259	$D00B	Sprite 5 vertical position register
53260	$D00C	Sprite 6 horizontal position register
53261	$D00D	Sprite 6 vertical position register
53262	$D00E	Sprite 7 horizontal position register
53263	$D00F	Sprite 7 vertical position register
53264	$D010	Sprites 0-7 horizontal position (most significant bits)
53265	SD011	Control/vertical fine scrolling register
53266	$D012	Raster scan line register
53267	$D013	Light pen horizontal position
53268	$D014	light pen vertical position
53269	$D015	Sprite enable register
53270	$D016	Control /horizontal fine scrolling register
53271	$D017	Sprite vertical expansion register
53272	$D018	Memory control register
53273	$D019	Interrupt flag register
53274	$D01A	Interrupt mask register
53275	$D018	Sprite-to-foreground priority register
53276	$D01C	Sprite multicolor mode register
53277	$D01D	Sprite horizontal expansion register
53278	SD01E	Sprite-to-sprite collision register
53279	$D01F	Sprite-to-foreground collision register
53280	$D020	Border color register
53281	$D021	Background color (source 0) register
532S2	$D022	Background color source 1 register
53283	$D023	Background color source 2 register
53284	$D024	Background color source 3 register
53285	$D025	Sprite multicolor source 0 register
53286	$D026	Sprite multicolor source 1 register
53287	$D027	Sprite 0 color register
53288	$D028	Sprite 1 color register
53289	$D029	Sprite 2 color register
53290	$D02A	Sprite 3 color register
53291	$D02B	Sprite 4 color register
53292	SD02C	Sprite 5 color register
53293	$D02D	Sprite 6 color register
53294	$D02E	Sprite 7 color register
53295	SD02F	Extended keyboard scan register
53296	$D030	Processor dock rate control register

SID (Sound Interface Device) Chip Registers

54272	$D400	Frequency register for voice 1 (low byte)
54273	$D401	Frequency register for voice 1 (high byte)
54274	$D402	Pulsewidth for voice 1 (low byte)
54275	$D403	Pulse width for voice 1 (high byte)
54276	SD404	Control register for voice 1
54277	$D405	Attack/decay register for voice 1
54278	$D406	Sustain/release register for voice 1
54279	$D407	Frequency register for voice 2 (low byte)
54280	$D408	Frequency register for voice 2 (high byte)

54281	$D409	Pulsewidth for voice 2 (low byte)
54282	$D40A	Pulsewidth for voice 2 (high byte)
54283	JD40B	Control register for voice 2
54284	$D40C	Attack/decay register for voice 2
54285	$D40D	Sustain/release register for voice 2
54286	$D40E	Frequency register for voice 3 (low byte)
54287	$D40F	Frequency register for voice 3 (high byte)
54288	$D410	Pulsewidth for voice 3 (low byte)
54289	$D411	Fulsewidth for voice 3 (high byte)
54290	$D412	Control register for voice 3
54291	$D413	Attack/decay register for voice 3
54292	$D414	Sustain/release register for voice 3
54293	$D415	Filter cutoff frequency (low byte)
54294	$D416	Filter cutoff frequency (high byte)
54295	$D417	Resonance/filter control register
54296	$D418	Volume/filter mode register
54297	$D419	Potentiometer 0 reading
54298	$D41A	Potentiometer 1 reading
54299	$D41B	Voice 3 oscillator output
54300	$D41C	Voice 3 envelope generator output

MMU (Memory Management Unit) Chip Registers

54528	$D500	Configuration register
54529	$D501	Preconfiguration register A
54530	$D502	Preconfiguration register B
54531	$D503	Preconfiguration register C
54532	$D504	Preconfiguration register D
54533	$D505	Mode configuration register
54534	$D506	RAM configuration register
54535	$D507	Page 0 page pointer
54536	$D508	Page 0 block pointer
54537	$D509	Page 1 page pointer
54538	$D50A	Page 1 block pointer
54539	$D50B	MMU version register

Chip External Registers

VDC (80-column) Video

54784	$D600	VDC address/status register
54785	$D601	VDC data register

VDC Chip Internal Registers

		Total number of horizontal character positions
		Number of visible horizontal character positions
2	$02	Horizontal sync position
3	$03	Horizontal and vertical sync width
i	$04	Total number of screen rows
5	$05	Vertical fine adjustment
6	$06	Number of visible screen rows
7	$07	Vertical sync position
8	$08	Interlace mode control register
9	$09	Number of scan lines per character
10	$0A	Cursor mode control
11	SOB	Ending scan line for cursor
12	$0C	Screen memory starting address (high byte)
13	SOD	Screen memory starting address (low byte)
14	$0E	Cursor position address (high byte)
15	$0F	Cursor position address (low byte)
16	$10	Light pen vertical position
17	$11	Light pen horizontal position
18	$12	Current memory address (high byte)

19	$13	Current memory address {low byte)
20	$14	Attribute memory starting address (high byte)
21	$15	Attribute memory starting address (low byte)
22	$16	Character horizontal size control register
23	$17	Character vertical size control register
24	$18	Vertical smooth scrolling and control register
25	$19	Horizontal smooth scrolling and control register
26	$1A	Foreground/background color register
27	$1B	Address increment per row
2$	$1C	Character set address and memory type register
29	$1D	Underline scan line position register
30	$1E	Number of bytes for block write or copy
31	$1F	Memory read/write register
32	$20	Block copy source address (high byte)
33	$21	Block copy source address (low byte)
34	$22	Beginning position for horizontal blanking
35	$23	Ending position for horizontal blanking
36	S24	Number of memory refresh cycles per scan line
55296	$D800	VIC color memory (two blocks)

CIA (Complex Interface Adapter) Chip #1

56320	$DC00	Port A data I/O register
56321	$DC01	Port B data I/O register
56322	SDC02	Port A data direction register
56323	SDC03	Port B data direction register
56324	$DC04	Timer A latch/counter (low byte)
56325	$DC05	Timer A latch/counter (high byte)
56326	$DC06	Timer B latch/counter (low byte)
56327	$DC07	Timer B latch/counter (high byte)
56328	$DC08	Time-of-day dock (1/10 seconds)
56329	$DC09	Time-of-day clock (seconds)
56330	$DC0A	Time-of-day Hock (minutes)
56331	$DC0B	Time-of-day clock (hours)
56332	$DC0C	Serial data register
56333	SDCOD	Interrupt control register
56334	$DC0E	Control register A
56335	$DC0F	Control register B

CIA (Complex Interface Ada]

56576	$DD00	Port A data I/O register
56577	$DD01	Port B data I/O register
56578	SDDO2	Port A data direction register
56579	$DD03	Fort B data direction register
56580	$DD04	Timer A latch/counter (low byte)
56581	$DD05	Timer A latch/counter (high byte)
56582	$DD06	Timer B latch/counter (low byte)
56583	$DD07	Timer B latch/counter (high byte)
56584	$DD08	Time-of-day clock (1/10 seconds)
56585	$DD09	Time-of-day clock (seconds)
56586	$DD0A	Time-of-day clock (minutes)
56587	$DD0B	Time-of-day clock (hours)
56588	$DD0C	Serial data register
56589	$DD0D	Interrupt control register
56590	$DD0E	Control register A
56591	$DD0F	Control register B
56832-57087	$DE00-$DEFF	I/O Expansion Slot #1
57088-57343	$DF00-$DFFF	I/O Expansion Slot #2

KEC (**RAM** Expansion Controller) Chip Registers

57088	SDF00	Status register
57089	$DF01	Command register
57090	$DF02	System RAM base address (low byte)
57091	$DF03	System RAM base address (high byte)
57092	$DF04	Expansion RAM base address (low byte)
57093	$DF05	Expansion RAM base address (high byte)
57094	$DF06	Expansion RAM bank
57095	$DF07	Count of bytes to transfer (low byte)
57096	$DF08	Count of bytes to transfer (high byte)
57097	$DF09	Interrupt mask register
57098	$DF0A	Address control register
53248-57343	$D000-$DFFF	Character pattern ROM (bank 14)

Chapter 9: Kerna

57344	$EO00	Performs power-on/reset sequence
57419	$E04B	Table of default MMU register settings
57430	$E056	Restores Kemal indirect vectors to their default values
57435	$E05B	Loads or copies Kernal indirect vector values
57459	E073	Table of default Kernal indirect vector values
57491	$E093	Initializes zero page and Kemal pointers
57549	$E0CD	Initializes all RAM-resident Kernal routines
57609	$E109	Initializes I/O chip registers
57820	$E1DC	Initializes 80-column video chip registers
57840	$E1FO	Initializes or jumps through the soft reset vector
57892	$E224	Initializes the soft reset vector
57922	$E242	Checks for the presence of 64 cartridges or 128 function ROMs
57931	$E24B	Switches the system into 64 mode
57963	$E26B	Logs 128 function ROMs
58052	$E2C4	Initialization test pattern
58055	$E2C7	Table of default VIC chip register values
58104	$E2FS	Table of default 8563 chip register values
58171	$E33B	Sends TALK command to a serial device
58174	$E33E	Sends LISTEN command to a serial device
58252	$E38C	Sends buffered byte to a serial device
58430	$E43E	Reads a byte from a serial device
58578	$E4D2	Sends secondary address after LISTEN
58583	$E4D7	Allows the serial bus ATN output line to go high
58592	$E4E0	Sends secondary address after TALK
58601	$E4E9	Performs talk-listen turnaround
58627	$E503	Sends a byte to a serial device
58645	$E515	Sends UNTALK command to a serial device
58662	$E526	Sends UNLISTEN command to a serial device
58693	$E545	Allows serial bus CLK output line to go high
58702	$E54E	Fulls serial bus CLK output line low
58711	$E557	Allows serial bus DATA output line to go high
58720	$E560	Pulls serial bus DATA output line low
58729	$E569	Reads the serial bus DATA and CLK input lines
58739	$E573	Disables IRQ interrupts and standardizes timing during I/O operations
58783	$E59F	Reenables interrupts and restores clock mode after I/O operations
58812	$E5BC	Performs fast serial turnaround
58819	$E5C3	Sets serial device for fast serial input

58838	SE5D6	Sets serial device for fast serial output
58875	$E5FB	Sets serial device for fast serial input or output
58879	$E5FF	Prepares next bit for RS-232 transmission
58907	$E61B	Prepares parity and stop bits
58948	$E644	Prepares to send a stop bit
58954	$E64A	Prepares to transmit next byte
59007	$E67F	Sets CIA interrupt register and RS-232 activity flag
59022	$E68E	Computes bit count for the RS-232 operation
59037	$E69D	Processes received bits
59058	$E6B2	Tests for stop bit
59074	$E6C2	Prepares to receive next byte
59092	SE6D4	Tests for start bit
59103	$E6DF	Stores received character in buffer and checks parity
59177	$E729	Handles CKOUT for RS-232 device
5922B	$E75C	Handles BSOUT for RS-232 device
59285	$E795	Handles CHKIN for RS-232 device
59342	$E7CE	Handles GETIN for RS-232 device
59372	$E7EC	Disables RS-232 activity during tape or serial bus operations
59397	$E805	Handles NMI interrupts for RS-232
59472	$E850	Table of baud rate timing constants for NTSC systems
59492	$E864	Table of baud rate timing constants for PAL systems
59512	$E878	Reads a bit from RS-232 device
59561	$E8A9	Initiates reception of RS-232 byte
59600	$E8D0	Reads next header block from tape
59673	$E919	Writes a header block to tape
59776	$E980	Loads and tests cassette buffer address
59783	$E987	Sets buffer address as block address
59802	$E99A	Searches for a specified header
59838	SE9BE	Checks for cassette buffer filled or emptied
59848	$E9C8	Requests PLAY button if necessary
59871	$E9DF	Checks tape buttons
59881	$E9E9	Requests RECORD and PLAY buttons if necessary
59890	$E9F2	Reads next header or data block from tape
59899	$E9FB	Reads or verifies a block from tape
59925	$EA15	Writes a header or data block to tape
59928	$EA18	Writes a block to tape
59942	$EA26	Initiates tape I/O operation
60047	$EA8F	Checks for RUN/STOP keypress during tape operations
60065	$EAA1	Sets timer A to check FLAG interrupts
60139	$EAEB	Reads or verifies a block of data from tape
60753	$ED51	Loads working pointer with starting address
60762	$ED5A	Initializes tape variables between each byte
60777	$ED69	Initiates writing of a tape half-dipoie
60816	$ED90	Writes a block of data to tape
60974	$EE2E	Writes a leader to tape and prepares to write a data block
61015	$EE57	Restores IRQ vector and operating modes after tape operation
61077	$EE95	Ends tape write interrupts
61083	$EE9B	Loads IIRQ vector for tape operation
61104	$EEB0	Turns cassette motor off
6U11	$EEB7	Tests whether ending address has been reached
61121	$EECI	Increments the working pointer

61128	$EEC8	Handles FLAG interrupts for tape
61136	$EED0	Controls tape motor interlock
61163	$EEEB	Retrieves a byte from the current input device
61190	$EF06	Accepts a byte from the current input device
61224	$EF28	Accepts a byte from tape
61276	$EF5C	Accepts a byte from a serial device
61287	$EF67	Accepts a byte from RS-232
61305	$EF79	Sends a byte to the current output device
61332	$EF94	Sends a byte to tape
61367	$EFB7	Sends a byte to the RS-232 port
61373	SEFBD	Opens a logical file to a specified device
61430	$EFF6	Opens a file for input or output to tape
61504	JF040	Opens a file for RS-232 communications
61616	$F0B0	Sets up CIA #2 ports for RS-232 communications
61643	$F0CB	Opens a file for serial bus communications
61702	$F106	Sets the current input file for GETIN and BASIN
61735	$F127	Prepares a serial device file for input
61772	$F14C	Sets the current output file for BSOUT
61805	SF16D	Prepares a serial device file for output
61832	$F188	Closes a specified logical file
61S65	$F1A9	Closes a tape file
61903	$F1CF	Closes a file on a serial device
61924	$F1E4	Removes an entry from the logical file tables
61954	$F202	Checks whether a file is already open
61970	$F212	Loads parameters for a logical Sle
61986	$F222	Clears file table entries
61990	$F226	Resets default I/O channels
62013	$F23D	Closes all open files for a specified serial device
62053	$F265	Loads or verifies a program file from disk or tape
62075	F276	Loads or verifies a file from a serial device
62246	$F326	Loads or verifies a program file from tape
62369	$F3A1	Attempts to set up fast serial load or verify
62442	$F3EA	Loads or verifies a file using fast serial burst mode
62616	$F496	Handles read error during burst mode load/verify
62630	$F4A6	Stops burst mode load/verify if RUN/STOP key pressed
62642	SF4B2	Aborts burst mode load/verify if maximum address exceeded
62650	SF4BA	Reads a byte using fast serial hardware
6266]	SF4C5	Loads or verifies a block of data using burst mode
62723	$F503	Toggles state of serial bus CLK line
62735	SF50F	Displays SEARCHING FOR message
62771	$F533	Displays LOADING or VERIFYING message
62782	SF53E	Saves a block of memory to tape or disk
62817	$F561	Saves a block of memory to a serial device
62878	SF59E	Closes a file on a serial device
62901	$F5B5	Aborts LOAD or SAVE to serial device
62908	$F5BC	Displays SAVING message and filename
62920	$F5C8	Saves a block of memory to tape
62968	$F5F8	Updates jiffy timers and checks RUN/STOP key column
63037	$F63D	Scans RUN/STOP key column
63070	$F65E	Reads the software jiffy clock
63077	$F665	Sets the software jiffy clock
63086	$F66E	Tests for a RUN/STOP keypress
63100	$F67C	Handles Kemal I/O errors

63152	$F6B0	Table of Kernal control messages
63262	$F71E	Handles Kemal control messages
63281	$F731	Sets the length and address of filename for I/O operations
63288	SF738	Sets logical file number, device number, and secondary address for I/O operations
63295	$F73F	Sets data and filename banks for I/O operations
63300	$F744	Reads the tape/serial or RS-232 status byte
63324	$F75C	Sets the Kemal message control flag
63327	$F75F	Sets the IEEE timeout flag
63331	$F763	Sets or reads the system's top-of-memory pointer
63346	$F772	Sets or reads the system's bottom-of-memory pointer
63361	$F781	Returns base address of I/O block
63366	$F?86	Checks whether a secondary address value is used
63389	$F79D	Checks whether a logical file number value is used
63397	$F7A5	Performs a DMA operation
63406	$F7AE	Retrieves a character from the current filename
63420	$F7BC	Writes a byte value to memory
63423	$F7BF	Writes a byte value to memory
63433	$F7C9	Reads a byte value from memory
63436	$F7CC	Reads a byte value from memory
63440	$F7D0	Retrieves a character from any bank
63450	$F7DA	Stores the accumulator contents in any bank
63459	$F7E3	Compares the accumulator contents with a value from any bank
63468	$F7EC	Translates a bank number into an MMU register setting
63472	$F7F0	Table of MMU register settings for standard banks
63488	$F800	Code for Kemal RAM-based subroutines
63591	$F867	Initializes function ROMs and attempts to boot a disk in the default drive
63632	SF890	Attempts to boot a disk
63883	$F98B	Resets the disk drive
63923	$F9B3	Loads additional *b*< Jt sectors
63995	$F9FB	Converts a byte value into two ASCII digits
64008	$FA08	Table of disk commands for booting
64023	$FA17	Handles PRIMM (print immediate) function
64064	$FA40	Handles NMI interrupts
64101	$FA65	Handles IRQ interrupts
64128	$FA80	Standard keyboard decoding tables

MMU (Memory Management Unit) Chip Registers

65280	5FF00	Configuration register
65281	$FF01	Load configuration register A
65282	$FF02	Load configuration register B
65283	$FF03	Load configuration register C
65284	$FF04	Load configuration register D
65285	$FF05	Jump to NMI handler routine
65303	$FF17	Jump to IRQ or BRK handler routine
65331	$FF33	Common exit routine for all interrupt routines
65341	SFF3D	Jump to reset handler routine
65351	$FF47	Entry point for the Kemal SPIIsLSPOUT routine
65354	$FF4A	Entry point for the Kemal CLOSF_J\LL routine
65357	$FF4D	Entry point for the Kemal C64—MODE routine
65360	$FF50	Entry point for the Kema! DMA_CALL routine
65363	$FF53	Entry point for the Kemal BOOT_CALL routine
65366	SFF56	Entry point for the Kemal PHOENIX routine

65369	$FF59	Entry point for the Kernal LKUPLA routine
65372	$FF5C	Entry point for the Kernal LKUTSA routine
65375	$FF5F	Calls the screen editor SWAPPER routine
65378	$FF62	Calls the screen editor INIT80 routine
65381	$FF65	Calls the screen editor KEYSET routine
65384	$FF68	Entry point for the Kemal JSETBNK routine
65387	$FF6B	Entry point for the Kemal GETCFG routine
65390	$FF6E	Entry point for the Kernal JSRFAR routine
65393	$FF71	Entry point for trie Kemal JMPFAR routine
65396	$FF74	Entry point for the Kemal INDFET calling routine
65399	$FF77	Entry point for the Kemal INDSTA calling routine
65402	$FF7A	Entry point for the Kemal INTJCMP calling routine
65405	$FF7D	Entry point for the Kemal PRIMM routine
65409	$FF81	Calls the screen editor CINT routine
65412	$FF84	Entry point for the Kernal IOINIT routine
65415	$FF87	Entry point for the Kemal RAMTAS routine
65418	$FF8A	Entry point for the Kemal RESTOR routine
65421	SFF8D	Entry point for the Kemal VECTOR routine
65424	$FF90	Entry point for the Kemal SETMSG routine
65427	$FF93	Entry point for the Kemal SECOND routine
65430	$FF96	Entry point for the Kemal TKSA routine
65433	$FF99	Entry point for the Kemal MEMTOP routine
65436	$FF9C	Entry point for the Kernal MEMBOT routine
65439	$FF9F	Calls the screen editor SCNKEY routine
65442	$FFA2	Entry point for the Kemal SETTMO routine
65445	$FFA5	Entry point for the Kemal ACPTR at routine
65448	$FFA8	Entry point for the Kemal C1OUT routine
65451	$FFAB	Entry point for the Kemal UNTLK routine
65454	$FFAE	Entry point for the Kemal UNLSN routine
65457	$FFB1	Entry point for the Kemal L1STN routine
65460	$FFB4	Entry point for the Kernal TALK routine
65463	$FFB7	Entry point for the Kemal READSS routine
65466	$FFBA	Entry point for the Kemal SETLFS routine
65469	$FFBD	Entry point for the Kemal SETNAM routine
65472	$FFC0	Entry point for the Kemal OPEN routine
65475	$EFC3	Entry point for the Kemal CLOSE routine
65478	$FFC6	Entry point for the Kemal CHKIN routine
65481	$FFC9	Entry point for the Kernal CKOUT routine
65484	$FFCC	Entry point for the Kemal CLRCH routine
65487	$FFCE	Entry point for the Kemal BASIN routine
65490	$FFD2	Entry point for the Kernal BSOUT routine
65493	$FFD5	Entry point for the Kemal LOAD routine
65496	$EFD8	Entry point for the Kemal SAVE routne
65499	$FFDB	Entry point for the Kemal SETT1M routine
65502	$FFDE	Entry point for the Kernal RDTIM routine
65505	$FFE1	Entry point for the Kernal STOP routine
65508	$FFE4	Entry point for the Kemal GETIN routine
65511	$FFE7	Entry point for the Kemal CLALL routine
65514	$FFEA	Entry point for the Kemal UDTIM routine
65? 17	$FFED	Calls the screen editor SCRORG routine
65520	$FFF0	Calls the screen editor PLOT routine
65523	$FFF3	Entry point for the Kernal IOBASE routine
65528	$FFF8	Soft reset vector (bank 1)
65530	$FFFA	Processor NMI vector
65532	SFFFC	Processor reset vector
65534	$IH1:	Processor BRK/IRQ vector

Here's the complete guire to understanding the inner workings of the Comrodore 128. Collected in this one volume are the answe's to practically any questions you might have about thy -vav +he Commodore 128 operates.

Moreover, the information presented here isn't just educational—it has tremendous practical value as well. Increasing your understanding of how the computer operates puts you more fully in control. Whether you program in BASIC or machine language, or whether you are just starting out or are an expert programmer, you'll find that this book unlocks secrets of the 128 that will allow you to write more powerful and more sophisticated programs. Inside you'll find:

- A complete reference to BASIC, the Kerndl, the screen editor, even the machine language monitor. Not just a list of addresses, but detailed discussions of all the important ROM routines.
« Thorough descriptions of all the video, sound, and Interface chips—VIC, SID, VDC, MMU, CIA, and REC. Every register of every chip is explained in detail.
- A clear and straightforward explanation of how the 128's elaborate memory management system operates, and how you can create your own custom memory configurations.
- Hundreds of suggestions and hints on how to take full advantage of the 128's enhanced capabilities.
- Examples of ways you can modify and improve the performance of your computer, including how to add new statements to BASIC and how to customize Kernal functions.
- Explanations of the bugs in BASIC and the Kernal, and how to avoid them.

If you've mastered the fundamentals of your 128 and are seeking to expand your knowledge of this powerful, versatile system. *Mapping the Commodore 128* can be your guide, If you're a veteran programmer who moved up to the 128 after years of experience on an older Commodore model, this book will help you adapt your established techniques to your new machine. For advanced programmers. *Mapping the Commodore 128* provides a wealth of ideas for exciting new programming projects, Much of the information gathered here is available elsewhere only in bits and scraps, if at all. You'll want to keep a copy close at hand every time you turn on your 128,

ISBN 9-87455-060-2