MAPPING THE Commodore 64 & 64C

Sheldon Leemon

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

Copyright 1987, 1984, 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 0-87455-082-3

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 directly, 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 Box 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 64C are trademarks of Commodore Electronics Limited. *GEOS* is a trademark of Berkeley Softworks.

Contents

Foreword v
Acknowledgments v
Introduction vii
Chapter 1. Page 0 1
Chapter 2. Page 1 45
Chapter 3. Pages 2 and 3
BASIC and Kernal Working Storage 49
Chapter 4. 1K to 40K
Screen Memory, Sprite Pointers, and
BASIC Program Text 79
Chapter 5. 8K BASIC ROM and 4K Free RAM 85
Chapter 6. VIC-II, SID, I/O Devices, Color RAM,
and Character ROM 119
Chapter 7. 8K Operating System Kernal ROM 203
Chapter 8. *GEOS* 247
Appendices
Appendix A. A Beginner's Guide to Typing In Programs 303
Appendix B. How to Type In Programs 305
Appendix C. Screen Location Table 307
Appendix D. Screen Color Memory Table 308
Appendix E. Screen Color Codes 309
Appendix F. ASCII Codes 311
Appendix G. Screen Codes 315
Appendix H. Commodore 64 Keycodes 317
Index (By Memory Location) 318
***GEOS* Index** (By Memory Location) 322

Foreword

Since it was released in 1984, this definitive guide to the Commodore 64's memory has gone through four printings and proved indispensable to thousands of 64 programmers.

The Commodore 64 is now the Commodore *64C*. The same computer, though in a different case, the 64C includes *GEOS*, the new Graphics Environment Operating System. Icons, pull-down menus, and a desktop appearance make the 64C and *GEOS* easy to learn and easy to use. *GEOS* is an important addition for the Commodore 64. If you upgraded your Commodore 64 by purchasing *GEOS*, if you got *GEOS* when you bought the 64C, or even if you haven't yet added *GEOS* to your system, you'll find this book a valuable addition to your computer library. Just like its predecessor, *Mapping the Commodore 64 and 64C* contains a complete memory map of the Commodore 64 and 64C computers, presented in concise and understandable language.

A memory map is a list of the memory locations in a computer. But *Mapping the Commodore 64 and 64C* is a memory map that goes much further. It explains the *purpose* of each location and shows you how to change the contents of many of them. You can make the computer do what *you* want it to. It also includes a complete map of *GEOS*, the new graphics environment from Berkeley Softworks.

If you're a BASIC programmer, you'll find easy-to-understand explanations of how to use the advanced features of the Commodore 64 and 64C, and how to include these features in your own programs. If you're already using machine language to write your own programs, you'll use this guide over and over again, referring to specific memory locations and routines. You'll have the most complete guide to Commodore 64 and 64C memory available.

Acknowledgements

I would like to thank Russ Davies, author of *Mapping the VIC*, for his generosity in sharing the results of his research. Without his help, and that of Dan Heeb, author of *The Commodore 64 And VIC Tool Kit*, this book would have been much less complete.

Of all the published sources of information about the 64, I have found the *Commodore 64 Programmer's Reference Guide* to be the most valuable for technical information. That Commodore has come out with such a thorough reference work so quickly after the introduction of the machine is much to their credit. Because of the similarities between the 64 and previous Commodore computers, I have

also relied heavily on books that deal with the PET/CBM. Of particular interest are *Programming the PET/CBM* by Raeto West, which should be subtitled "Everything You Ever Wanted to Know about Commodore Computers," and the *PET/CBM Personal Computer Guide,* Third Edition, Osborne/McGraw-Hill. These important reference works contain more information of real benefit to the 64 owner than most books which deal specifically with the 64.

Finally, I would like to thank my wife Lenore. Although her contribution to this book was nontechnical in nature, it was as important as any other.

Introduction

To many computer users, the concept of a memory map may be an unfamiliar one. Simply stated, it is a guide to your computer's internal hardware and software. A memory map can help you use the PEEK and POKE instructions to extend your BASIC programming powers. If you plan to do machine language programming, it is necessary that you be able to find your way around the system.

Many computer owners think of a program as something they buy at a store and feed into the computer so that it will let them play games, or do word processing, or keep financial records. But this type of *applications* software is by no means the only kind.

It is important to remember that a computer cannot do *anything* without some kind of program. When the computer displays the READY prompt, or blinks the cursor, or displays the letters that you type in, it can only do so by executing a program. In the examples above, it is the master control program, the Operating System (OS), which is being executed.

When you give the computer a command such as LOAD, the BASIC interpreter program translates the English-like request to the language of numbers which the computer understands. The Operating System and BASIC interpreter programs are contained in permanent Read Only Memory (ROM), and are available as soon as you turn the power on. Without them, your computer would be a rather expensive paperweight.

This permanent software co-exists inside your computer with the applications program that you load. Since the system software performs many of the same functions as the applications program (such as reading information from the keyboard and displaying it on the screen), it is often possible for the applications program to make use of certain parts of the Operating System. This not only makes the task of the programmer easier, but in some cases it allows him or her to do things that otherwise would not be possible.

The Commodore 64 also has hardware support chips which enable the graphics display, sound synthesis, and communications with external devices. Since these chips are addressed like memory, they occupy space in our map. Control

over these chips, and the graphics, sound, and I/O functions they make possible, can only be accomplished by manipulation of the memory addresses which correspond to these devices. Therefore, a guide to these addresses is necessary in order to take advantage of the graphics, music, and communications power that they offer.

The purpose of this book is to describe the memory locations used by the system, and to show, wherever possible, how to utilize them in your own programs. The book should clear up some of the mystery surrounding the way your computer works.

How to Use This Book

The Commodore 64 can communicate with 64K or 65536 (64*1024) bytes of memory. Each of these bytes of memory can store a number from 0 to 255. The computer can read (and sometimes write) information from any of these 65536 locations by using its address, a number from 0 to 65535.

Bits and Bytes

Each byte is made up of eight binary digits called bits. These bits are the computer's smallest unit of information. They can contain only the number one or the number zero, but when combined, they can be used to form any number needed. To see how that is possible, let's look at a single bit.

A single bit can only be a one or a zero. But if two bits are combined the number increases.

00,01,10,11

That makes four possible combinations. And if a third bit is added:

000,001,010,011,100,101,110,111

When eight bits are put together, the number of combinations increases to 256. These eight bits are called a byte and can be used to represent the numbers from 0 to 255.

This system of numbering is known as binary (or base two). It works much like the decimal (base ten) system. In the base ten numbering system, the rightmost digit is known as the one's place, and holds a number from 0 to 9. The next place to the left is known as the ten's place, and also holds a number from 0 to 9, which represents the number of times the one's place has been used in counting (the number of tens).

In the binary system, there is a one's place, then a two's

place, a four's place, etc. The bits are counted from right to left, starting with Bit 0. Here are the values of each bit:

Bit 0 = 1
Bit 1 = 2
Bit 2 = 4
Bit 3 = 8
Bit 4 = 16
Bit 5 = 32
Bit 6 = 64
Bit 7 = 128

If all the bits are added together (128+64+32+16+8+4+2+1), they total 255, which is the maximum value of one byte. What if you need to count higher than 255? Use two bytes.

By using a second byte to count the number of 256's, 65536 combinations are possible (256*256). This is the same number as the bytes of memory in the Commodore 64. Therefore, any byte can be addressed by a number using a maximum of two bytes.

When discussing large, even units of memory, the second byte, the number of 256's, is often used alone. These units are known as *pages.* Page 0 starts at location zero (0*256), Page 1 starts at location 256 (1*256), etc.

You may see the terms *low-byte, high-byte order* or LSB (Least Significant Byte), MSB (Most Significant Byte) order, mentioned later in this book. That refers to the way in which the Commodore 64 usually deals with a two-byte address. The byte of the address that represents the number of 256's (MSB) is usually stored higher in memory than the part that stores the leftover value from 0 to 255 (LSB). Therefore, to find the address, you must add the LSB to 256* MSB.

Hexadecimal

One other numbering system that is used in speaking about computers is the hexadecimal (base 16) system. Each hexadecimal digit can count a number from 0 to 15. Since the highest numeric digit is 9, the alphabet must be used: A=10, B=11, and so on up to F=15. With just two digits, 256 combinations are possible (16*16). That means that each byte can be represented by just two hexadecimal digits, each of which stands for four bits of memory. These four-bit units are smaller than a byte, so they are known as nybbles.

Since programmers often find that hexadecimal numbers are easier to use than binary or decimal numbers, many numbers in this book are given in both decimal and hexadecimal format. A dollar sign ($) has been placed in front of each hexadecimal number.

AND, OR and EOR

Certain functions on the 64 (particularly those using the sound and graphics chips) are controlled by a single bit. You will often see references to setting Bit 6 to a value of one, or setting Bit 3 to a value of zero. This can be done by adding or subtracting the bit value for that particular bit from the value of the whole byte.

Adding or subtracting the bit value will work only if you know the status of that bit already. If Bit 4 is off, and you add 16 (the bit value of Bit 4) to the byte, it will turn Bit 4 on. But if it were on already, adding 16 would turn Bit 4 off, and another bit on.

This is where logical (sometimes called Boolean) functions come in handy. Two functions, OR and AND, are available in BASIC, and a third, EOR, can be used in machine language programming for bit manipulation.

AND is usually used to zero out (or mask) unwanted bits. When you AND a number with another, a 1 will appear in the resulting number only if identical bits in the ANDed numbers had been set to 1. For example, if you wanted to turn off Bit 4 in the number 154, you could AND it with 239 (which is the maximum bit combination minus the bit value to be masked; 255-16):

```
      10011010   =   154
AND   11101111   =   239
      ----------
    = 10001010   =   138
```

By using the AND function, nothing would be harmed if we tried to turn off a bit that wasn't on. You can always turn a bit off by using the formula BYTEVALUE AND (255–BITVALUE). Remember, there must be a 1 in the same bit of both numbers in order for the same bit in the result to have a 1.

The opposite function, turning a bit on, is performed by the OR statement. The OR function puts a 1 in the bit of the resulting number if there was a 1 in the same bit in either of the two numbers. For example, to turn Bit 4 back on in the

number 138, we would use the statement 138 OR 16 (the bit value for the bit we want to turn on):

```
  10001010  =  138
OR 00010000  =   16
-----------
= 10011010  =  154
```

Again, no harm would be done if the bit was already on. A bit can always be turned on with the formula BYTEVALUE OR BITVALUE.

The third operation, EOR, can be done only in machine language. It is used to reverse the value of a bit. If the bit of the second number holds a 1, it will reverse the value of the corresponding bit in the first number. Therefore, to switch all of the bits, you can EOR a number by 255:

```
  10011010  =  154
EOR 11111111  =  255
------------
= 01100101  =  101
```

Notice that this produces the complement of the original number (256 minus the number). Anytime you wish to flip a bit from 0 to 1, or 1 to 0, you can EOR it with the value of that bit.

The Format of Entries

The entries in this book are organized at the most general level by large blocks (the 256 page 0 locations, for example, or the 8192 locations in the BASIC ROM). At the beginning of these blocks, you will often find an explanation that will give you an idea of how the locations in that section are related. Usually, this overview will make it easier to understand the more detailed explanations of the individual locations in that block.

Within these larger blocks, you will sometimes see a few locations grouped under the heading *Location Range.* This grouping is done where the locations are so interrelated that it would be meaningless to try to explain one without explaining them all.

Finally come the entries for individual locations. These give the address, first in decimal, then in hexadecimal, and sometimes a label. This label is merely a mnemonic device, used in machine language programming for easier reference to

a particular memory location. Although Commodore has not released the source code for their Operating System or BASIC, they have published some of these labels in the *Commodore 64 Programmer's Reference Guide.*

Other labels used here are taken from Jim Butterfield's PET memory maps, which have enjoyed a wide circulation among Commodore users. Their use here should help 64 owners adapt information about the PET to their own machines.

The mnemonic label for an entry is followed by a one-line description of the location. Often, a more detailed explanation will appear under that, ranging from a couple of sentences to several pages. Occasionally, program samples will accompany these explanations.

Sometimes the single-line descriptions will identify a location as a flag, a vector, or a pointer. A flag is just a number that the program uses to store the outcome of a previous operation. A pointer or vector is usually a two-byte location that holds a significant address. Generally, the term *pointer* is used when the address points to the start of some data, and *vector* is used when the address points to the start of a machine language program. However, sometimes these terms will be used interchangeably, with the meaning clear from the context.

Chapter 1

Page 0

Page 0

Memory locations 0-255 ($0-$FF) have a special significance in 6502 machine language programming (the 6510 microprocessor in the Commodore 64 shares the same instruction set as the 6502). Since these addresses can be expressed using just one byte, instructions which access data stored in these locations are shorter and execute more quickly than do instructions which operate on addresses in higher memory, which require two bytes.

Because of this relatively fast execution time, most 6502 software makes heavy use of zero-page locations. The 64 is no exception, and uses this area for many important system variables and pointers.

In addition, locations 0 and 1 have special Input/Output functions on the 6510. In the case of the 64, this on-chip I/O port is used to select the possible combinations of ROM, as we will see below, and to control cassette I/O.

Location Range: 0-143 ($0-$8F)

BASIC Working Storage

This portion of zero page is used by BASIC only. Therefore, a program written entirely in machine language that does not interact with BASIC can freely use this area.

0 $0 D6510

6510 On-Chip I/O DATA Direction Register

Bit 0: Direction of Bit 0 I/O on port at next address. Default=1 (output)
Bit 1: Direction of Bit 1 I/O on port at next address. Default=1 (output)
Bit 2: Direction of Bit 2 I/O on port at next address. Default=1 (output)
Bit 3: Direction of Bit 3 I/O on port at next address. Default=1 (output)
Bit 4: Direction of Bit 4 I/O on port at next address. Default=0 (input)
Bit 5: Direction of Bit 5 I/O on port at next address. Default=1 (output)
Bit 6: Direction of Bit 6 I/O on port at next address. Not used.
Bit 7: Direction of Bit 7 I/O on port at next address. Not used.

1

This location is the first of a number of *hardware registers* that we will discuss. Although they can be written to and/or read like RAM, they are connected to hardware devices, and their contents affect the operation of the devices.

Each bit of this Data Direction Register determines whether the contents of the corresponding bit on the Internal I/O Port (see location 1) can be written to by peripheral devices. If the bit is set to 0, it indicates the direction of data flow as Input, which means that the corresponding bit of the I/O Port *will* be affected by peripheral devices. If the bit is set to 1, it indicates Output. On the 64, only Bits 0-5 are significant. On power-up, this register is set to 239 ($EF), which indicates that all bits, except for Bit 4 (which senses the cassette switch), are set up for Output.

1 $1 R6510

Bit 0: LORAM signal. Selects ROM or RAM at 40960 ($A000). 1=BASIC, 0=RAM
Bit 1: HIRAM signal. Selects ROM or RAM at 57344 ($E000). 1=Kernal, 0=RAM
Bit 2: CHAREN signal. Selects character ROM or I/O devices. 1= I/O, 0=ROM
Bit 3: Cassette Data Output line.
Bit 4: Cassette Switch Sense. Reads 0 if a button is pressed, 1 if not.
Bit 5: Cassette Motor Control. A 1 turns the motor on, a 0 turns it off.
Bits 6-7: Not connected—no function presently defined.

The chief function of this register is to determine which blocks of RAM and ROM the 6510 microprocessor will address. The Commodore 64 comes with 64K RAM, even though it normally does not use all of that RAM at once. In addition, it has an 8K BASIC Interpreter ROM, an 8K Operating System Kernal ROM, a 4K Character Generator ROM, a Sound Interface Device (SID), a 6566 Video Interface Controller (VIC-II), and two 6526 Complex Interface Adapter chips.

To address all of these at once would require 88K, 24K past the addressing limit of the 6510 microprocessor. In order to allocate address space, the I/O Port is used to affect the addressing lines, and thus determine which segments of RAM and ROM will be addressed at any one time.

Bit 0. This bit controls the LORAM signal. A 0 in this bit position switches the BASIC ROM out, and replaces it with RAM at addresses 40960-49151 ($A000-$BFFF). The default value of this bit is 1.

Bit 1. Bit 1 controls the HIRAM signal. A 0 in this bit position switches the Kernal ROM out, and replaces it with RAM at 57344-65535 ($E000-$FFFF). As the BASIC interpreter uses the Kernal, it is also switched out and replaced by RAM. The default value of this bit is 1.

This system allows a wide range of combinations of RAM and ROM to be utilized. Of course, the BASIC programmer will have little need, in the ordinary course of events, to switch out the BASIC ROM and the Kernal. To do so without first replacing them would just hang the system up. But one way to make use of this feature is to move the contents of ROM to the corresponding RAM addresses. That way, you can easily modify and customize the BASIC interpreter and OS Kernal routines, which are ordinarily fixed in ROM. For example, to move BASIC into RAM, just type:

```
FOR I=40960 TO 49151:POKE I, PEEK (I):NEXT
```

Though it appears that such a program would not do anything, it in fact copies bytes from ROM to RAM. This is because any data which is written to a ROM location is stored in the RAM which resides at the same address. So while you are PEEKing ROM, you are POKEing RAM. To switch to your RAM copy of BASIC, type in:

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

Now you are ready to make modifications. Examples of simple modifications include changing the text which the interpreter prints, such as the READY prompt, the power-up message, or the keyword table.

An example of the latter would be POKE 41122,69. This changes the FOR keyword to FER, so that BASIC would respond normally to a FER-NEXT loop, but fail to recognize FOR as syntactically correct.

On the more practical side, you could change the prompt that INPUT issues to a colon, rather than a question mark:

```
POKE 43846,58
```

You are not limited to just cosmetic changes of text. Jim Butterfield has given an example in *COMPUTE!* magazine of changing the interpreter so that it assigns a null string the ASCII value 0. In the ROM version, the command PRINT ASC ("") will return ?ILLEGAL QUANTITY ERROR. This is inconvenient when INPUTting a string, because if the user presses RETURN and you try to check the ASCII value of the string that has been entered, you will get this error. By entering POKE 46991,5, this is changed so that PRINT ASC ("") now responds with a zero.

For the more serious machine language programmer, it is quite feasible to add new commands or modify existing ones by diverting the vectors which are discussed in the section covering the BASIC interpreter ROM. For a good example of this technique, see the article "Hi-Res Graphics Made Simple" by Paul Schatz in *COMPUTE!'s First Book of Commodore 64 Sound and Graphics*. The program example there inserts new graphics commands into a RAM version of BASIC. When you want to switch back to the ROM BASIC, enter POKE 1,PEEK (1) OR 1.

For machine language applications, it would be possible to replace the ROM programs with an entirely different operating system, or an application that has its own screen editing and I/O functions included. Such an application would first have to be loaded from disk into RAM. A language other than BASIC could be loaded, and could then just switch out the BASIC ROM, while still using the OS Kernal.

Or a spreadsheet application that contained its own I/O routines could switch out all ROMs and have the use of all of RAM that is not actually needed for the program itself, for data. It should be remembered, however, that before switching the Kernal out, it is necessary to disable interrupts, as the vectors for these interrupts are contained in the Kernal.

Bit 2. This bit controls the CHAREN signal. A 0 in this position switches the character generator ROM in, so that it can be read by the 6510 at addresses 53248-57343 ($D000-$DFFF). Normally, this bit is set to 1, so that while the VIC-II chip has access to the character generator ROM for purposes of creating the screen display, the user cannot PEEK into it. Since this ROM is switched into the system in the same location as the I/O devices (SID chip, VIC-II chip, and 6526 CIA's), no I/O can occur when this ROM is switched in.

The ability to switch in the character generator ROM is very useful to the programmer who wishes to experiment with user-defined characters. Modified character graphics is one of the more powerful graphics tools available, but often the user will not want to redefine a whole character set at one time. By reading the character ROM and duplicating its contents in RAM, the user can replace only a few characters in the set. The method for reading this ROM into RAM from BASIC is as follows:

```
10 POKE 56333,127:POKE1,PEEK(1) AND 251: FOR I=0 TO 2048
20 POKE BASE+I,PEEK(53248+I):NEXT:POKE 1,PEEK(1) OR 4:
   POKE 56333,129
```

The first POKE is necessary to turn off the system timer interrupt. Since the I/O devices are addressed in the same space as the character ROM, switching that ROM in switches all I/O out, making it necessary to turn off any interrupts which use these devices.

The second POKE is the one which switches in the character ROM. The program loop then reads this ROM memory into RAM, starting with the address BASE. Note that this address should start on an even 2K boundary (an address evenly divisible by 2048) within the block of memory presently being addressed by the VIC-II chip (for more information on where to put user-defined character sets, and how to use them, see the section on the VIC-II chip, under location 53272 ($D018), the section on the character ROM at 53248 ($D000), and the section on banking VIC-II memory at 56576 ($DD00)). After reading the contents of the ROM into RAM, the next

POKEs switch out the character ROM, and restore the interrupt.

It should be noted that while Bits 0-2 of this register allow software control of some signals that determine the memory configuration that is used by the Commodore 64 at any given time, they are not the only determining factor. Signals can also be generated by means of plug-in expansion cartridges which are connected to the expansion port, and these can change the memory map.

Two lines located on the expansion port are called GAME and EXROM. When used in conjunction with the software-controlled lines noted above, these two hardware lines can enable cartridge ROM to replace various segments of ROM and/or RAM.

Possible configurations include 8K of cartridge ROM to be switched in at $8000-$9FFF, for a BASIC enchancement program; an 8K cartridge ROM at $A000-$BFFF, replacing BASIC; or at $E000-$FFFF, replacing the Kernal, or a 16K cartridge at $8000-$C000.

When cartridge ROM is selected to replace the Kernal, a Max emulator mode is entered, which mimics the specification of the ill-fated Max Machine, a game machine which Commodore never produced for sale in the U.S. In this mode, only the first 6K of RAM are used, there is no access to the character ROM, and graphics data such as character dot-data is mapped down from 57344 ($E000) to 8192 ($2000). Further hardware information may be obtained from the *Commodore 64 Programmer's Reference Guide.*

Bits 3-5 of this register have functions connected with the Datassette recorder. These are as follows:

Bit 3. This is the Cassette Data Output line. This line is connected to the Cassette Data Write line on the cassette port, and is used to send the data which is written to tape.

Bit 4. This bit is the Cassette Switch Sense line. This bit enables a program to tell whether or not one of the buttons that moves the tape on the cassette recorder is pressed down. If the switch on the recorder is down, this bit will have a value of 1. Remember that Bit 4 of the data direction register at location 0 must contain a 0 for this bit to properly reflect the status of the switch.

Bit 5. Bit 5 is the Cassette Motor Control. Setting this bit to zero allows the motor to turn when you press one of the buttons on the recorder, while setting it to one disables it from turning.

Most of the time, the setting of this bit will be controlled by the interrupt routine that is used to read the keyboard every sixtieth of a second. If none of the buttons on the recorder is pressed, that interrupt routine shuts the motor off and sets the interlock at location 192 ($C0) to zero. When a button is pressed, if the interlock location is zero, Bit 5 of this register is set to zero to turn the motor on.

When the interlock location contains a zero, the keyscan routine will not let you control the setting of this bit of the register (and the interlock is always set to zero when no buttons are pressed). In order

for you to gain control of the motor, you must POKE a nonzero value into 192 after a button on the recorder has been pressed. You can then shut off the motor and turn it back on as you please, by manipulating this bit, so long as a button stays pressed.

2 $2

Unused

3-4 $3-$4 ADRAY1

Vector: Routine to Convert a Number from Floating Point to Signed Integer

This vector points to the address of the BASIC routine which converts a floating point number to an integer. In the current Kernal version, the address that it points to is 45482 ($B1AA). Disassembly of the ROMs indicates that BASIC does not use this vector. However, it may be of real assistance to the programmer who wishes to use data that is stored in floating point format. The parameter that is passed by the USR command is available only in that format, for example.

Since it is extremely difficult to decipher and use a floating point number, the simplest way to deal with such data is to use the conversion routines that are built into BASIC to change it into a two-byte signed integer. This could be accomplished by jumping directly into the BASIC ROM, if you know the location of the routine. It is preferable to use this vector because it will always point to the location of the routine. Therefore, if the address changes in future versions of the 64 or future Commodore computers, you won't have to modify your program to make it work with them.

See the entry for the USR vector at 785 ($311) for an explanation of how to use this routine in connection with the USR command.

5-6 $5-$6 ADRAY2

Vector: Routine to Convert a Number from Integer to Floating Point

This vector points to the address of the BASIC routine which converts an integer to a floating point number. This routine is currently located at 45969 ($B391). BASIC does not appear to reference this location. It is available for use by the programmer who needs to make such a conversion for a machine language program that interacts with BASIC. For an explanation of how to use this routine in connection with the USR command, see the entry for the USR vector at 785 ($311).

7 $7 CHARAC

Search Character for Scanning BASIC Text Input

This location and the next are used heavily by the BASIC routines that scan the text that comes into the buffer at 512 ($200), in order to detect significant characters such as quotes, comma, the colon which separates BASIC statements, and end-of-line. The ASCII values of such special characters are usually stored here.

This location is also used as a work area by other BASIC routines that do not involve scanning text.

8 $8 ENDCHR

Search Character for Statement Terminator or Quote

Like location 7, this location is used as a work byte during the tokenization of a BASIC statement. Most of the time, its value is 0 or 34.

9 $9 TRMPOS

Column Position of the Cursor before the Last TAB or SPC

TRMPOS is used by TAB and SPC. The cursor column position prior to the TAB or SPC is moved here from 211 ($D3), and is used to calculate where the cursor ends up after one of these functions is invoked. Note that the value contained here shows the position of the cursor on a logical line. Since one logical line can be up to two physical lines long, the value stored here can range from 0 to 79.

10 $A VERCK

Flag: LOAD or VERIFY

BASIC uses one Kernal routine to perform either the LOAD or VERIFY function, depending on whether the Accumulator (.A) is set to 0 or 1 upon entry to the routine. BASIC sets the value of VERCK to 0 for a LOAD, or 1 for a VERIFY. Its contents are passed to the Kernal LOAD routine, which in turn stores it in location 147 ($93).

11 $B COUNT

Index into the Text Input Buffer/Number of Array Subscripts

The routines that convert the text in the input buffer at 512 ($200) into lines of executable program tokens, and the routines that link these program lines together, use this location as an index into the input buffer area. When the job of converting text to tokens is finished, the value in this location is equal to the length of the tokenized line.

The routines which build an array or locate an element in an array use this location to calculate the number of DIMensions called for and the amount of storage required for a newly created array, or the number of subscripts specified when referencing an array element.

12 $C DIMFLG

Flags for the Routines That Locate or Build an Array

This location is used as a flag by the routines that build an array or reference an existing array. It is used to determine whether a variable is an array, whether the array has already been DIMensioned, and whether a new array should assume the default dimensions.

13 $D VALTYP

Flag: Type of Data (String or Numeric)

This flag is used internally to indicate whether data being operated upon is string or numeric. A value of 255 ($FF) in this location indicates string data, while a 0 indicates numeric data. This determination is made every time a variable is located or created.

14 $E INTFLG

Flag: Type of Numeric Data (Integer or Floating Point)

If data which BASIC is using is determined to be numeric, it is further classified here as either a floating point number or as an integer. A 128 ($80) in this location identifies the number as an integer, and a 0 indicates a floating point number.

15 $F GARBFL

Flag for LIST, Garbage Collection, and Program Tokenization

The LIST routine uses this byte as a flag to let it know when it has come to a character string in quotes. It will then print the string, rather than search it for BASIC keyword tokens.

The garbage collection routine uses this location as a flag to indicate that garbage collection has already been tried before adding a new string. If there is still not enough memory, an OUT OF MEMORY message will result.

This location is also used as a work byte for the process of converting a line of text in the BASIC input buffer (512, $200) into a linked program line of BASIC keyword tokens.

16 $10 SUBFLG

Flag: Subscript Reference to an Array or User-Defined Function Call (FN)

This flag is used by the PTRGET routine which finds or creates a variable, at the time it checks whether the name of a variable is valid. If an opening parenthesis is found, this flag is set to indicate that the variable in question is either an array variable or a user-defined function.

You should note that it is perfectly legal for a user-defined function (FN) to have the same name as a floating point variable. Moreover, it is also legal to redefine a function. Using a FN name in an already defined function results in the new definition of the function.

17 $11 INPFLG

Flag: Is Data Input to GET, READ or INPUT?

Since the keywords GET, INPUT, and READ perform similar functions, BASIC executes some of the same instructions for all three. There are also many areas of difference, however, and this flag indicates which of the three keywords is currently being executed, so that BASIC will know whether or not to execute the instructions which relate to the areas in which the commands differ (152 ($98)=READ, 64 ($40)= GET, 0=INPUT).

As a result, INPUT will show the ? prompt, will echo characters back to the screen, and will wait for a whole line of text ended by a carriage return. GET gives no prompt and accepts one character without waiting. The colon character and the comma are valid data for GET, but are treated as delimiters between data by INPUT and READ.

As each command has its own error messages, this flag is used to determine the appropriate message to issue in case of an error.

18 $12 TANSGN

Flag: Sign of the Result of the TAN or SIN Function

This location is used to determine whether the sign of the value returned by the functions SIN or TAN is positive or negative.

Additionally, the string and numeric comparison routines use this location to indicate the outcome of the comparison. For a comparison of variable A to variable B, the value here will be 1 if A is greater than B, 2 if A equals B, and 4 if A is less than B. If more than one comparison operator was used to compare the two variables (e.g., >= or <=), the value here will be a combination of the above values.

19 $13 CHANNL

Current I/O Channel (CMD Logical File) Number

Whenever BASIC inputs or outputs data, it looks here to determine which I/O device is currently active for the purpose of prompting or output control. It uses location 184, $B8 for purposes of deciding what device actually to input from or output to.

When the default input device (number 0, the keyboard) or output device (number 3, the display screen) is used, the value here will be a zero, and the format of prompting and output will be the standard screen output format.

When another device is used, the logical file number (CMD channel number) will be placed here. This lets the system know that it may have to make some subtle changes in the way it performs the I/O operation. For example, if TAB is used with the PRINT command, cursor right characters are used if the device PRINTed to is the screen. Otherwise, spaces are output when the number here is

other than zero (the assumption being that you can't tab a printer like you can the screen).

Likewise, the ? prompt for INPUT is suppressed if the file number here is nonzero, as is the EXTRA IGNORED message, and input of a carriage return by itself is ignored, rather than being treated as a null string (""). Therefore, by OPENing the screen as a device, and issuing the CMD statement, you can force the suppression of the ? prompt, and the other effects above.

CMD places the new output file number here, and calls the Kernal to open the device for output, leaving it LISTENing for output (such as the READY prompt, which is diverted to the new device).

Many routines reset this location and UNLISTEN the device, defeating the CMD and once again sending output to the screen. If an error message has to be displayed, for example, this location will be reset and the message will be displayed on the screen. GET, GET#, INPUT, INPUT#, and PRINT# all will reset this location after the I/O is completed, effectively redirecting output back to the screen. PRINT and LIST are the only I/O operations that will not undo the CMD.

This location can also be used to fool BASIC into thinking that data it is reading from the tape is actually being entered into the keyboard in immediate mode.

For a look at a technique that uses a different approach to accomplish the same thing for disk or tape users, see location 512 ($200), the keyboard buffer.

20-21 $14-$15 LINNUM

Integer Line Number Value

The target line number for GOTO, LIST, ON, and GOSUB is stored here in low-byte, high-byte integer format, as is the number of a BASIC line that is to be added or replaced.

LIST saves the highest line number to list (or 65535—$FFFF if program is to be listed to the end) at this location.

GOTO tests the target line number to see if it is greater than the line number currently being executed. If it is greater, GOTO starts its search for the target line at the current line number. If it is not greater, GOTO must search for the target line from the first line of the program. It is interesting to note that the test is of the most significant byte only. Therefore, INT (TARGETLINE/256) must be greater than INT(CURRENT LINE/256) in order for the search to start with the current line, instead of at the beginning of the program.

PEEK, POKE, WAIT, and SYS use this location as a pointer to the address which is the subject of the command.

22 $16 TEMPPT

Pointer to the Next Available Space in the Temporary String Stack

This location points to the next available slot in the temporary string descriptor stack located at 25-33 ($19-$21). Since that stack has room for three descriptors of three bytes each, this location will point to 25 ($19) if the stack is empty, to 28 ($1C) if there is one entry, to 31 ($1F) if there are two entries, and to 34 ($22) if the stack is full.

If BASIC needs to add an entry to the temporary string descriptor stack, and this location holds a 34, indicating that the stack is full, the FORMULA TOO COMPLEX error message is issued. Otherwise, the entry is added, and three is added to this pointer.

23-24 $17-$18 LASTPT

Pointer to the Address of the Last String in the Temporary String Stack

This pointer indicates the last slot used in the temporary string descriptor stack. Therefore, the value stored at 23 ($17) should be 3 less than that stored at 22 ($16), while 24 ($18) will contain a 0.

25-33 $19-$21 TEMPST

Descriptor Stack for Temporary Strings

The temporary string descriptor stack contains information about temporary strings which have not yet been assigned to a string variable. An example of such a temporary string is the literal string "HELLO" in the statement PRINT "HELLO".

Each three-byte descriptor in this stack contains the length of the string, and its starting and ending locations, expressed as displacements within the BASIC storage area.

34-37 $22-$25 INDEX

Miscellaneous Temporary Pointers and Save Area

This area is used by many BASIC routines to hold temporary pointers and calculation results.

38-42 $26-$2A RESHO

Floating Point Multiplication Work Area

This location is used by BASIC multiplication and division routines. It is also used by the routines which compute the size of the area required to store an array which is being created.

43-44 $2B-$2C TXTTAB

Pointer to the Start of BASIC Program Text

This two-byte pointer lets BASIC know where program text is stored. Ordinarily, such text is located beginning at 2049 ($801). Using this

pointer, it is possible to change the program text area. Typical reasons for doing so include:

1. Conforming the memory configuration to that of other Commodore computers. On 32K PET and CBM computers, for example, screen memory starts at 32768 ($8000), and BASIC text begins at 1025 ($401). You can emulate this configuration with the 64 with the following short program:

```
10 POKE 55,0:POKE 56,128: CLR: REM LOWER TOP OF ME
   MORY TO 32768
20 POKE 56576,PEEK(56576) AND 253: REM{2 SPACES}EN
   ABLE BANK 2
30 POKE 53272,4: REM TEXT DISPLAY MEMORY NOW START
   S AT 32768
40 POKE 648,128:REM OPERATING SYSTEM PRINTS TO SCR
   EEN AT 32768 (128*256)
50 POKE 44,4:POKE 1024,0: REM MOVE START OF BASIC
   {SPACE}TO 1025 (4*256+1)
60 POKE 792,193: REM DISABLE RESTORE KEY
70 PRINT CHR$(147);"NOW CONFIGURED LIKE PET":NEW
80 REM ALSO SEE ENTRIES FOR LOCATION 55, 56576, AN
   D 648
```

Such reconfiguring can be helpful in transferring programs from the 64 to the PET, or vice versa. Since the 64 automatically relocates BASIC program text, it can load and list PET programs even though the program file indicates a loading address that is different from the 64 start of BASIC. The PET does not have this automatic relocation feature, however, and it loads all BASIC programs at the two-byte address indicated at the beginning of the disk or tape file.

So if the PET loads a 64 program at its normal starting address of 2049 ($801), it will not recognize its presence because it expects a BASIC program to start at 1025 ($401). Therefore, if you want to let a PET and 64 share a program, you must either reconfigure the 64 to start BASIC where the PET does, or reconfigure the PET to start BASIC where the 64 does (with a POKE 41,8:POKE 2048,0).

2. Raising the lowest location used for BASIC text in order to create a safe area in low memory. For example, if you wish to use the high-resolution graphics mode, you may want to put the start of screen memory at 8192 ($2000). The high-resolution mode requires 8K of memory, and you cannot use the lowest 8K for this purpose because it is already being used for the zero-page assignments.

Since BASIC program text normally starts at 2049 ($801), this means that you only have 6K for program text before your program runs over into screen memory. One way around this is by moving the start of BASIC to 16385 ($4001) by typing in direct entry mode:

POKE 44,64: POKE 64*256, 0:NEW

Other uses might include setting aside a storage area for sprite shape data, or user-defined character sets.

3. Keeping two or more programs in memory simultaneously. By changing this pointer, you can keep more than one BASIC program in memory at one time, and switch back and forth between them. Examples of this application can be found in *COMPUTE!'s First Book of PET/CBM*, pages 66 and 163.

This technique has a number of offshoots that are perhaps of more practical use.

a) You can store two programs in memory simultaneously for the purpose of appending one to the other. This technique requires that the line numbers of the two programs do not overlap. (See *Programming the PET/CBM* by Raeto Collin West, pages 41-42, for a discussion of this technique.)

b) You can have two programs in memory at once and use the concept in (2) above to allow an easier way to create a *safe* area in low memory. The first program is just one line that sets the start of BASIC pointer to the address of the second program which is located higher in memory, and then runs that second program.

4. Since this address is used as the address of the first byte to SAVE, you can save any section of memory by changing this pointer to indicate the starting address, and the pointer 45–46 ($2D–$2E) to indicate the address of the byte after the last byte that you wish to save.

45-46 $2D-$2E VARTAB

Pointer to the Start of the BASIC Variable Storage Area

This location points to the address which marks the end of the BASIC program text area, and the beginning of the variable storage area. All nonarray variables are stored here, as are string descriptors (for the address of the area where the actual text of strings is stored, see location 51, $33).

Seven bytes of memory are allocated for each variable. The first two bytes are used for the variable name, which consists of the ASCII value of the first two letters of the variable name. If the variable name is a single letter, the second byte will contain a zero.

The seventh bit of one or both of these bytes can be set (which would add 128 to the ASCII value of the letter). This indicates the variable type. If neither byte has the seventh bit set, the variable is the regular floating point type. If only the second byte has its seventh bit set, the variable is a string. If only the first byte has its seventh bit set, the variable is a defined function (FN). If both bytes have the seventh bit set, the variable is an integer.

The use of the other five bytes depends on the type of variable. A floating point variable will use the five bytes to store the value of the variable in floating point format. An integer will have its value

stored in the third and fourth bytes, high byte first, and the other three will be unused.

A string variable will use the third byte for its length, and the fourth and fifth bytes for a pointer to the address of the string text, leaving the last two bytes unused. Note that the actual string text that is pointed to is located either in the part of the BASIC program where the string is first assigned a value, or in the string text storage area pointed to by location 51 ($33).

A function definition will use the third and fourth bytes for a pointer to the address in the BASIC program text where the function definition starts. It uses the fifth and sixth bytes for a pointer to the dependent variable (the X of FN A (X)). The final byte is not used.

Knowing something about how variables are created can help your BASIC programming. For example, you can see that nonarray integer variables take up no less space than floating point variables, and since most BASIC commands convert the integers to floating point, they do not offer a speed advantage either, and in many cases will actually slow the program down. As will be seen below, however, integer arrays can save a considerable amount of space.

Variables are stored in the order in which they are created. Likewise, when BASIC goes looking for a variable, it starts its search at the beginning of this area. If commonly used variables are defined at the end of the program, and are thus at the back of this area, it will take longer to find them. It may help program execution speed to define the variables that will be used most frequently right at the beginning of the program.

Also, remember that once created, variables do not go away during program execution. Even if they are never used again, they still take up space in the variable storage area, and they slow down the routine that is used to search for variables that are referenced.

Another point to consider about the order in which to define variables is that arrays are created in a separate area of memory which starts at the end of the nonarray variable area. Therefore, every time a nonarray variable is created, all of the arrays must be moved seven bytes higher in memory in order to make room for the new variable. Therefore, it may help performance to avoid defining nonarray variables after defining arrays.

This pointer will be reset to one byte past the end of the BASIC program text whenever you execute the statements CLR, NEW, RUN, or LOAD. Adding or modifying a BASIC statement will have the same effect, because the higher numbered BASIC statements have to be moved up into memory to make room for the new statements, and can therefore overwrite the variable storage area. This means that if you wish to check the value of a variable after stopping a program, you can only do so *before* modifying the program.

The exception to the above is when the LOAD command is

issued from a program. The purpose of not resetting this pointer in such a case is to allow the chaining of programs by having one program load and run the next (that is also why a LOAD issued from a program causes a RUN from the beginning of the program). This allows the second program to share variables with the first. There are problems with this, however. Some string variable descriptors and function definitions have their pointers set to areas within the program text. When this text is replaced by a load, these pointers are no longer valid, which will lead to errors if the FN or string value is referenced. And if the second program text area is larger than that of the first, the second program will overwrite some of the first program's variables, and their values will be lost.

The ability to chain short programs is a holdover from the days of the 8K PET, for which this BASIC was written, but with the vastly increased memory of the 64, program chaining should not be necessary.

You should also note that SAVE uses this pointer as the address of the byte after the last byte to SAVE.

47-48 $2F-$30 ARYTAB

Pointer to the Start of the BASIC Array Storage Area

This location points to the address of the end of nonarray variable storage, and the beginning of array variable storage. The format for array storage is as follows:

The first two bytes hold the array name. The format and high-bit patterns are the same as for nonarray variables (see 45, $2D above), except that there is no equivalent to the function definition.

Next comes a two-byte offset to the start of the next array, low byte first. Then there is a one-byte value for the number of array dimensions (e.g., 2 for a two-dimensional array like A(x,y)). That byte is followed by pairs of bytes which hold the value of each array dimension + 1 (DIMensioning an array always makes space for 0, so A(0) can and should be used).

Finally come the values of the variables themselves. The format for these values is the same as with nonarray values, but each value only takes up the space required; that is, floating point variables use five bytes each, integers two bytes, and strings descriptors three bytes each.

Remember that as with nonarray strings, the actual string text is stored elsewhere, in the area which starts at the location pointed to in 51-52 ($33–$34).

49-50 $31-$32 STREND

Pointer to End of the BASIC Array Storage Area (+1), and the Start of Free RAM

This location points to the address of the end of BASIC array storage

space and the start of free RAM. Since string text starts at the top of memory and builds downward, this location can also be thought of as the last possible address of the string storage area. Defining new variables pushes this pointer upward, toward the last string text.

If a string for which space is being allocated would cross over this boundary into the array storage area, garbage collection is performed, and if there still is not enough room, an OUT OF MEMORY error occurs. FRE performs garbage collection, and returns the difference between the address pointed to here and the address of the end of string text storage pointed to by location 51 ($33).

51-52 $33-$34 FRETOP

Pointer to the Bottom of the String Text Storage Area

This pointer marks the current end of the string text area, and the top of free RAM (strings are built from the top of memory downward). Additional string texts are added to the area below the address pointed to here. After they are added, this pointer is lowered to point below the newly added string text. The garbage collection routine (which is also called by FRE) readjusts this pointer upward.

While the power-on/reset routines set this pointer to the top of RAM, the CLR command sets this pointer to the end of BASIC memory, as indicated in location 55 ($37). This allows the user to set aside an area of BASIC memory that will not be disturbed by the program, as detailed at location 55 ($37).

53-54 $35-$36 FRESPC

Temporary Pointer for Strings

This is used as a temporary pointer to the most current string added by the routines which build strings or move them in memory.

55-56 $37-$38 MEMSIZ

Pointer to the Highest Address Used by BASIC

The power-on/reset routine tests each byte of RAM until it comes to the BASIC ROM, and sets this pointer to the address of the highest byte of consecutive RAM found (40959, $9FFF).

There are two circumstances under which this pointer may be changed after power-up to reflect an address lower than the actual top of consecutive RAM:

1. Users may wish to lower this pointer themselves, in order to set aside an area of free RAM that will not be disturbed by BASIC. For example, to set aside a 1K area at the top of BASIC, start your program with the line:

```
POKE 56, PEEK(56)-4:CLR
```

The CLR is necessary to insure that the string text will start below your safe area.

You may wish to store machine language programs, sprites, or alternate character sets in such an area. For the latter two applications, however, keep in mind the 16K addressing range limitation of the VIC-II chip. If you do not assign the VIC-II to a bank other than the default memory bank of 0-16383 ($0-$3FFF), you must lower the top of memory below 16383 ($3FFF) if you wish your sprite or character data area to be within its addressing range.

2. When the RS-232 device (number 2) is opened, this pointer and the pointer to the end of user RAM at 643 are lowered by 512 bytes in order to create two 256-byte buffers, one for input and the other for output.

Since the contents of these buffers will overwrite any variables at the top of memory, a CLR command is issued at the time device 2 is opened. Therefore, the RS-232 device should be opened before defining any variables, and before setting aside a safe area for machine language programs or other uses, as described above.

57-58 $39-$3A CURLIN

Current BASIC Line Number

This location contains the line number of the BASIC statement which is currently being executed, in LSB/MSB format. A value of 255 ($FF) in location 58 ($3A), which translates to a line number of 65280 or above (well over the 63999 limit for a program line), means that BASIC is currently in immediate mode, rather than RUN mode.

BASIC keywords that are illegal in direct mode check 58 ($3A) to determine whether or not this is the current mode.

When in RUN mode, this location is updated as each new BASIC line is fetched for execution. Therefore, a TRACE function could be added by diverting the vector at 776 ($308), which points to the routine that executes the next token, to a user-written routine which prints the line number indicated by this location before jumping to the token execution routine. (LISTing the line itself would be somewhat harder, because LIST uses many Page 0 locations that would have to be preserved and restored afterwards.)

This line number is used by BREAK and error messages to show where program execution stopped. The value here is copied to 59 ($3B) by STOP, END and the STOP-key BREAK, and copied back by CONT.

59-60 $3B-$3C OLDLIN

Previous BASIC Line Number

When program execution ends, the last line number executed is stored here, and restored to location 57 ($39) by CONT.

61-62 $3D-$3E OLDTXT

Pointer to the Address of the Current BASIC Statement

This location contains the address (not the line number) of the text of the BASIC statement that is being executed. The value of TXTPTR (122, $7A), the pointer to the address of the BASIC text character currently being scanned, is stored here each time a new BASIC line begins execution.

END, STOP, and the STOP-key BREAK save the value of TXTPTR here, and CONT restores this value to TXTPTR. CONT will not continue if 62 ($3E) has been changed to a zero by a LOAD, a modification to the program text, or by error routines.

63-64 $3F-$40 DATLIN

Current DATA Line Number

This location holds the line number of the current DATA statement being READ. It should be noted that this information is not used to determine where the next DATA item is read from (that is the job of the pointer at 65-66 ($41-$42) below). But if an error concerning the DATA occurs, this number will be moved to 57 ($39), so that the error message will show that the error occurred in the line that contains the DATA statement, rather than in the line that contains the READ statement.

65-66 $41-$42 DATPTR

Pointer to the Address of the Current DATA Item

This location points to the address (not the line number) within the BASIC program text area where DATA is currently being READ. RESTORE sets this pointer back to the address indicated by the start of BASIC pointer at location 43 ($2B).

The sample program below shows how the order in which DATA statements are READ can be changed using this pointer. The current address of the statement before the DATA statement is stored in a variable, and then used to change this pointer.

```
10 A1=PEEK(61):A2=PEEK(62)
20 DATA THIS DATA WILL BE USED SECOND
30 B1=PEEK(61):B2=PEEK(62)
40 DATA THIS DATA WILL BE USED FIRST
50 C1=PEEK(61):C2=PEEK(62)
60 DATA THIS DATA WILL BE USED THIRD
70 POKE 65,B1:POKE 66,B2:READ A$:PRINT A$
80 POKE 65,A1:POKE 66,A2:READ A$:PRINT A$
90 POKE 65,C1:POKE 66,C2:READ A$:PRINT A$
```

67-68 $43-$44 INPPTR

Pointer to the Source of GET, READ, or INPUT Information

READ, INPUT, and GET all use this as a pointer to the address of the source of incoming data, such as DATA statements, or the text input buffer at 512 ($200).

69-70 $45-$46 VARNAM

Current BASIC Variable Name

The current variable name being searched for is stored here, in the same two-byte format as in the variable value storage area located at the address pointed to by 45 ($2D). See that location for an explanation of the format.

71-72 $47-$48 VARPNT

Pointer to the Current BASIC Variable Value

This location points to the address of the descriptor of the current BASIC variable (see location 45 ($2D) for the format of a variable descriptor). Specifically, it points to the byte just after the two-character variable name.

During a FN call, this location does not point to the dependent variable (the A of FN A), so that a real variable of the same name will not have its value changed by the call.

73-74 $49-$4A FORPNT

Temporary Pointer to the Index Variable Used by FOR

The address of the BASIC variable which is the subject of a FOR/NEXT loop is first stored here, but is then pushed onto the stack. That leaves this location free to be used as a work area by such statements as INPUT, GET, READ, LIST, WAIT, CLOSE, LOAD, SAVE, RETURN, and GOSUB.

For a description of the stack entries made by FOR, see location 256 ($100).

75-76 $4B-$4C OPPTR

Math Operator Table Displacement

This location is used during the evaluation of mathematical expressions to hold the displacement of the current math operator in an operator table. It is also used as a save area for the pointer to the address of program text which is currently being read.

77 $4D OPMASK

Mask for Comparison Operation

The expression evaluation routine creates a mask here which lets it know whether the current comparison operation is a less-than (1), equals (2), or greater-than comparison.

78-79 $4E-$4F DEFPNT

Pointer to the Current FN Descriptor

During function definition (DEF FN) this location is used as a pointer to the descriptor that is created. During function execution (FN) it points to the FN descriptor in which the evaluation results should be saved.

80-82 $50-$52 DSCPNT

Temporary Pointer to the Current String Descriptor

The string assignment and handling routines use the first two bytes as a temporary pointer to the current string descriptor, and the third to hold the value of the string length.

83 $53 FOUR6

Constant for Garbage Collection

The constant contained here lets the garbage collection routines know whether a three- or seven-byte string descriptor is being collected.

84-86 $54-$56 JMPER

Jump to Function Instruction

The first byte is the 6502 JMP instruction ($4C), followed by the address of the required function taken from the table at 41042 ($A052).

87-96 $57-$60

BASIC Numeric Work Area

This is a very busy work area, used by many routines.

97-102 $61-$66 FAC1

Floating Point Accumulator #1

The Floating Point Accumulator is central to the execution of any BASIC mathematical operation. It is used in the conversion of integers to floating point numbers, strings to floating point numbers, and vice versa. The results of most evaluations are stored in this location.

The internal format of floating point numbers is not particularly easy to understand (or explain). Generally speaking, the number is broken into the normalized mantissa, which represents a number between 1 and 1.99999..., and an exponent value, which represents a power of 2. Multiplying the mantissa by 2 raised to the value of the exponent gives you the value of the floating point number.

Fortunately, the BASIC interpreter contains many routines for the manipulation and conversion of floating point numbers, and these routines can be called by the user. See the entries for locations 3 and 5.

Floating Point Accumulator #1 can be further divided into the following locations:

97 $61 FACEXP

Floating Point Accumulator #1: Exponent

This exponent represents the closest power of two to the number, with 129 added to take care of the sign problem for negative exponents. An exponent of 128 is used for the value 0; an exponent of 129 represents 2 to the 0 power, or 1; an exponent of 130 represents 2 to the first power, or 2; 131 is 2 squared, or 4; 132 is 2 cubed, or 8; and so on.

98-101 $62-$65 FACHO

Floating Point Accumulator #1: Mantissa

The most significant digit can be assumed to be a 1 (remember that the range of the mantissa is from 1 to 1.99999...) when a floating point number is stored to a variable. The first bit is used for the sign of the number, and the other 31 bits of the four-byte mantissa hold the other significant digits.

The first two bytes (98-99, $62-$63) of this location will hold the signed integer result of a floating point to integer conversion, in high-byte, low-byte order.

102 $66 FACSGN

Floating Point Accumulator #1: Sign

A value of 0 here indicates a positive number, while a value of 255 ($FF) indicates a negative number.

103 $67 SGNFLG

Number of Terms in a Series Evaluation

This location is used by mathematical formula evaluation routines. It indicates the number of separate evaluations that must be done to resolve a complex expression down to a single term.

104 $68 BITS

Floating Point Accumulator #1: Overflow Digit

This location contains the overflow byte. The overflow byte is used in an intermediate step of conversion from an integer or text string to a floating point number.

105-110 $69-$6E FAC2

Floating Point Accumulator #2

A second Floating Point Accumulator, used in conjunction with Floating Point Accumulator #1 in the evaluation of products, sums, differences—in short, any operation requiring more than one value. The format of this accumulator is the same as FAC1.

105 $69 ARGEXP

Floating Point Accumulator #2: Exponent

106-109 $6A-$6D ARGHO

Floating Point Accumulator #2: Mantissa

110 $6E ARGSGN

Floating Point Accumulator #2: Sign

111 $6F ARISGN

Result of a Sign Comparison of Accumulator #1 to Accumulator #2

Used to indicate whether the two Floating Point Accumulators have like or unlike signs. A 0 indicates like signs, a 255 ($FF) indicates unlike signs.

112 $70 FACOV

Low Order Mantissa Byte of Floating Point Accumulator #1 (For Rounding)

If the mantissa of the floating point number has more significant figures than can be held in four bytes, the least significant figures are placed here. They are used to extend the accuracy of intermediate mathematical operations and to round the final figure.

113-114 $71-$72 FBUFPT

Series Evaluation Pointer

This location points to the address of a temporary table of values built in the free RAM area for the evaluation of formulas. It is also used for such various purposes as a TI$ work area, string setup pointer, and work space for the determination of the size of an array.

Although this is labeled a pointer to the tape buffer in the *Programmer's Reference Guide,* disassembly of the BASIC ROM reveals no reference to this location for that purpose (see 178, $B2 for pointer to tape buffer).

115-138 $73-$8A CHRGET

Subroutine: Get Next BASIC Text Character

This is actually a machine language subroutine, which at the time of a BASIC cold start (such as when the power is turned on) is copied from MOVCHG (58274, $E3A2) in the ROM to this zero page location.

CHRGET is a crucial routine which BASIC uses to read text characters, such as the text of the BASIC program which is being interpreted. It is placed on zero page to make the routine run faster. Since it keeps track of the address of the character being read within

the routine itself, the routine must be in RAM in order to update that pointer. The pointer to the address of the byte currently being read is really the operand of a LDA instruction. When entered from CHRGET, the routine increments the pointer by modifying the operand at TXTPTR (122, $7A), thus allowing the next character to be read.

Entry at CHRGOT (121, $79) allows the current character to be read again. The CHRGET routine skips spaces, sets the various flags of the status register (.P) to indicate whether the character read was a digit, statement terminator, or other type of character, and returns with the retrieved character in the Accumulator (.A).

Since CHRGET is used to read every BASIC statement before it is executed, and since it is in RAM, and therefore changeable, it makes a handy place to intercept BASIC to add new features and commands (and in the older PET line, it was the only way to add such features). Diversion of the CHRGET routine for this purpose is generally referred to as a wedge.

Since a wedge can greatly slow down execution speed, most of the time it is set up so that it performs its preprocessing functions only when in direct or immediate mode. The most well-known example of such a wedge is the "Universal DOS Support" program that allows easier communication with the disk drive command channel.

As this is such a central routine, a disassembly listing is given below to provide a better understanding of how it works.

```
115 $73     CHRGET  INC TXTPTR     ; increment low byte of TXTPTR
117 $75             BNE CHRGOT     ; if low byte isn't 0, skip next
119 $77             INC TXTPTR+1   ; increment high byte of TXTPTR
121 $79             CHRGOT LDA     ; load byte from where TXTPTR points
                                   ; entry here does not update TXTPTR,
                                   ; allowing you to read the old byte again
122 $7A     TXTPTR  $0207          ; pointer is really the LDA operand
                                   ; TXTPTR+1 points to 512-580 ($200-
                                   ; $250) when reading from the input buffer
                                   ; in direct mode.
124 $7C     POINTB  CMP #$3A       ; carry flag set if >ASCII numeral 9
126 $7E             BCS EXIT       ; character is not a numeral--exit
128 $80             CMP #$20       ; if it is an ASCII space...
130 $82             BEQ CHRGET     ; ignore it and get next character
132 $84             SEC            ; prepare to subtract
133 $85             SBC #$30       ; ASCII 0-9 are between 48-57 ($30-39)
135 $87             SEC            ; prepare to subtract again
136 $88             SBC #$D0       ; if <ASCII 0 (57, $39) then carry is set
138 $8A     EXIT    RTS            ; carry is clear only for numeral on return
```

The Accumulator (.A register) holds the character that was read on exit from the routine. Status register (.P) bits which can be tested for on exit are:

Carry Clear if the character was an ASCII digit 0-9; Carry Set, otherwise. Zero Set only if the character was a statement terminator 0 or an ASCII colon, 58 ($3A). Otherwise, Zero Clear.

One wedge insertion technique is to change CHRGET's INC $7A to a JMP WEDGE, have your wedge update TXTPTR itself, and then JSR CHRGOT. Another is to change the CMP #$3A at location 124 ($7C), which I have labeled POINTB, to a JMP WEDGE, do your wedge processing, and then exit through the ROM version of POINTB, which is located at 58283 ($E3AB). For more detailed information about wedges, see *Programming the PET/CBM*, Raeto Collin West, pages 365-68.

While the wedge is a good, quick technique for adding new commands, a much more elegant method exists for accomplishing this task on the VIC-20 and 64 without slowing BASIC down to the extent that the wedge does. See the entries for the BASIC RAM vector area at 768-779 ($300-$30B) for more details.

139-143 $8B-$8F RNDX

RND Function Seed Value

This location holds the five-byte floating point value returned by the RND function. It is initially set to a seed value copied from ROM (the five bytes are 128, 79, 199, 82, 88—$80, $4F, $C7, $52, $58).

When the function RND(X) is called, the numeric value of X does not affect the number returned, but its sign does. If X is equal to 0, RND generates a seed value from chip-level hardware timers. If X is a positive number, RND(X) will return the next number in an arithmetic sequence. This sequence continues for such a long time without repeating itself, and gives such an even distribution of numbers, that it can be considered random. If X is negative, the seed value is changed to a number that corresponds to a scrambled floating point representation of the number X itself.

Given a particular seed value, the same pseudorandom series of numbers will always be returned. This can be handy for debugging purposes, but not where you wish to have truly random numbers.

The traditional Commodore method of selecting a random seed is by using the expression RND (-TI), mostly because RND(0) didn't function correctly on early PETs. While the RND(0) form doesn't really work right on the 64 either (see location 57495, $E097), the expression RND(-RND(0)) may produce a more random seed value.

Location Range: 144-255 ($90-$FF)

Kernal Work Storage Area

This is the zero-page storage area for the Kernal. The user should take into account what effect changing a location here will have on the operation of Kernal functions before making any such changes.

At power-on, this range of locations is first filled with zeros, and then initialized from values stored in ROM as needed.

144 $90 STATUS

Kernal I/O Status Word (ST)

The Kernal routines which open I/O channels or perform input/output functions check and update this location. The value here is almost always the same as that returned to BASIC by use of the reserved variable ST. Note that BASIC syntax will not allow an assignment such as ST=4. A table of status codes for cassette and serial devices follows below.

Cassette:
Bit 2 (Bit Value of 4) = Short Block
Bit 3 (Bit Value of 8) = Long Block
Bit 4 (Bit Value of 16) = Unrecoverable error (Read), mismatch
Bit 5 (Bit Value of 32) = Checksum error
Bit 6 (Bit Value of 64) = End of file
Bit 7 (Bit Value of 128) = End of tape

Serial Devices:
Bit 0 (Bit Value of 1) = Time out (Write)
Bit 1 (Bit Value of 2) = Time out (Read)
Bit 6 (Bit Value of 64) = EOI (End or Identify)
Bit 7 (Bit Value of 128) = Device not present

Probably the most useful bit to test is Bit 6 (end of file). When using the GET statement to read in individual bytes from a file, the statement IF ST AND 64 will be true if you have got to the end of the file.

For status codes for the RS-232 device, see the entry for location 663 ($297).

145 $91 STKEY

Flag: Was STOP Key Pressed?

This location is updated every 1/60 second during the execution of the IRQ routine that reads the keyboard and updates the jiffy clock.

The value of the last row of the keyboard matrix is placed here. That row contains the STOP key, and although this location is used primarily to detect when that key has been pressed, it can also detect when any of the other keys in that row of the matrix have been pressed.

In reading the keyboard matrix, a bit set to 1 means that no key has been pressed, while a bit reset to 0 indicates that a key is pressed. Therefore, the following values indicate the keystrokes detailed below:

255 $FF = no key pressed
254 $FE = 1 key pressed
253 $FD = ← key pressed
251 $FB = **CTRL** key pressed

247 $F7 = **2** key pressed
239 $EF = **space bar** pressed
223 $DF = **Commodore logo** key pressed
191 $BF = **Q** key pressed
127 $7F = **STOP** key pressed

VIC owners will notice that the 64's keyboard matrix is very different from the VIC's. One of the advantages of this difference is that you can test for the STOP key by following a read of this location with a BPL instruction, which will cause a branch to occur anytime that the STOP key is pressed.

146 $92 SVXT

Timing Constant for Tape Reads

This location is used as an adjustable timing constant for tape reads, which can be changed to allow for the slight speed variations between tapes.

147 $93 VERCK

Flag for Load Routine: 0=LOAD, 1=VERIFY

The same Kernal routine can perform either a LOAD or VERIFY, depending on the value stored in the Accumulator (.A) on entry to the routine. This location is used to determine which operation to perform.

148 $94 C3PO

Flag: Serial Bus—Output Character Was Buffered

This location is used by the serial output routines to indicate that a character has been placed in the output buffer and is waiting to be sent.

149 $95 BSOUR

Buffered Character for Serial Bus

This is the character waiting to be sent. A 255 ($FF) indicates that no character is waiting for serial output.

150 $96 SYNO

Cassette Block Synchronization Number

151 $97 XSAV

Temporary .X Register Save Area

This .X register save area is used by the routines that get and put an ASCII character.

152 $98 LDTND

Number of Open I/O Files/Index to the End of File Tables

The number of currently open I/O files is stored here. The maximum number that can be open at one time is ten. The number stored here is used as the index to the end of the tables that hold the file numbers, device numbers, and secondary address numbers (see locations 601–631, $259–$277, for more information about these tables).

CLOSE decreases this number and removes entries from the tables referred to above, while OPEN increases it and adds the appropriate information to the end of the tables. The Kernal routine CLALL closes all files by setting this number to 0, which effrectively empties the tables.

153 $99 DFLTN

Default Input Device (Set to 0 for the Keyboard)

The default value of this location is 0, which designates the keyboard as the current input device. That value can be changed by the Kernal routine CHKIN (61966, $F20E), which uses this location to store the device number of the device whose file it defines as an input channel.

BASIC calls CHKIN whenever the command INPUT# or GET# is executed, but clears the channel after the input operation has been completed.

154 $9A DFLTO

Default Output (CMD) Device (Set to 3 for the Screen)

The default value of this location is 3, which designates the screen as the current output device. That value can be changed by the Kernal routine CHKOUT (62032, $F250), which uses this location to store the device number of the device whose file it defines as an output channel.

BASIC calls CHKOUT whenever the command PRINT# or CMD is executed, but clears the channel after the PRINT# operation has been completed.

155 $9B PRTY

Tape Character Parity

This location is used to help detect when bits of information have been lost during transmission of tape data.

156 $9C DPSW

Flag: Tape Byte Received

This location is used as a flag to indicate whether a complete byte of tape data has been received, or whether it has only been partially received.

157 $9D MSGFLG

Flag: Kernal Message Control

This flag is set by the Kernal routine SETMSG (65048, $FE18), and it controls whether or not Kernal error messages or control messages will be displayed.

A value of 192 ($C0) here means that both Kernal error and control messages will be displayed. This will never normally occur when using BASIC, which prefers its own plain text error messages over the Kernal's perfunctory I/O ERROR (number). The Kernal error messages might be used, however, when you are SAVEing or LOADing with a machine language monitor.

A 128 ($80) means that control messages only will be displayed. Such will be the case when you are in the BASIC direct or immediate mode. These messages include SEARCHING, SAVING, FOUND, etc.

A value of 64 means that Kernal error messages only are on. A 0 here suppresses the display of all Kernal messages. This is the value placed here when BASIC enters the program or RUN mode.

158 $9E PTR1

Tape Pass 1 Error Log Index

This location is used in setting up an error log of bytes in which transmission parity errors occur the first time that the block is received (each tape block is sent twice to minimize data loss from transmission error).

159 $9F PTR2

Tape Pass 2 Error Log Correction Index

This location is used in correcting bytes of tape data which were transmitted incorrectly on the first pass.

160-162 $A0-$A2 TIME

Software Jiffy Clock

These three locations are updated 60 times a second, and serve as a software clock which counts the number of jiffies (sixtieths of a second) that have elapsed since the computer was turned on.

The value of location 162 ($A2) is increased every jiffy (.01667 second), 161 ($A1) is updated every 256 jiffies (4.2267 seconds), and 160 ($A0) changes every 65536 jiffies (or every 18.2044 minutes).

After 24 hours, these locations are set back to 0.

The jiffy clock is used by the BASIC reserved variables TI and TI$. These are not ordinary variables that are stored in the RAM variable area, but are functions that call the Kernal routines RDTIM (63197, $F6DD) and SETTIM (63204, $F6E4). Assigning the value of TI or TI$ to another variable reads these locations, while assigning a given value to TI$ alters these locations.

To illustrate the relationship between these locations and TI$, try the following program. The program sets the jiffy clock to 23 hours, 59 minutes. After the program has been running for one minute, all these locations will be reset to 0.

```
100 TI$="235900"
110 PRINT TI$,PEEK(160),PEEK(161),PEEK(162)
120 GOTO 110
```

Since updating is done by the IRQ interrupt that reads the keyboard, anything which affects the operation of that interrupt routine will also interfere with this clock. A typical example is tape I/O operations, which steal the IRQ vector for their own use, and restore it afterwards. Obviously, user routines which redirect the IRQ and do not send it back to the normal routine will upset software clock operation as well.

163-164 $A3-$A4

Temporary Data Storage Area

These locations are used temporarily by the tape and serial I/O routines.

165 $A5 CNTDN

Cassette Synchronization Character Countdown

Used to count down the number of synchronization characters that are sent before the actual data in a tape block.

166 $A6 BUFPNT

Count of Characters in Tape I/O Buffer

This location is used to count the number of bytes that have been read in or written to the tape buffer. Since on a tape write, no data is sent until the 192 byte buffer is full, you can force output of the buffer with the statement POKE 166,191.

167 $A7 INBIT

RS-232 Input Bits/Cassette Temporary Storage Area

This location is used to temporarily store each bit of serial data that is received, as well as for miscellaneous tasks by tape I/O.

168 $A8 BITCI

RS-232 Input Bit Count/Cassette Temporary Storage

This location is used to count the number of bits of serial data that has been received. This is necessary so that the serial routines will know when a full word has been received. It is also used as an error flag during tape loads.

169 $A9 RINONE

RS-232 Flag: Check for Start Bit

This flag is used when checking for a start bit. A 144 ($90) here indicates that no start bit was received, while a 0 means that a start bit was received.

170 $AA RIDATA

RS-232 Input Byte Buffer/Cassette Temporary Storage

Serial routines use this area to reassemble the bits received into a byte that will be stored in the receiving buffer pointed to by 247 ($F7). Tape routines use this as a flag to help determine whether a received character should be treated as data or as a synchronization character.

171 $AB RIPRTY

RS-232 Input Parity/Cassette Leader Counter

This location is used to help detect if data was lost during RS-232 transmission, or if a tape leader is completed.

172-173 $AC-$AD SAL

Pointer to the Starting Address of a Load/Screen Scrolling

The pointer to the start of the RAM area to be SAVEd or LOADed at 193 ($C1) is copied here. This pointer is used as a working version, to be increased as the data is received or transmitted. At the end of the operation, the initial value is restored here. Screen management routines temporarily use this as a work pointer.

174-175 $AE-$AF EAL

Pointer to Ending Address of Load (End of Program)

This location is set by the Kernal routine SAVE to point to the ending address for SAVE, LOAD, or VERIFY.

176-177 $B0-$B1 CMP0

Tape Timing

Location 176 ($B0) is used to determine the value of the adjustable timing constant at 146 ($92). Location 177 is also used in the timing of tape reads.

178-179 $B2-$B3 TAPE1

Pointer: Start of Tape Buffer

On power-on, this pointer is set to the address of the cassette buffer (828, $33C). This pointer must contain an address greater than or equal to 512 ($200), or an ILLEGAL DEVICE NUMBER error will be sent when tape I/O is tried.

180 $B4 BITTS

RS-232 Output Bit Count/Cassette Temporary Storage

RS-232 routines use this to count the number of bits transmitted, and for parity and stop bit manipulation. Tape load routines use this location to flag when they are ready to receive data bytes.

181 $B5 NXTBIT

RS-232 Next Bit to Send/Tape EOT Flag

This location is used by the RS-232 routines to hold the next bit to be sent, and by the tape routines to indicate what part of a block the read routine is currently reading.

182 $B6 RODATA

RS-232 Output Byte Buffer

RS-232 routines use this area to disassemble each byte to be sent from the transmission buffer pointed to by 249 ($F9).

183 $B7 FNLEN

Length of Current Filename

This location holds the number of characters in the current filename. Disk filenames may have from 1 to 16 characters, while tape filenames range from 0 to 187 characters in length.

If the tape name is longer than 16 characters, the excess will be truncated by the SEARCHING and FOUND messages, but will still be present on the tape. This means that machine language programs meant to run in the cassette buffer may be saved as tape filenames.

A disk file is always referred to by a name, whether full or generic (containing the wildcard characters * or ?). This location will always be greater than 0 if the current file is a disk file. Tape LOAD, SAVE, and VERIFY operations do not require that a name be specified, and this location can therefore contain a 0. If this is the case, the contents of the pointer to the filename at 187 will be irrelevant.

An RS-232 OPEN command may specify a filename of up to four characters. These characters are copied to locations 659-662 ($293-$296), and determine baud rate, word length, and parity.

184 $B8 LA

Current Logical File Number

This location holds the logical file number of the device currently being used. A maximum of five disk files, and ten files in total, may be open at any one time.

File numbers range from 1 to 255 (a 0 is used to indicate system defaults). When printing to a device with a file number greater than 127, an ASCII linefeed character will be sent following each carriage return, which is useful for devices like serial printers that require linefeeds in addition to carriage returns.

The BASIC OPEN command calls the Kernal OPEN routine, which sets the value of this location. In the BASIC statement OPEN 4,8,15, the logical file number corresponds to the first parameter, 4.

185 $B9 SA

Current Secondary Address

This location holds the secondary address of the device currently being used. The range of valid secondary address numbers is 0 through 31 for serial devices, and 0 through 127 for other devices.

Secondary device numbers mean something different to each device that they are used with. The keyboard and screen devices ignore the secondary address completely. But any device which can have more than one file open at the same time, such as the disk drive, distinguishes between these files by using the secondary address. Therefore, it is necessary to specify a secondary address when opening a disk file. Secondary address numbers 0, 1, and 15-31 have a special significance to the disk drive, and therefore device numbers 2-14 only should be used as secondary addresses when opening a disk file.

OPENing a disk file with a secondary address of 15 enables the user to communicate with the Disk Operating System through that channel. A LOAD command which specifies a secondary address of 0 (for example, LOAD "AT BASIC", 8,0) results in the program being loaded not to the address specified on the file as the starting address, but rather to the address pointed to by the start of BASIC pointer (43, $2B).

A LOAD with a secondary address of 1 (for example, LOAD "HERE", 8,1) results in the contents of the file being loaded to the address specified in the file. A disk file that has been LOADed using a secondary address of 1 can be successfully SAVEd in the same manner (SAVE "DOS 5.1", 8,1).

LOADs and SAVEs that do not specify a secondary address will default to a secondary address of 0.

When OPENing a Datassette recorder file, a secondary address of 0 signifies that the file will be read, while a secondary address of 1 signifies that the file will be written to. A value of 2 can be added

to indicate that an End of Tape marker should be written as well. This marker tells the Datassette not to search past it for any more files on the tape, though more files can be written to the tape if desired.

As with the disk drive, the LOAD and SAVE commands use secondary addresses of 0 and 1 respectively to indicate whether the operation should be relocating or nonrelocating.

When the 1515 or 1525 Printer is opened with a secondary address of 7, the uppercase/lowercase character set is used. If it is opened with a secondary address of 0, or without a secondary address, the uppercase/graphics character set will be used.

186 $BA FA

Current Device Number

This location holds the number of the device that is currently being used. Device number assignments are as follows:

0=Keyboard; 1=Datassette Recorder; 2=RS-232/User Port; 3=Screen; 4-5=Printer, 8-11=Disk.

187-188 $BB-$BC FNADR

Pointer: Current Filename

This location holds a pointer to the address of the current filename. If an operation which OPENs a tape file does not specify a filename, this pointer is not used.

When a disk filename contains a shifted space character, the remainder of the name will appear outside the quotes in the directory, and may be used for comments. For example, if you SAVE "ML[shifted space]SYS 828", the directory entry will read "ML"SYS 828. You may reference the program either by the portion of the name that appears within quotes, or by the full name, including the shifted space. A program appearing later in the directory as "ML"SYS 900 would not be found just by reference to "ML", however.

A filename of up to four characters may be used when opening the RS-232 device. These four characters will be copied to 659-662 ($293-$296), where they are used to control such parameters as baud rate, parity, and word length.

189 $BD ROPRTY

RS-232 Output Parity/Cassette Temporary Storage

This location is used by the RS-232 routines as an output parity work byte, and by the tape as temporary storage for the current character being read or sent.

190 $BE FSBLK

Cassette Read/Write Block Count

Used by the tape routines to count the number of copies of a data block remaining to be read or written.

191 $BF MYCH

Tape Input Byte Buffer

This is used by the tape routines as a work area in which incoming characters are assembled.

192 $C0 CAS1

Tape Motor Interlock

This location is maintained by the IRQ interrupt routine that scans the keyboard. Whenever a button is pressed on the recorder, this location is checked. If it contains a 0, the motor is turned on by setting Bit 5 of location 1 to 0. When the button is let up, the tape motor is turned off, and this location is set to 0.

Since the interrupt routine is executed 60 times per second, you will not be able to keep the motor bit set to keep the motor on if no buttons are pushed. Likewise, if you try to turn the motor off when a button is pressed and this location is set to 0, the interrupt routine will turn it back on.

To control the motor via software, you must set this location to a nonzero value after one of the buttons on the recorder has been pressed.

193-194 $C1-$C2 STAL

I/O Start Address

This location points to the beginning address of the area in RAM which is currently being LOADed or SAVEd. For tape I/O, it will point to the cassette buffer, which is used for the first block, while the rest of the I/O operation uses the area of RAM pointed to by location 195 ($C3).

195-196 $C3-$C4 MEMUSS

Tape Load Temporary Addresses

During a tape LOAD or SAVE, the first block, which contains the header, is loaded to or from the cassette buffer, and the rest of the data is LOADed or SAVEd directly to or from RAM. This location points to the beginning address of the area of RAM to be used for the blocks of data that come after the initial header.

197 $C5 LSTX

Matrix Coordinate of Last Key Pressed, 64=None Pressed

During every normal IRQ interrupt this location is set with the value

of the last keypress, to be used in keyboard debouncing. The Operating System can check if the current keypress is the same as the last one, and will not repeat the character if it is.

The value returned here is based on the keyboard matrix values as set forth in the explanation of location 56320 ($DC00). The values returned for each key pressed are shown at the entry for location 203 ($CB).

198 $C6 NDX

Number of Characters in Keyboard Buffer (Queue)

The value here indicates the number of characters waiting in the keyboard buffer at 631 ($277). The maximum number of characters in the keyboard buffer at any one time is determined by the value in location 649 ($289), which defaults to 10.

If INPUT or GET is executed while there are already characters in the buffer, those characters will be read as part of the data stream. You can prevent this by POKEing a 0 to this location before those operations, which will always cause any character in the buffer to be ignored. This technique can be handy when using the joystick in Controller Port #1, which sometimes causes false keypresses to be registered, placing unwanted characters in the keyboard buffer.

Not only is this location handy for taking unwanted characters out of the keyboard buffer, but it can also be used to put desired characters into the buffer, and thus to *program* the keyboard buffer. This technique (dynamic keyboard) allows you to simulate keyboard input in direct mode from a program.

The dynamic keyboard technique is an extremely useful one, as it enables you to add, delete, or modify program lines while the program is running. The basic scheme is to POKE the PETASCII character values that you wish to be printed (including cursor control characters and carriage returns) into the buffer, and POKE this location with the number of characters in the buffer. Then, when an END statement is executed, the characters in the buffer will be printed, and entered by the carriage returns.

This technique can help with the problem of trying to use data separator and terminator characters with INPUT statements. If you try to INPUT a string that has a comma or colon, the INPUT will read only up to that character and issue an EXTRA IGNORED error message. You can avoid this by entering the input string in quotes, but this places on the user the burden of remembering the quote marks. One solution is to use the statement

```
POKE 198,3: POKE 631,34: POKE 632,34: POKE 633,20
```

before the input. This will force two quote marks and a delete into the buffer. The first quote mark allows the comma or colon to be INPUT, the second is used to get the editor out of quote mode, and the delete removes that second quote.

For more specific information and programming examples, see the description of location 631 ($277), the keyboard buffer.

199 $C7 RVS

Flag: Print Reverse Characters? 0=No

When the [CTRL] [RVS-ON] characters are printed (CHR$(18)), this flag is set to 18 ($12), and the print routines will add 128 ($80) to the screen code of each character which is printed, so that the character will appear on the screen with its colors inverted.

POKEing this location directly with a nonzero number will achieve the same results. You should remember, however, that the contents of this location are returned to 0 not only upon entry of a [CTRL] [RVS-OFF] character (CHR$(146)), but also at every carriage return. When this happens, characters printed thereafter appear with the normal combination of colors.

200 $C8 INDX

Pointer: End of Logical Line for Input

This pointer indicates the column number of the last nonblank character on the logical line that is to be input. Since a logical line can be up to 80 characters long, this number can range from 0-79.

201-202 $C9-$CA LXSP

Cursor X,Y Position at Start of Input

These locations keep track of the logical line that the cursor is on, and its column position on that logical line (in line, column format).

Each logical line may contain one or two 40-column physical lines. Thus there may be as many as 25 logical lines, or as few as 13 at any one time. Therefore, the logical line number might be anywhere from 1-25. Depending on the length of the logical line, the cursor column may be from 1-40 or 1-80.

For a more detailed explanation of logical lines, see the description of the screen line link table, 217 ($D9).

203 $CB SFDX

Matrix Coordinate of Current Key Pressed

The keyscan interrupt routine uses this location to indicate which key is currently being pressed. The value here is then used as an index into the appropriate keyboard table to determine which character to print when a key is struck.

The correspondence between the key pressed and the number stored here is as follows:

0 =INSERT/DELETE
1= RETURN
2= CURSOR RIGHT
3= f7
4= f1
5= f3
6= f5
7= CURSOR DOWN

8= 3
9= W
10= A
11= 4
12= Z
13= S
14= E
15= NOT USED (WOULD BE LEFT SHIFT)
16= 5
17= R
18= D
19= 6
20= C
21= F
22= T
23= X
24= 7
25= Y
26= G
27= 8
28= B
29= H
30= U
31= V
32= 9
33= I
34= J
35= 0
36= M
37= K
38= O
39= N
40= +
41= P
42= L
43= -
44= .
45= :
46= @
47= ,
48= LIRA (BRITISH POUND SIGN)
49= *
50= ;
51= CLEAR/HOME
52= NOT USED (WOULD BE RIGHT SHIFT)
53= =
54= UP ARROW (EXPONENTIATION SIGN)
55= /
56= 1
57= LEFT ARROW
58= NOT USED (WOULD BE CONTROL)
59= 2
60= SPACE BAR
61= NOT USED (WOULD BE COMMODORE LOGO)
62= Q
63= RUN/STOP
64= NO KEY PRESSED

The RESTORE key is not accounted for, because it is not part of the normal keyboard matrix. Instead, it is connected directly to the microprocessor NMI line, and causes an NMI interrupt whenever it is pressed.

204 $CC BLNSW

Cursor Blink Enable: 0=Flash Cursor

When this flag is set to a nonzero value, it indicates to the routine that normally flashes the cursor not to do so. The cursor blink is turned off when there are characters in the keyboard buffer, or when the program is running.

You can use this location to turn the cursor on during a program (for a series of GET operations, for example, to show the user that input is expected) by using the statement POKE 204,0.

205 $CD BLNCT

Timer: Countdown to Blink Cursor

The interrupt routine that blinks the cursor uses this location to tell when it's time for a blink. First the number 20 is put here, and every jiffy (1/60 second) the value here is decreased by one, until it

reaches zero. Then the cursor is blinked, the number 20 is put back here, and the cycle starts all over again. Thus, under normal circumstances, the cursor blinks three times per second.

206 $CE GDBLN

Character under Cursor

The cursor is formed by printing the inverse of the character that occupies the cursor position. If that character is the letter A, for example, the flashing cursor merely alternates between printing an A and a reverse-A. This location keeps track of the normal screen code of the character that is located at the cursor position, so that it may be restored when the cursor moves on.

207 $CF BLNON

Flag: Was Last Cursor Blink on or off?

This location keeps track of whether, during the current cursor blink, the character under the cursor was reversed, or was restored to normal. This location will contain a 0 if the character is reversed, and a 1 if the character is restored to its nonreversed status.

208 $D0 CRSW

Flag: Input from Keyboard or Screen

This flag is used by the Kernal CHRIN (61783, $F157) routine to indicate whether input is available from the screen (3), or whether a new line should be obtained from the keyboard (0).

209-210 $D1-$D2 PNT

Pointer to the Address of the Current Screen Line

This location points to the address in screen RAM of the first column of the logical line upon which the cursor is currently positioned.

211 $D3 PNTR

Cursor Column on Current Line

The number contained here is the cursor column position within the logical line pointed to by 209 ($D1). Since a logical line can contain up to two physical lines, this value may be from 0 to 79 (the number here is the value returned by the POS function).

212 $D4 QTSW

Flag: Editor in Quote Mode? 0=No

A nonzero value in this location indicates that the editor is in quote mode. Quote mode is toggled every time that you type in a quotation mark on a given line—the first quote mark turns it on, the second turns it off, the third turns it back on, etc.

If the editor is in this mode when a cursor control character or

other nonprinting character is entered, a printed equivalent will appear on the screen instead of the cursor movement or other control operation taking place. Instead, that action is deferred until the string is sent to the screen by a PRINT statement, at which time the cursor movement or other control operation will take place.

The exception to this rule is the DELETE key, which will function normally within quote mode. The only way to print a character which is equivalent to the DELETE key is by entering insert mode (see location 216, $D8). Quote mode may be exited by printing a closing quote character, or by hitting the RETURN or SHIFT-RETURN keys.

Sometimes, it would be handy to be able to escape from quote mode or insert mode without skipping to a new line. The machine language program below hooks into the keyscan interrupt routine, and allows you to escape quote mode by changing this flag to 0 when you press the f1 key:

```
10 FOR I=850 TO I+41:READ A:POKE I,A:NEXT
20 PRINTCHR$(147)"PRESS F1 KEY TO ESCAPE QUOTE MOD
   E"
30 PRINT"TO RESTART AFTER RESTORE ONLY, SYS 850":S
   YS 850:NEW
40 DATA{2 SPACES}173 , 143 , 2 , 141 , 46 , 3 , 17
   3 , 144 , 2 , 141
50 DATA 47 , 3 , 120 , 169 , 107 , 141 , 143 , 2 ,
    169 , 3
60 DATA 141 , 144 , 2 , 88 , 96 , 165 , 203 , 201
   {SPACE}, 4 , 208
70 DATA 8 , 169 , 0 , 133 , 212 , 133 , 216 , 133
   {SPACE}, 199 , 108, 46, 3
```

213 $D5 LNMX

Maximum Length of Physical Screen Line

The line editor uses this location when the end of a line has been reached to determine whether another physical line can be added to the current logical line, or if a new logical line must be started.

214 $D6 TBLX

Current Cursor Physical Line Number

This location contains the current physical screen line position of the cursor (0-24). It can be used in a fashion to move the cursor vertically, by POKEing the target screen line (1-25) minus 1 here, followed by a PRINT command. For example,

POKE 214,9:PRINT:PRINT "WE'RE ON LINE ELEVEN"

prints the message on line 11. The first PRINT statement allows the system to update the other screen editor variables so that they will

also show the new line. The cursor can also be set or read using the Kernal PLOT routine (58634, $E50A) as explained in the entry for locations 780-783 ($30C-30F).

215 $D7

Temporary Storage Area for ASCII Value of Last Character Printed

The ASCII value of the last character printed to the screen is held here temporarily.

216 $D8 INSRT

Flag: Insert Mode (Any Number Greater Than 0 Is the Number of Inserts)

When the INST key is pressed, the screen editor shifts the line to the right, allocates another physical line to the logical line if necessary (and possible), updates the screen line length in 213 ($D5), and adjusts the screen line link table at 217 ($D9). This location is used to keep track of the number of spaces that has been opened up in this way.

Until the spaces that have been opened up are filled, the editor acts as if in quote mode (see location 212 ($D4), the quote mode flag). This means that cursor control characters that are normally nonprinting will leave a printed equivalent on the screen when entered, instead of having their normal effect on cursor movement, etc. The only difference between insert and quote mode is that the DELETE key will leave a printed equivalent in insert mode, while the INST key will insert spaces as normal.

217-242 $D9-$F2 LDTB1

Screen Line Link Table/Editor Temporary Storage

This table contains 25 entries, one for each row of the screen display. Each entry has two functions. Bits 0-3 indicate on which of the four pages of screen memory the first byte of memory for that row is located. This is used in calculating the pointer to the starting address of a screen line at 209 ($D1).

While earlier PETs used one table for the low bytes of screen rows and another for the high bytes, this is not possible on the 64, where screen memory is not fixed in any one spot. Therefore, the Operating System uses a table of low bytes at 60656 ($ECF0), but calculates the high byte by adding the value of the starting page of screen memory held in 648 ($288) to the displacement page held here.

The other function of this table is to establish the makeup of logical lines on the screen. While each screen line is only 40 characters long, BASIC allows the entry of program lines that contain up to 80 characters. Therefore, some method must be used to determine

which pairs of physical lines are linked into a longer logical line, so that this longer logical line may be edited as a unit.

The high bit of each byte here is used as a flag by the screen editor. That bit is set (leaving the value of the byte over 128, $80) when a line is the first or only physical line in a logical line. The high bit is reset to 0 only when a line is the second half of a logical line.

243-244 $F3-$F4 USER

Pointer to the Address of the Current Screen Color RAM Location

This pointer is synchronized with the pointer to the address of the first byte of screen RAM for the current line kept in location 209 ($D1). It holds the address of the first byte of color RAM for the corresponding screen line.

245-246 $F5-$F6 KEYTAB

Vector: Keyboard Decode Table

KEYTAB points to the address of the keyboard matrix lookup table currently being used. Although there are only 64 keys on the keyboard matrix, each key can be used to print up to four different characters, depending on whether it is struck by itself or in combination with the SHIFT, CONTROL, or Commodore logo keys.

The tables pointed to by this address hold the ASCII value of each of the 64 keys for one of these possible combinations of keypresses. When it comes time to print the character, the table that is used determines which character is printed.

The addresses of the four tables are:

60289 ($EB81) = default uppercase/graphics characters (unshifted)
60354 ($EBC2) = shifted characters
60419 ($EC03) = Commodore logo key characters
60536 ($EC78) = CONTROL characters

The concept of the keyboard matrix tables should not be confused with changing the character sets from uppercase/graphics to upper/lowercase. The former involves determining what character is to be placed into screen memory, while the latter involves determining which character data table is to be used to decode the screen memory into individual dots for the display of characters on the screen. That character base is determined by location 53272 ($D018) of the VIC-II chip.

247-248 $F7-$F8 RIBUF

Pointer: RS-232 Input Buffer

When device number 2 (the RS-232 channel) is opened, two buffers of 256 bytes each are created at the top of memory. This location points to the address of the one which is used to store characters as

they are received. A BASIC program should always OPEN device 2 before assigning any variables to avoid the consequences of overwriting variables which were previously located at the top of memory, as BASIC executes a CLR after opening this device.

249-250 $F9-$FA ROBUF

Pointer: RS-232 Output Buffer

This location points to the address of the 256-byte output buffer which is used for transmitting data to RS-232 devices (device number 2).

251-254 $FB-$FE FREKZP

Four Free Bytes of Zero Page for User Programs

These locations were specifically set aside for user-written ML routines that require zero-page addressing. While other zero-page locations can be used on a noninterference basis, it is guaranteed that BASIC will not alter these locations.

255 $FF BASZPT

BASIC Temporary Data for Floating Point to ASCII Conversion

This location is used for temporary storage in the process of converting floating point numbers to ASCII characters.

Chapter 2

Page 1

Page 1

256-511 $100-$1FF

Microprocessor Stack Area

Locations 256-511 are reserved for the 6510 microprocessor hardware stack. The organization of this temporary storage area has often been compared to that of a push-down stack of trays at a cafeteria. The first number placed on the stack goes to the bottom, and subsequent entries are placed on top of it. When you pull a number off the stack, you come up with the last number that was pushed on (such a stack is called a Last In, First Out, or LIFO stack).

The stack is controlled by one of the microprocessor registers called the Stack Pointer, which keeps track of the last stack location used. The first number placed on the stack goes to location 511 ($1FF), and subsequent entries are built downward toward 256 ($100). If more than 256 numbers are pushed onto the stack, the Stack Pointer will start counting from 0 again, and an overflow error will result. Likewise, if you try to pull more items off the stack than have been pushed on, an underflow error will result. Most often, such errors will cause the system to go haywire, and nothing will operate until you turn the power off and on again.

The stack is used by the system to keep track of the return addresses of machine language subroutines and interrupt calls and to save the contents of internal registers.The stack can also be used by the programmer for temporary storage. BASIC and the Kernal make heavy use of the stack.

Microsoft BASIC uses part of the stack for a temporary work area. Therefore, the stack may be broken down into the following subregions:

256-266 $100-$10A

Work Area for Floating Point to String Conversions

Used in the conversion of numbers to the equivalent ASCII digits, and in scanning strings.

256-318 $100-$13E BAD

Tape Input Error Log

Each tape block is saved twice consecutively, in order to minimize loss of data from transmission errors. These 62 bytes serve as indices

of which bytes in the tape block were not received correctly during the first transmission, so that corrrections might be made on the second pass.

319-511 $13F-$1FF

This area is used exclusively for the microprocessor stack. Some BASIC commands, such as FOR-NEXT loops require many stack entries at a time. Therefore, BASIC frequently checks the stack before pushing entries on, and returns an OUT OF MEMORY error if an operation would result in less than 62 bytes of available stack memory.

Each FOR statement causes 18 bytes to be pushed onto the stack, which come off in the following order:

First comes a one-byte constant of 129 ($81). Next is a two-byte pointer to the address of the subject variable (the X of FOR X=1 to 10). This is followed by the five-byte floating point representation of the STEP value, the one-byte sign of the STEP, and the five-byte floating point representation of the TO value. Finally comes the two-byte line number of the line to which the program returns after a NEXT, and the two-byte address of the next character to read in that line after the FOR statement.

Each GOSUB call places five bytes on the stack. The first byte to come off is a one-byte constant of 141 ($8D). The next two bytes contain the line number of the statement to which the program will RETURN after the subroutine ends. And the final two bytes are a pointer to the address of the BASIC program text for that statement to which the program RETURNs.

DEF also leaves a five-byte entry on the stack. It is the same as that described for GOSUB, except that instead of a constant byte of 141, the first number is a dummy byte, whose value has no significance.

Chapter 3

Pages 2 and 3

BASIC and the Kernal Working Storage

Pages 2 and 3
BASIC and Kernal Working Storage

This area is used to store important information for the Operating System and BASIC. It contains vectors to certain BASIC routines as well as Operating System Kernal routines. Registers for RS-232 serial I/O are located here. Buffer space is allocated in this area for tape I/O, BASIC text input, and the keyboard queue. In addition, there are a number of Operating System variables and pointers here which the programmer can utilize.

512-600 $200-$258 BUF
BASIC Line Editor Input Buffer

When you are in the BASIC immediate mode, and type in a line of characters, those characters are stored here. BASIC then scans the string of characters, converts the text to tokenized BASIC program format, and either stores it or executes the line, depending on whether or not it started with a line number.

This same area is also used to store data which is received via the INPUT and GET commands. This explains why these commands are illegal in immediate mode—they must use the same buffer space that is required by the immediate mode statement itself.

It is interesting to note that this buffer is 89 bytes long. The screen editor will allow a maximum of only 80 characters in a program line, with one extra byte required for a 0 character, marking the end of the line. This presumably is a carry-over from the VIC, which allows a line length of up to 88 characters. The last eight bytes of this buffer are therefore normally not used, and can be considered free space for the programmer to use as he or she sees fit.

Location Range: 601-630 ($259-$276)
Tables for File Numbers, Device Numbers, and Secondary Addresses

All three of the tables here have room for ten one-byte entries, each of which represents an active Input/Output file. When an I/O file is opened, its logical file number is put into the table at 601 ($259), the

device number of the I/O device is put into the table at 611 ($263), and its secondary address is put into the table at 621 ($26D).

The entry for any particular I/O file will occupy the same position in each of the three tables. That is, if logical file number 2 is the third entry in the file number table, its secondary address will be the third entry in the secondary address table, and its corresponding device number will occupy the third spot in the device number table.

Every time a device is OPENed, its information is added as the last entry in each table, and the value at location 152 ($98) is increased by one, indicating that there is one more active I/O file. When a device is CLOSEd, the value at location 152 is decreased by one, and all entries that occupy a position in the tables that is higher than that of the closed device are moved down one position, thus eliminating the entry for that device. The Kernal CLALL routine (62255, $F32F) simply zeros location 152, which has the effect of emptying these tables.

601-610 $259-$262 LAT
Kernal Table of Active Logical File Numbers

611-620 $263-$26C FAT
Kernal Table of Device Numbers for Each Logical File

621-630 $26D-$276 SAT
Kernal Table of Secondary Addresses for Each Logical File

631-640 $277-$280 KEYD
Keyboard Buffer (Queue)

This buffer, sometimes also referred to as the keyboard queue, holds the ASCII values of the characters entered from the keyboard. The interrupt routine which scans the keyboard deposits a character here each time a key is pressed. When BASIC sees that there are characters waiting, it removes and prints them, one by one, in the order in which they were entered.

This kind of a buffer is known as FIFO, for First In, First Out. The buffer will hold up to ten characters, allowing you to type faster than the computer prints characters, without losing characters. The maximum number of characters this buffer can hold at one time is ten (as determined by the value at 649, $289). Characters entered after the buffer is full will be ignored.

The commands GET and INPUT retrieve characters from this buffer. If one of these is executed while there are already characters waiting in the buffer, those characters will be fetched as if they were part of the data being input. To prevent this from happening, you can clear the buffer by POKEing a 0 into location 198 ($C6), which holds the number of characters that are waiting in the buffer.

One of the most interesting and useful techniques for programming Commodore computers is to have a program simulate direct entry of commands from the keyboard. This dynamic keyboard trick is achieved by first POKEing PETASCII characters, usually cursor movement characters and carriage returns, into the buffer, and setting location 198 ($C6) to show how many characters are waiting in the buffer.

Next, you clear the screen, and PRINT the statements that you wish to have executed on the screen, carefully positioning them so that the first statement to be entered is on the fourth line of the screen.

You then home the cursor and execute an END statement. This causes the keyboard buffer to be read, and the carriage returns to be executed, thus entering the printed lines as if they had been typed in immediate or direct mode. The program can be continued by including a GOTO statement in the last line entered.

Many interesting effects can be achieved using this method. Examples of a few of these are included below. For example, program lines can be added, modified, or deleted, while the program is running. The following example shows how this is done:

```
10 REM THIS LINE WILL BE DELETED
20 REM A NEW LINE 30 WILL BE CREATED
40 PRINT CHR$(147):PRINT:PRINT
50 PRINT "80 LIST":PRINT"30 REM THIS LINE WASN'T H
   ERE BEFORE"
60 PRINT "10":PRINT "GOTO 80"CHR$(19)
70 FOR I=631 TO 634:POKE I,13:NEXT:POKE 198,4:END
80 REM THIS LINE WILL BE REPLACED
```

You can use this technique to enter numbered DATA statements automatically, using values in memory. These statements become a permanent part of the program.

Another interesting application is taking ASCII program lines from a tape data file, or sequential disk file, and having them entered automatically. This can be used for merging programs, or for transferring programs between computers with a modem and a terminal program. To create the ASCII program file, you use CMD to direct a LISTing to the desired device as follows:

For tape: OPEN 1,1,1,"ASCII":CMD 1:LIST
After the listing has ended: PRINT #1:CLOSE 1

For disk: OPEN 8,8,8,"ASCII,S,W":CMD 8:LIST
After the listing has ended: PRINT#8:CLOSE 8

This file can then be uploaded using a modem and appropriate terminal software, entered by itself or merged with another program by using the following program. Be sure to save this program before you run it, because it will erase itself when it is done.

```
60000 OPEN 1,8,8,"ASCII"
60010 POKE 152,1:B=0:GOSUB 60170
60020 GET #1,A$:IF A$=""THEN60020
60030 IF ST AND 64 THEN 60120
60040 IF A$=CHR$(13)AND B=0THEN 60020
60050 PRINT A$;:B=1:IF A$=CHR$(34)THEN POKE 212,0
60060 IF A$<>CHR$(13) THEN 60020
60070 PRINT CHR$(5);"GOTO 60010";CHR$(5):PRINT:PRI
      NT:POKE 198,0
60080 PRINT "RETURN=KEEP LINE{4 SPACES}S=SKIP LINE
      ":B=0
60090 GET A$:IF A$=""THEN 60090
60100 IF A$="S" THEN 60010
60110 GOTO 60180
60120 PRINT "END OF FILE--HIT RETURN TO FINISH MER
      GE"
60130 IF PEEK(197)<>1THEN60130
60140 A=60000
60150 GOSUB 60170:FOR I=A TO A+60 STEP10:PRINTI:NE
      XT
60160 PRINT "A="I":GOTO 60150":GOTO 60180
60170 PRINT CHR$(147):PRINT:PRINT:RETURN
60180 FOR I=631TO640:POKEI,13:NEXT:POKE198,10:PRIN
      TCHR$(19);:END
```

If you wish to merge additional programs at the same time, when it indicates that the file has ended, press the STOP key rather than RETURN, enter the name of the new file in line 60000, and RUN 60000.

641-642 $281-$282 MEMSTR

Pointer: O.S. Start of Memory

When the power is first turned on, or a cold start RESET is performed, the Kernal routine RAMTAS (64848, $FD50) sets this location to point to address 2048 ($800). This indicate that this is the starting address of user RAM. BASIC uses this location to set its own start of memory pointer at location 43 ($2B), and thereafter uses only its own pointer.

The Kernal routine MEMBOT (65076, $FE34) may be used to read or set this pointer, or these locations may be directly PEEKed or POKEd from BASIC.

643-644 $282-$284 MEMSIZ

Pointer: O.S. End of Memory

When the power is first turned on, or a cold start RESET is performed, the Kernal routine RAMTAS (64848,$FD50) performs a non-destructive test of RAM from 1024 ($400) up, stopping when the test

fails, indicating the presence of ROM. This will normally occur at 40960 ($A000), the location of the BASIC ROM. The top of user RAM pointer is then set to point to that first ROM location.

After BASIC has been started, the system will alter this location only when an RS-232 channel (device number 2) is OPENed or CLOSEd. As 512 bytes of memory are required for the RS-232 transmission and reception buffers, this pointer, as well as the end of BASIC pointer at 55 ($37), is lowered to create room for those buffers when the device is opened. CLOSing the device resets these pointers.

The Kernal routine MEMTOP (65061, $FE25) may be used to read or set this pointer.

645 $285 TIMOUT

Flag: Kernal Variable for IEEE Time-Out

This location is used only with the external IEEE interface card (which was not yet available from Commodore at the time of writing). For more information, see the entry for the Kernal SETTMO routine at 65057 ($FE21).

646 $286 COLOR

Current Foreground Color for Text

The process of PRINTing a character to the screen consists of both placing the screen code value for the character in screen memory and placing a foreground color value in the corresponding location in color RAM. Whenever a character is PRINTed, the Operating System fetches the value to be put in color RAM from this location.

The foreground color may be changed in a number of ways. Pressing the CTRL or Commodore logo key and numbers 1-8 at the same time will change the value stored here, and thus the color being printed. PRINTing the PETASCII equivalent character with the CHR$ command will have the same effect. But probably the easiest method is to POKE the color value directly to this location. The table below lists the possible colors that may be produced, and shows how to produce them using all three methods.

POKE COLOR#	COLOR	CHR$	KEYS TO PRESS
0	BLACK	144	CTRL-1
1	WHITE	5	CTRL-2
2	RED	28	CTRL-3
3	CYAN	159	CTRL-4
4	PURPLE	156	CTRL-5
5	GREEN	30	CTRL-6
6	BLUE	31	CTRL-7
7	YELLOW	158	CTRL-8
8	ORANGE	129	Logo-1

POKE COLOR#	COLOR	CHR$	KEYS TO PRESS
9	BROWN	149	Logo-2
10	LT RED	150	Logo-3
11	DARK GRAY	151	Logo-4
12	MED GRAY	152	Logo-5
13	LT GREEN	153	Logo-6
14	LT BLUE	154	Logo-7
15	LT GRAY	155	Logo-8

647 $287 GDCOL

Color of Character under Cursor

This location is used to keep track of the original color code of the character stored at the present cursor location. Since the blinking cursor uses the current foreground color at 646 ($286), the original value must be stored here so that if the cursor moves on without changing that character, its color code can be restored to its original value.

648 $288 HIBASE

Top Page of Screen Memory

This location contains the value used by the Operating System routines that print to the screen as the base address for screen RAM. The top of screen memory can be found by multiplying this location by 256. The default value for screen RAM is set on power-up to location 1024 ($400), and this location therefore usually contains a 4.

Screen display memory on the Commodore 64 can be moved to start on any 1K boundary (location evenly divisible by 1024). This is done by manipulating the VIC-II memory control register at 53272, ($D018), and the VIC chip memory bank select at location 56576 ($DD00).

It is important to note, however, that while any area may be displayed, the Operating System will look here to find out where it should PRINT characters. Therefore, if you change the screen location by altering the contents of one of the two addresses listed above, the Operating System will still not know where to PRINT characters unless you also change this address as well. The result will be that characters entered from the keyboard or PRINTed will not appear on the screen.

Examples of how to properly relocate the screen can be found at the entries for location 53272 ($D018) and 43 ($2B).

Since the PRINT command in essence just POKEs a lot of values to screen and color memory, by changing this pointer you can *print* a string of characters to memory locations other than screen RAM. For example, you could PRINT a sprite shape to memory without

having to READ a lot of DATA statements. The program below PRINTs different sprite shapes into the sprite data area:

```
10 SP=53248:POKESP,170:POKESP+1,125:POKESP+21,1:PO
   KE 2040,13:PRINT CHR$(147)
20 A$="THIS TEXT WILL BE PRINTED TO THE SPRITE SHA
   PE DATA AREA AND DISPLAYED"
30 GOSUB 100
40 A$="THIS IS SOME DIFFERENT TEXT TO BE PRINTED T
   O THE SPRITE SHAPE AREA"
50 GOSUB 100
60 COUNT=COUNT+1:IF COUNT<15 THEN 20
70 END
100 POKE 648,3:PRINTCHR$(19);CHR$(17);SPC(24);A$;:
    POKE 648,4:RETURN
```

Since PRINTing also changes color memory, you can change the pointer to print the characters harmlessly to ROM, while changing a lot of screen RAM at one time, as the following program demonstrates:

```
10 D$=CHR$(94):FOR I=1 TO 4:D$=D$+D$:NEXT
20 PRINT CHR$(147);:FOR I=1 TO 7:PRINT TAB(10) D$:
   NEXT:PRINT:PRINT:PRINT:PRINT
30 PRINT TAB(9);CHR$(5);"HIT ANY KEY TO STOP"
40 DIM C(15):FOR I=0 TO 14:READ A:C(I)=A:NEXT:DATA
    2,8,7,5,6,4,1,2,8,7,5,6,4,1,2
50 POKE 53281,0:POKE 648,212:FOR J=0 TO 6:PRINT CH
   R$(19);
60 FOR I=J TO J+6:POKE 646,C(I):PRINT TAB(10) D$:N
   EXT I,J
70 GET A$:IF A$="" THEN 50
80 POKE 648,4:POKE 646,1
```

649 $289 XMAX

Maximum Keyboard Buffer Size

The value here indicates the maximum number of characters that the keyboard buffer at 631 ($277) may hold at any one time. Anytime that the current buffer length in location 198 ($C6) matches the value here, further keypresses will be ignored.

Although the maximum size of the keyboard buffer is usually 10 characters, it may be possible to extend it to up to 15 characters by changing the number here. This could cause the Operating System pointers to the bottom and top of memory at 641-644 ($281-$284) to be overwritten, but no real harm should result.

650 $28A RPTFLG

Flag: Which Keys Will Repeat?

The flag at this location is used to determine whether to continue printing a character as long as its key is held down, or whether to wait until the key is let up before allowing it to be printed again. The default value here is 0, which allows only the cursor movement keys, insert/delete key, and the space bar to repeat.

POKEing this location with 128 ($80) will make all keys repeating, while a value of 64 ($40) will disable all keys from repeating.

651 $28B KOUNT

Counter for Timing the Delay Between Key Repeats

This location is used as a delay counter to determine how long to wait while a key is being held down until the next repeat printing of that key.

The value here starts at 6. If location 652 ($28C) contains a 0, the value in this location is counted down once every 1/60 second, so long as the same key is held down. When this counter gets to 0, and if the repeat flag at 650 ($28A) allows that key to repeat, its ASCII equivalent will once again be placed in the keyboard buffer. A value of 4 is then placed in location 651, allowing subsequent repeats to occur at a rate of 15 per second.

652 $28C DELAY

Counter for Timing the Delay Until the First Key Repeat Begins

This location is used as a delay counter to determine how long a key must be held down before the entry of that key should be repeated.

The initial value of 16 is counted down every 1/60 second, as long as the same key remains pressed. When the value gets to 0, location 651 ($28B) is counted down from 6, and the key is repeated when the value there reaches 0. Thus a total of 22/60, or approximately 1/3, second will elapse before the first repeat of a key. The value here will be held to 0 after the first repeat, so that subsequent keystroke repetitions occur much more quickly.

653 $28D SHFLAG

Flag: SHIFT/CTRL/Logo Keypresss

This flag signals which of the SHIFT, CTRL, or Commodore logo keys are currently being pressed, if any.

A value of 1 signifies that one of the SHIFT keys is being pressed, a 2 shows that the Commodore logo key is down, and 4 means that the CTRL key is being pressed. If more than one key is held down, these values will be added; for example, a 3 indicates that SHIFT and logo are both held down.

The value here is used by the Operating System when

determining how to convert a keypress into a PETASCII character. There are four different tables used to translate one of the 64 keys on the keyboard matrix into a PETASCII character, and the combination of special SHIFT keys determines which of these tables will be used (see the entry for location 245 ($F5) for more details on the keyboard tables).

Pressing the SHIFT and Commodore logo keys at the same time will toggle the character set that is presently being used between the uppercase/graphics set, and the lowercase/uppercase set (provided that the flag at 657 ($291) has not been set to disable this switch). This changes the appearance of all of the characters on the screen at once. It has nothing whatever to do with the keyboard shift tables, however, and should not be confused with the printing of SHIFTed characters, which affects only one character at a time. Rather, it is the result of the value of the character dot data table base address in 53272 ($D018) being changed. The same result may be obtained by POKEing that address directly.

654 $28E LSTSHF

Last Pattern of SHIFT/CTRL/Logo Keypress

This location is used in combination with the one above to debounce the special SHIFT keys. This will keep the SHIFT/logo combination from changing character sets back and forth during a single pressing of both keys.

655-656 $28F-$290 KEYLOG

Vector to Keyboard Table Setup Routine

This location points to the address of the Operating System routine which actually determines which keyboard matrix lookup table will be used.

The routine looks at the value of the SHIFT flag at 653 ($28D), and based on what value it finds there, stores the address of the correct table to use at location 245 ($F5).

The interrupt driven keyboard-scanning routine jumps through this RAM vector to get to the table setup routine. Therefore, it is possible to alter the address contained in this vector, and direct the keyscan routine to your own routine, which can check the keypress and SHIFT combination, and act before a character is printed.

Since this routine comes after the keypress, but before it is printed, this is a very good place to have your preprocessor routine check for a particular keypress. An excellent example of such a program is the "VICword" program by Mark Niggemann, *COMPUTE!'s Second Book of VIC*. This program adds a machine language routine that checks if the SHIFT or Commodore logo key is pressed while not in quote mode. If it finds one of these keypresses, it substitutes an entire BASIC keyword for the letter (A-Z) of the key that was pressed.

An adaptation of that program for the 64 appears below.

```
100 IF PEEK(PEEK(56)*256)<>120THENPOKE56,PEEK(56)-
    1:CLR
110 HI=PEEK(56):BASE=HI*256
120 PRINTCHR$(147)"READING DATA"
130 FOR AD=0 TO 211:READ BY
140 POKE BASE+AD,BY:NEXT AD
150 :
200 REM RELOCATION ADJUSTMENTS
210 POKE BASE+26,HI:POKE BASE+81,HI
220 POKE BASE+123,HI:POKE BASE+133,HI
230 :
240 PRINT CHR$(147) TAB(15)"***64WORD***":PRINT
250 PRINT"TO TOGGLE THE PROGRAM ON/OFF:":PRINT:PRI
    NT:PRINT "SYS";BASE;
260 PRINT CHR$(145);CHR$(145);
270 DATA 120,173,143,2,201,32
280 DATA 208,12,169,220,141,143
290 DATA 2,169,72,141,144,2
300 DATA 88,96,169,32,141,143
310 DATA 2,169,0,141,144,2
320 DATA 88,96,165,212,208,117
330 DATA 173,141,2,201,3,176
340 DATA 110,201,0,240,106,169
350 DATA 194,133,245,169,235,133
360 DATA 246,165,215,201,193,144
370 DATA 95,201,219,176,91,56
380 DATA 233,193,174,141,2,224
390 DATA 2,208,3,24,105,26
400 DATA 170,189,159,0,162,0
410 DATA 134,198,170,160,158,132
420 DATA 34,160,160,132,35,160
430 DATA 0,10,240,16,202,16
440 DATA 12,230,34,208,2,230
450 DATA 35,177,34,16,246,48
460 DATA 241,200,177,34,48,17
470 DATA 8,142,211,0,230,198
480 DATA 166,198,157,119,2,174
490 DATA 211,0,40,208,234,230
500 DATA 198,166,198,41,127,157
510 DATA 119,2,230,198,169,20
520 DATA 141,119,2,76,72,235
530 DATA 76,224,234
550 REM TOKENS FOR SHIFT KEY
570 DATA 153,175,199,135,161,129
580 DATA 141,164,133,137,134,147
590 DATA 202,181,159,151,163,201
600 DATA 196,139,192,149,150,155
610 DATA 191,138
```

```
630 REM TOKENS FOR COMMODORE KEY
650 DATA 152,176,198,131,128,130
660 DATA 142,169,132,145,140,148
670 DATA 195,187,160,194,166,200
680 DATA 197,167,186,157,165,184
690 DATA 190,158,0
```

Commodore 64word: Keys into BASIC Commands

KEY	*SHIFT*	*COMMODORE*
A	PRINT	PRINT#
B	AND	OR
C	CHR$	ASC
D	READ	DATA
E	GET	END
F	FOR	NEXT
G	GOSUB	RETURN
H	TO	STEP
I	INPUT	INPUT#
J	GOTO	ON
K	DIM	RESTORE
L	LOAD	SAVE
M	MID$	LEN
N	INT	RND
O	OPEN	CLOSE
P	POKE	PEEK
Q	TAB(	SPC(
R	RIGHT$	LEFT$
S	STR$	VAL
T	IF	THEN
U	TAN	SQR
V	VERIFY	CMD
W	DEF	FN
X	LIST	FRE
Y	SIN	COS
Z	RUN	SYS

657 $291 MODE

Flag: Enable or Disable Changing Character Sets with SHIFT/ Logo Keypress

This flag is used to enable or disable the feature which lets you switch between the uppercase/graphics and upper/lowercase character sets by pressing the SHIFT and Commodore logo keys simultaneously.

This flag affects only this special SHIFT key function, and does not affect the printing of SHIFTed characters. POKEing a value of

128 ($80) here will disable this feature, while POKEing a value of 0 will enable it once more. The same effect can be achieved by PRINTing CHR$(8) or CTRL-H to disable the switching of character sets, and CHR$(9) or CTRL-I to enable it. See entries for locations 53272 ($D018) and 53248 ($D000) for more information on switching character sets.

658 $292 AUTODN

Flag: Screen Scroll Enabled

This location is used to determine whether moving the cursor past the fortieth column of a logical line will cause another physical line to be added to the logical line.

A value of 0 enables the screen to scroll the following lines down in order to add that line; any nonzero value will disable the scroll. This flag is set to disable the scroll temporarily when there are characters waiting in the keyboard buffer (these may include cursor movement characters that would eliminate the need for a scroll).

Location Range:659-663 ($293-$297)

RS-232 Pseudo 6551 Registers

For serial Input/Output via the RS-232 port, the internal software of the Commodore 64 emulates the operation of a 6551 UART chip (that's Universal Asynchronous Receiver/Transmitter, for you acronym buffs), also known as an ACIA (Asynchronous Communications Interface Adapter).

These RAM locations are used to mimic the functions of that chip's hardware command, control, and status registers. Although RAM locations are allocated for mimicking the 6551's ability to use either an on-board baud rate generator or an external clock crystal, this function is not implemented by the internal software.

Provisions have been made for the user to communicate with these registers through the RS-232 OPEN command. When device 2 is opened, a filename of up to four characters may be appended. These four characters are copied to locations 659-662 ($293-$296), although the last two, which specify a nonstandard baud rate, are not used because that feature is not implemented.

659 $293 M51CTR

RS-232: Mock 6551 Control Register

This location is used to control the RS-232 serial I/O baud rate (speed at which data is transmitted and received), the word length (number of bits per data character), and the number of stop bits used to mark the end of a transmitted character. It uses the same format as that of the 6551 UART control register to set these parameters, although, as you will see, some of the 6551 configurations are not

implemented by the software that emulates the UART device. For example, the standard baud rates which are higher than 2400 baud are not implemented, presumably because the software cannot keep up at higher rates. The meanings of the various bit-patterns are as follows:

Bit 7: STOP Bits

0	(bit value of 0)	=1 STOP Bit
1	(bit value of 128)	=2 STOP Bits

Bits 6-5: WORD LENGTH

00	(bit value of 0)	=8 DATA Bits
01	(bit value of 32)	=7 DATA Bits
10	(bit value of 64)	=6 DATA Bits
11	(bit value of 96)	=5 DATA Bits

Bit 4: Unused

Bits 3-0: BAUD RATE

0000	(bit value of 0)	=Nonstandard (User-Defined) Rate (Not Implemented)
0001	(bit value of 1)	=50 Baud
0010	(bit value of 2)	=75 Baud
0011	(bit value of 3)	=110 Baud
0100	(bit value of 4)	=134.5 Baud
0101	(bit value of 5)	=150 Baud
0110	(bit value of 6)	=300 Baud
0111	(bit value of 7)	=600 Baud
1000	(bit value of 8)	=1200 Baud
1001	(bit value of 9)	=1800 Baud
1010	(bit value of 10)	=2400 Baud
1011	(bit value of 11)	=3600 Baud (Not Implemented on the Commodore 64)
1100	(bit value of 12)	=4800 Baud (Not Implemented on the Commodore 64)
1101	(bit value of 13)	=7200 Baud (Not Implemented on the Commodore 64)
1110	(bit value of 14)	=9600 Baud (Not Implemented on the Commodore 64)
1111	(bit value of 15)	=19200 Baud (Not Implemented on the Commodore 64)

This register is the only one which must be set when opening RS-232 device (number 2). The first character of the filename will be stored here. For example, the statement OPEN 2,2,0,CHR$(6+32) will set the value of this location to 38. As you can see from the above chart, this sets up the RS-232 device for a data transfer rate of 300 baud, using seven data bits per character and one stop bit.

660 $294 M51CDR

RS-232: Mock 6551 Command Register

This location performs the same function as the 6551 UART chip's command register, which specifies type of parity, duplex mode, and handshaking protocol.

The type of parity used determines how the 64 will check that RS-232 data is received correctly.

The duplex mode can be either full duplex (the 64 will be able to transmit at the same time it is receiving) or half duplex (it will take turns sending and receiving).

The handshaking protocol has to do with the manner in which the sending device lets the receiver know that it is ready to send data, and the receiver lets the sender know that it has gotten the data correctly. The meanings of the bit patterns in this register are as follows:

Bits 7-5: Parity

XX0 (bit value of 0,64,128 or 192)	=No Parity Generated or Received
001 (bit value of 32)	=Odd Parity Transmitted and Received
011 (bit value of 96)	=Even Parity Transmitted and Received
101 (bit value of 160)	=Mark Parity Transmitted, None Checked
111 (bit value of 224)	=Space Parity Transmitted, None Checked

Bit 4: Duplex

0	(bit value of 0)	=Full Duplex
1	(bit value of 16)	=Half Duplex

Bits 3-1: Unused

Bit 0: Handshake Protocol

0	(bit value of 0)	=3 Line
1	(bit value of 1)	=X Line

This register can be set at the user's option when opening RS-232 device (number 2). The second character of the filename will be stored here. For example, the statement

```
OPEN 2,2,0,CHR$(6+32)+CHR$(32+16)
```

will set the value of this location to 48, which is the value of the second character in the filename portion of the statement. As you can see from the above chart, this configures the RS-232 device for half duplex data transfer using odd parity and three-line handshaking.

661-662 $295-$296 M51AJB

RS-232: Nonstandard Bit Timing

These locations are provided for storing a nonstandard user-defined baud rate, to be used when the low nybble of the control register at 659 ($293) is set to 0. They were presumably provided to conform to

the model of the 6551 UART device, which allows a nonstandard baud rate to be generated from an external reference crystal. However, the software emulation of that feature is not provided in the current version of the Kernal, and thus these locations are currently nonfunctional.

Nonetheless, Commodore has specified that if the nonstandard baud rate feature is implemented, the value placed here should equal the system clock frequency divided by the baud rate divided by 2 minus 100, stored in low-byte, high-byte order. The system clock frequency for American television monitors (NTSC standard) is 1.02273 MHz, and for European monitors (PAL standard) .98525 MHz.

663 $297 RSSTAT

RS-232: Mock 6551 Status Register

The contents of this register indicate the error status of RS-232 data transmission. That status can be determined by PEEKing this location directly, by referencing the BASIC reserved variable ST, or by using the Kernal READST (65031, $FE07) routine.

Note that if you use ST or Kernal, this location will be set to 0 after it is read. Therefore, if you need to test more than one bit, make sure that each test preserves the original value, because you won't be able to read it again. The meaning of each bit value is specified below:

Bit 7: 1 (bit value of 128)=Break Detected
Bit 6: 1 (bit value of 64)=DSR (Data Set Ready) Signal Missing
Bit 5: Unused
Bit 4: 1 (bit value of 16)=CTS (Clear to Send) Signal Missing
Bit 3: 1 (bit value of 8)=Receiver Buffer Empty
Bit 2: 1 (bit value of 4)=Receiver Buffer Overrun
Bit 1: 1 (bit value of 2)=Framing Error
Bit 0: 1 (bit value of 1)=Parity Error

The user is responsible for checking these errors and taking appropriate action. If, for example, you find that Bit 0 or 1 is set when you are sending, indicating a framing or parity error, you should resend the last byte. If Bit 2 is set, the GET#2 command is not being executed quickly enough to empty the buffer (BASIC should be able to keep up at 300 baud, but not higher). If Bit 7 is set, you will want to stop sending, and execute a GET#2 to see what is being sent.

664 $298 BITNUM

RS-232: Number of Bits Left to be Sent/Received

This location is used to determine how many zero bits must be added to the data character to pad its length out to the word length specified in 659 ($293).

665-666 $299-$29A BAUDOF

Time Required to Send a Bit

This location holds the prescaler value used by CIA #2 timers A and B.

These timers cause an NMI interrupt to drive the RS-232 receive and transmit routines CLOCK/PRESCALER times per second each, where CLOCK is the system 02 frequency of 1,022,730 Hz (985,250 if you are using the European PAL television standard rather than the American NTSC standard), and PRESCALER is the value stored at 56580-1 ($DD04-5) and 56582-3 ($DD06-7), in low-byte, high-byte order. You can use the following formula to figure the correct prescaler value for a particular RS-232 baud rate:

PRESCALER=((CLOCK/BAUDRATE)/2)-100

The American (NTSC standard) prescaler values for the standard RS-232 baud rates which the control register at 659 ($293) makes available are stored in a table at 65218 ($FEC2), starting with the two-byte value used for 50 baud. The European (PAL standard) version of that table is located at 58604 ($E4EC).

Location Range: 667-670 ($29B-$29E)

Byte Indices to the Beginning and End of Receive and Transmit Buffers

The two 256-byte First In, First Out (FIFO) buffers for RS-232 data reception and transmission are dynamic wraparound buffers. This means that the starting point and the ending point of the buffer can change over time, and either point can be anywhere within the buffer. If, for example, the starting point is at byte 100, the buffer will fill towards byte 255, at which point it will wrap around to byte 0 again. To maintain this system, the following four locations are used as indices to the starting and the ending point of each buffer.

667 $29B RIDBE

RS-232: Index to End of Receive Buffer

This index points to the ending byte within the 256-byte RS-232 receive buffer, and is used to add data to that buffer.

668 $29C RIDBS

RS-232: Index to Start of Receive Buffer

This index points to the starting byte within the 256-byte RS-232 receive buffer, and is used to remove data from that buffer.

669 $29D RODBS

RS-232: Index to Start of Transmit Buffer

This index points to the starting byte within the 256-byte RS-232 transmit buffer, and is used to remove data from that buffer.

670 $29E RODBE

RS-232: Index to End of Transmit Buffer

This index points to the ending byte within the 256-byte RS-232 transmit buffer, and is used to add data to that buffer.

671-672 $29F-$2A0 IRQTMP

Save Area for IRQ Vector During Cassette I/O

The routines that read and write tape data are driven by an IRQ interrupt. In order to hook one of these routines into the interrupt, the RAM IRQ vector at 788-789 ($314-$315) must be changed to point to the address at which it starts. Before that change is made, the old IRQ vector address is saved at these locations, so that after the tape I/O is finished, the interrupt that is used for scanning the keyboard, checking the stop key, and updating the clock can be restored.

You will note that all of the above functions will be suspended during tape I/O.

673 $2A1 ENABL

RS-232 Interrupts Enabled

This location holds the active NMI interrupt flag byte from CIA #2 Interrupt Control Register (56589, $DD0D). The bit values for this flag are as follows:

Bit 4: 1 (bit value of 16)=System is Waiting for Receiver Edge
Bit 1: 1 (bit value of 2) =System is Receiving Data
Bit 0: 1 (bit value of 1) =System is Transmitting Data

674 $2A2

Indicator of CIA #1 Control Register B Activity During Cassette I/O

675 $2A3

Save Area for CIA #1 Interrupt Control Register During Cassette Read

676 $2A4

Save Area for CIA #1 Control Register A During Cassette Read

677 $2A5

Temporary Index to the Next 40-Column Line for Screen Scrolling

678 $2A6

PAL/NTSC Flag

At power-on, a test is performed to see if the monitor uses the NTSC (North American) or PAL (European) television standard.

This test is accomplished by setting a raster interrupt for scan line 311, and testing if the interrupt occurs. Since NTSC monitors have only 262 raster scan lines per screen, the interrupt will occur only if a PAL monitor is used. The results of that test are stored here, with a 0 indicating an NTSC system in use, and one signifying a PAL system.

This information is used by the routines which set the prescaler values for the system IRQ timer, so that the IRQ occurs every 1/60 second. Since the PAL system 02 clock runs a bit slower than the NTSC version, this prescaler value must be adjusted accordingly.

679-767 $2A7-$2FF

Unused

The programmer may use this area for machine language subroutines, or for graphics data storage.

If the VIC-II chip is using the bottom 16K for graphics memory (the default setting when the system is turned on), this is one of the few free areas available for storing sprite or character data. Locations 704-767 could be used for sprite data block number 11, without interfering with BASIC program text or variables.

Location Range: 768-779 ($300-$30B)

BASIC Indirect Vector Table

Several important BASIC routines are vectored through RAM. This means that the first instruction executed by the routine is an indirect jump to a location pointed to by one of the vectors in this table.

On power-up, the system sets these vectors to point to the next instruction past the original JuMP instruction. The routine then continues with that instruction as if the jump never took place. For example, the BASIC error message routine starts at 42039 ($A437) with the instruction JMP ($300). The indirect vector at 768 ($300) points to 42042 ($A43A), which is the instruction immediately following JMP ($300).

Although this may seem like a fancy way of accomplishing nothing, using these indirect vectors serves two important purposes. First, it allows you to use these important BASIC routines without knowing their address in the BASIC ROM.

For example, the routine to LIST the ASCII text of the single-byte BASIC program token that is currently in the Accumulator (.A) is located at one address in the VIC, and another in the 64. On future Commodore computers it may be found at still another location. Yet as long as the routine is vectored in RAM at 774 ($306), the statement QP=PEEK(774)+256*PEEK(775) would find the address of that routine on any of the machines. Thus, entering such routines through RAM vectors rather than a direct jump into the ROMs helps

to keep your programs compatible with different machines.

The other important effect of having these vectors in RAM is that you can alter them. In that way, you can redirect these important BASIC routines to execute your own preprocessing routines first.

If you wanted to add commands to BASIC, for example, how would you go about it? First, you would need to change the BASIC routines that convert ASCII program text to tokenized program format, so that when a line of program text was entered, the new keyword would be stored as a token.

Next, you would need to change the routine that executes tokens, so that when the interpreter comes to your new keyword token, it will take the proper action.

You would also have to change the routine that converts tokens back to ASCII text, so that your program would LIST the token out correctly. And you might want to alter the routine that prints error messages, to add new messages for your keyword.

As you will see, vectors to all of these routines can be found in the following indirect vector table. Changing these vectors is a much more elegant and efficient solution than the old *wedge* technique discussed at location 115 ($73).

768-769 $300-$301 IERROR

Vector to the Print BASIC Error Message Routine

This vector points to the address of the ERROR routine at 58251 ($E38B).

770-771 $302-$303 IMAIN

Vector to the Main BASIC Program Loop

This vector points to the address of the main BASIC program loop at 42115 ($A483). This is the routine that is operating when you are in the direct mode (READY). It executes statements, or stores them as program lines.

772-773 $304-$305 ICRNCH

Vector to the Routine That Crunches the ASCII Text of Keywords into Tokens

This vector points to the address of the CRUNCH routine at 42364 ($A57C).

774-775 $306-$307 IQPLOP

Vector to the Routine That Lists BASIC Program Token as ASCII Text

This vector points to the address of the QPLOP routine at 42778 ($A71A).

776-777 $308-$309 IGONE

Vector to the Routine That Executes the Next BASIC Program Token

This vector points to the address of the GONE routine at 42980 ($A7E4) that executes the next program token.

778-779 $30A-$30B IEVAL

Vector to the Routine That Evaluates a Single-Term Arithmetic Expression

This vector points to the address of the EVAL routine at 44678 ($AE86) which, among other things, is used to evaluate BASIC functions such as INT and ABS.

Location Range: 780-783 ($30C-$30F)

Register Storage Area

The BASIC SYS command uses this area to store 6510 internal registers—the Accumulator (.A), the .X and Y. index registers, and the status register, .P.

Before every SYS command, each of the registers is loaded with the value found in the corresponding storage address. After the ML program finishes executing, and returns to BASIC with an RTS instruction, the new value of each register is stored in the appropriate storage address. This is true only of SYS, not of the similar USR command.

This feature allows you to place the necessary preentry values into the registers from BASIC before you SYS to a Kernal or BASIC ML routine. It also enables you to examine the resulting effect of the routine on the registers, and to preserve the condition of the registers on exit for subsequent SYS calls.

An extremely practical application comes immediately to mind. Although the 64's BASIC 2 has many commands for formatting printed characters on the monitor screen (for example, TAB, SPC, PRINT A$,B), there are none to adjust the vertical cursor position.

There is a Kernal routine, PLOT (58634, $E50A), which will allow you to position the cursor anywhere on the screen. In order to use it, however, you must first clear the carry flag (set it to 0), and then place the desired horizontal column number in the .Y register and the vertical row number in the .X register before entering the routine with a SYS 65520. Using the register storage area, we can print the word HELLO at row 10, column 5 with the following BASIC line:

```
POKE 781,10:POKE 782,5:POKE 783,0:SYS 65520:PRINT "HELLO"
```

You can also use these locations to help you take advantage of Kernal routines that return information in the register. For example,

the SCREEN routine (58629, $E505) returns the number of screen rows in the .Y register, and the number of columns in the .X register. Using this routine, a BASIC program could be written to run on machines with different screen formats (for example, the 64 and the VIC-20). Just PEEK(781) after a SYS 65517 to see how many screen columns the computer display has.

780 $30C SAREG
Storage Area for .A Register (Accumulator)

781 $30D SXREG
Storage Area for .X Index Register

782 $30E SYREG
Storage Area for .Y Index Register

783 $30F SPREG
Storage Area for .P (Status) Register

The Status (.P) register has seven different flags. Their bit assignments are as follows:

Bit 7 (bit value of 128)	= Negative
Bit 6 (bit value of 64)	= Overflow
Bit 5 (bit value of 32)	= Not Used
Bit 4 (bit value of 16)	= BREAK
Bit 3 (bit value of 8)	= Decimal
Bit 2 (bit value of 4)	= Interrupt Disable
Bit 1 (bit value of 2)	= Zero
Bit 0 (bit value of 1)	= Carry

If you wish to clear any flag before a SYS, it is safe to clear them all with a POKE 783,0. The reverse is not true, however, as you must watch out for the Interrupt disable flag.

A 1 in this flag bit is equal to an SEI instruction, which turns off all IRQ interrupts (like the one that reads the keyboard, for example). Turning off the keyboard could make the computer very difficult to operate! To set all flags except for Interrupt disable to 1, POKE 783,251.

784 $310 USRPOK
Jump Instruction for User Function ($4C)

The value here (76, $4C) is first part of the 6510 machine langauge JuMP instruction for the USR command.

785-786 $311-$312 USRADD

Address of USR Routine (Low Byte First)

These locations contain the target address of the USR command. They are initialized by the Operating System to point to the BASIC error message handler routine, so that if you try to execute a USR call without changing these values, you will receive an ILLEGAL QUANTITY error message.

In order to successfully execute a USR call, you must first POKE in the target address in low-byte, high-byte order. You can calculate these two values for any address with the formula:

HI=INT(AD/256):LO=AD-(HI*256)

For example, if the USR routine started at 49152 ($C000), you would POKE 786,INT(49152/256):POKE 785,49152-(PEEK(786)*256) before executing the USR command.

What makes the USR command different from SYS is that you can pass a parameter into the machine language routine by placing it in parentheses after the USR keyword, and you can pass a parameter back to a variable by assigning its value to the USR function.

In other words, the statement X=USR(50) will first put the number 50 in floating point format into the Floating Point Accumulator (FAC1) at 97–102 ($61–$66). Then, the machine language program designated by the address at this vector will be executed. Finally, the variable X will be assigned the floating point value which ends up in FAC1 after the user-written routine is finished.

Since floating point representation is difficult to work with, it is handy to change these floating point parameters to integers before working with them. Fortunately, there are vectored routines which will do the conversions for you. The routine vectored at locations 3-4 converts the number in FAC1 to a two-byte signed integer, with the low byte in the .Y register (and location 101 ($65)) and the high byte in the Accumulator (.A). Remember, that number is converted to a signed integer in the range between 32767 and -32768, with Bit 7 of the high byte used to indicate the sign.

To pass a value back through the USR function, you need to place the number into FAC1. To convert a singed integer to floating point format, place the high byte into the Accumulator (.A), the low byte into the .Y register, and jump through the vector at location 5–6 with a JMP ($0005) instruction. The floating point result will be left in FAC1.

787 $313

Unused

788-789 $314-$315 CINV

Vector to IRQ Interrupt Routine

This vector points to the address of the routine that is executed when an IRQ interrupt occurs (normally 59953 ($EA31)).

At power-on, the CIA #1 Timer B is set to cause an IRQ interrupt to occur every 1/60 second. This vector is set to point to the routine which updates the software clock and STOP key check, blinks the cursor, maintains the tape interlock, and reads the keyboard. By changing this vector, the user can add or substitute a machine language routine that will likewise execute every 1/60 second. The user who is writing IRQ interrupt routines should consider the following:

1. It is possible for an IRQ interrupt to occur while you are changing this vector, which would cause an error from which no recovery could be made. Therefore, you must disable all IRQ interrupts before changing the contents of this location, and reenable them afterwards, by using the 6510 SEI and CLI instructions, or by using the Kernal VECTOR routine (64794, $FD1A) to set this vector.

2. There is some code in ROM that is executed before the interrupt routine is directed through this vector. This code checks whether the source of the interrupt was an IRQ or the BRK instruction. It first preserves the contents of all the registers by pushing them onto the stack in the following sequence: PHA, TXA, PHA, TYA, PHA. It is up to the user to restore the stack at the end of his routine, either by exiting through the normal IRQ, or with the sequence: PLA, TAY, PLA, TAX, PLA, RTI.

3. There is only one IRQ vector, but there are many sources for IRQ interrupts (two CIA chips, and several VIC chip IRQs). If you plan to enable IRQs from more than one source, the IRQ routine here must determine the source, and continue the routine in the appropriate place for an IRQ from that source.

In the same vein, if you replace the normal IRQ routine with your own, you should be aware that the keyboard's scanning and clock update will not occur unless you call the old interrupt routine once every 1/60 second. It is suggested that if you plan to use that routine, you save the old vector address in some other location. In that way, you can JuMP to the keyboard interrupt routine through this alternate vector, rather than assuming that the ROM address will never change and that it is safe to jump into the ROM directly.

790-791 $316-$317 CBINV

Vector: BRK Instruction Interrupt

This vector points to the address of the routine which will be executed anytime that a 6510 BRK instruction (00) is encountered.

The default value points to a routine that calls several of the

Kernal initialization routines such as RESTOR, IOINIT and part of CINT, and then jumps though the BASIC warm start vector at 40962. This is the same routine that is used when the STOP and RESTORE keys are pressed simulatneously, and is currently located at 65126 ($FE66).

A machine language monitor program will usually change this vector to point to the monitor warm start address, so that breakpoints may be set that will return control to the monitor for debugging purposes.

792-793 $318-$319 NMINV

Vector: Non-Maskable Interrupt

This vector points to the address of the routine that will be executed when a Non-Maskable Interrupt (NMI) occurs (currently at 65095, $FE47).

There are two possible sources for an NMI interrupt. The first is the RESTORE key, which is connected directly to the 6510 NMI line. The second is CIA #2, the interrupt line of which is connected to the 6510 NMI line.

. When an NMI interrupt occurs, a ROM routine sets the Interrupt disable flag, and then jumps through this RAM vector. The default vector points to an interrupt routine which checks to see what the cause of the NMI was.

If the cause was CIA #2, the routine checks to see if one of the RS-232 routines should be called. If the source was the RESTORE key, it checks for a cartridge, and if present, the cartridge is entered at the warm start entry point. If there is no cartridge, the STOP key is tested. If the STOP key was pressed at the same time as the RESTORE key, several of the Kernal initialization routines such as RESTOR, IONIT and part of CINT are executed, and BASIC is entered through its warm start vector at 40962. If the STOP key was not pressed simultaneously with RESTORE, the interrupt will end without letting the user know that anything happened at all when the RESTORE key was pressed.

Since this vector controls the outcome of pressing the RESTORE key, it can be used to disable the STOP/RESTORE sequence. A simple way to do this is to change this vector to point to the RTI instruction. A simple POKE 792,193 will accomplish this. To set the vector back, POKE 792,71. Note that this will cut out all NMIs, including those required for RS-232 I/O.

Location Range: 794-813 ($31A-$32D)

Kernal Indirect Vectors

There are 39 Kernal routines for which there are vectors in the jump table located at the top of the ROM (65409, $FF81). For ten of these

routines, the jump table entry contains a machine language instruction to jump to the address pointed to by the RAM vector in this table. The addresses in this table are initialized to point to the corresponding routines in the Kernal ROM. Since these addresses are in RAM, however, any entry in this table may be changed. This enables the user to add to these routines, or to replace them completely.

You will notice, for example, that many of these routines involve Input/Output functions. By changing the vectors to these routines, it is possible to support new I/O devices, such as an IEEE disk drive used through an adapter.

The user should be cautioned that since some of these routines are interrupt-driven, it is dangerous to change these vectors without first turning off all interrupts. For a safe method of changing all of these vectors at one time, along with the interrupt vectors above, see the entry for the Kernal VECTOR routine at 64794 ($FD1A).

More specific information about the individual routines can be found in the descriptions given for their ROM locations.

794-795 $31A-$31B IOPEN
Vector to Kernal OPEN Routine (Currently at 62282, $F34A)

796-797 $31C-$31D ICLOSE
Vector to Kernal CLOSE Routine (Currently at 62097, $F291)

798-799 $31E-$31F ICHKIN
Vector to Kernal CHKIN Routine (Currently at 61966, $F20E)

800-801 $320-$321 ICKOUT
Vector to Kernal CKOUT Routine (Currently at 62032, $F250)

802-803 $322-$323 ICLRCH
Vector to Kernal CLRCHN Routine (Currently at 62259, $F333)

804-805 $324-$325 IBASIN
Vector to Kernal CHRIN Routine (Currently at 61783, $F157)

806-807 $326-$327 IBSOUT
Vector to Kernal CHROUT Routine (Currently at 61898, $F1CA)

808-809 $328-$329 ISTOP
Vector to Kernal STOP Routine (Currently at 63213, $F6ED)
This vector points to the address of the routine that tests the STOP key. The STOP key can be disabled by changing this with a POKE 808,239. This will not disable the STOP/RESTORE combination, however. To disable both STOP and STOP/RESTORE, POKE

808,234 (POKEing 234 here will cause the LIST command not to function properly). To bring things back to normal in either case, POKE 808,237.

810-811 $32A-$32B IGETIN

Vector to Kernal GETIN Routine (Currently at 61758, $F13E)

812-813 $32C-$32D ICLALL

Vector to Kernal CLALL Routine (Currently at 62255, $F32F)

814-815 $32E-$32F USRCMD

Vector to User-Defined Command (Currently Points to BRK at 65126, $FE66)

This appears to be a holdover from PET days, when the built-in machine language monitor would JuMP through the USRCMD vector when it encountered a command that it did not understand, allowing the user to add new commands to the monitor.

Although this vector is initialized to point to the routine called by STOP/RESTORE and the BRK interrupt, and is updated by the Kernal VECTOR routine (64794, $FD1A), it does not seem to have the function of aiding in the addition of new commands.

816-817 $330-$331 ILOAD

Vector to Kernal LOAD Routine (Currently at 62622, $F49E)

818-819 $332-$333 ISAVE

Vector: Kernal SAVE Routine (Currently at 62941, $F5DD)

820-827 $334-$33B

Unused

Eight free bytes for user vectors or other data.

828-1019 $33C-$3FB TBUFFER

Cassette I/O Buffer

This 192-byte buffer area is used to temporarily hold data that is read from or written to the tape device (device number 1).

When not being used for tape I/O, the cassette buffer has long been a favorite place for Commodore programmers to place short machine language routines (although the 64 has 4K of unused RAM above the BASIC ROM at 49152 ($C000) that would probably better serve the purpose).

Of more practical interest to the 64 programmer is the possible use of this area for VIC-II chip graphics memory (for example, sprite shape data or text character dot data). If the VIC-II chip is banked to the lowest 16K of memory (as is the default selection), there is very little memory space which can be used for such things as sprite

shape data without conflict. If the tape is not in use, locations 832-895 ($340-$37F) can be used as sprite data block number 13, and locations 896-959 ($380-$3BF) can be used as sprite data block number 14.

The types of tape blocks that can be stored here are program header blocks, data header blocks, and data storage blocks.

The first byte of any kind of block (which is stored at location 828, $33C) identifies the block type. Header blocks follow this identifier byte with the two-byte starting RAM address of the tape data, the two-byte ending RAM address, and the filename, padded with blanks so that the total length of the name portion equals 187 bytes. Data storage blocks have 191 bytes of data following the identifier byte. The meanings of the various identifier blocks are as follows:

A value of 1 signifies that the block is the header for a relocatable program file, while a value of 3 indicates that the block is the header for a nonrelocatable program file.

A relocatable file is created when a program is SAVEd with a secondary address of 0 (or any even number), while a nonrelocatable program file is created if the secondary SAVE address is 1 (or any odd number). The difference between the two types of files is that a nonrelocatable program will always load at the address specified in the header. A relocatable program will load at the current start-of-BASIC address unless the LOAD statement uses a secondary address of 1, in which case it will also be loaded at the address specified in the header.

You should note that a program file uses the cassette buffer only to store the header block. Actual program data is transferred directly to or from RAM, without first being buffered.

An identifier value of 4 means that the block is a data file header. Such a header block is stored in the cassette buffer whenever a BASIC program OPENs a tape data file for reading or writing. Subsequent data blocks start with an identifier byte of 2. These blocks contain the actual data byte written by the PRINT #1 command, and read by the GET #1 and INPUT #1 commands. Unlike the body of a program file, these blocks are temporarily stored in the cassette buffer when being written or read.

An identifier byte of 5 indicates that this block is the logical end of the tape. This signals the Kernal not to search past this point, even if there are additional tape blocks physically present on the tape.

1020-1023 $3FC-$3FF

Unused

Four more free bytes.

Chapter 4

1K to 40K
Screen Memory, Sprite Pointers, and BASIC Program Text

1K to 40K
Screen Memory, Sprite Pointers, and BASIC Program Text

1024-2047 $400-$7FF VICSCN
Video Screen Memory Area

This is the default location of the video screen memory area, which contains the video matrix and the sprite data pointers. Keep in mind, however, that the video screen memory area can be relocated to start on any even 1K boundary. Its location at any given moment is determined by the VIC-II chip memory control register at 53272 ($D018), and the VIC-II memory bank select bits on CIA #2 Data Port A (56576, $DD00).

1024-2023 $400-$7E7
Video Matrix: 25 Lines by 40 Columns

The video matrix is where text screen characters are stored in RAM. Normally, the VIC-II chip will treat each byte of memory here as a screen display code and will display the text character that corresponds to that byte of code. The first byte of memory here will be displayed in the top-left corner of the screen, and subsequent bytes will be displayed in the columns to the right and the rows below that character.

It is possible to make text or graphics characters appear on the screen by POKEing their screen codes directly into this area of RAM. For example, the letter A has a screen code value of 1. Therefore, POKE 1024,1 should make the letter A appear in the top-left corner of the screen.

However, you should be aware that the most current version of the Operating System initializes the color RAM which is used for the foreground color of text characters to the same value as the background color every time that the screen is cleared. The result is that although the POKE will put a blue A on the screen, you won't be able to see it because it is the same color blue as the background. This can be remedied by POKEing a different value into color RAM (which starts at 55296, $D800).

A POKE 1024,1:POKE 1024+54272,1 will put a white A in the upper-left corner of the screen. The loop

FOR I=0 TO 255:POKE 1024+I,I:POKE 1024+54272+I,1:NEXT

will display all of the characters in white at the top of the screen. Another solution to the color RAM problem is to fool the Operating System into initializing the color RAM for you. If you change the background color to the desired foreground color before you clear the screen, color RAM will be set to that color. Then, all you have to do is change the background color back to what it was. This example will show how it's done:

```
10 POKE 53281,2:REM BACKGROUND IS RED
20 PRINT CHR$(147):REM CLEAR SCREEN
30 POKE 53281,1:REM BACKGROUND IS WHITE
40 POKE 1024,1:REM RED "A" APPEARS IN TOP LEFT CORNER
```

2040-2047 $7F8-$7FF

Sprite Shape Data Pointers

The last eight bytes of the video matrix (whether it is here at the default location, or has been relocated elsewhere) are used as pointers to the data blocks used to define the sprite shapes.

Each sprite is 3 bytes wide (24 bits) by 21 lines high. It therefore requires 63 bytes for its shape definition, but it actually uses 64 bytes in order to arrive at an even 256 shape blocks in the 16K area of RAM which the VIC-II chip addresses.

Each pointer holds the current data block being used to define the shape of one sprite. The block number used to define the shape of Sprite 0 is held in location 2040 ($7F8), the Sprite 1 shape block is designated by location 2041 ($7F9), etc. The value in the pointer times 64 equals the starting location of the sprite shape data table. For example, a value of 11 in location 2040 indicates that the shape data for Sprite 0 starts at address 704 (11*64), and continues for 63 more bytes to 767.

For additional information on sprite graphics, see the entries for individual VIC-II chip sprite graphics locations, and the summary at the beginning of the VIC-II chip section, at 53248 ($D000).

2048-40959 $800-$9FFF

BASIC Program Text

This is the area where the actual BASIC program text is stored. The text of a BASIC program consists of linked lines of program tokens. Each line contains the following:

1. A two-byte pointer to the address of the next program line (in standard low-byte, high-byte order). After the last program line, a link pointer consisting of two zeros marks the end of the program.

2. A two-byte line number (also in low-byte, high-byte order).

3. The program commands. Each keyword is stored as a one-byte character whose value is equal to or greater than 128. PRINT, for example, is stored as the number 151. Other elements of the BASIC command, such as variable names, string literals ("HELLO"), and numbers, are stored using their ASCII text equivalents.

4. A 0 character, which acts as a line terminator. In order for BASIC to work correctly, the first character of the BASIC text area must be 0.

A quick review of the BASIC pointers starting at 43 ($2B) reveals that the layout of the BASIC program area (going from lower memory addresses to higher) is as follows:

BASIC Program Text
Non-Array Variables and String Descriptors
Array Variables
Free Area (Reported by FRE(0))
String Text Area (Strings build from top of memory down into free area)
BASIC ROM

It is interesting to note that the NEW command does not zero out the text area, but rather replaces the first link address in the BASIC program with two zeros, indicating the end of the program. Therefore, you can recover a program from a NEW by replacing the first link address, finding the address of the two zeros that actually mark the end of the program, and setting the pointers at 45, 47, and 49 (which all point to the end of a BASIC program before the program is RUN) to the byte following those zeros.

4096-8191 $1000-$1FFF

Character ROM Image for VIC-II Chip When Using Memory Bank 0 (Default)

Though the VIC-II chip shares memory with the 6510 processor chip, it does not always see that memory in exactly the same way as the main microprocessor.

Although the 6510 accesses RAM at these locations, when the VIC-II is banked to use the first 16K of RAM (which is the default condition), it sees the character ROM here (the 6510 cannot access this ROM unless it is switched into its memory at 49152 ($C000)). This solves the riddle of how the VIC-II chip can use the character ROM at 49152 ($C000) for character shape data and RAM at 1024 ($400), when it can only address memory within a 16K range. It also means that the RAM at 4096-8192 cannot be used for screen display memory or user-defined character dot data, and sprite data blocks 64-127 are not accessible.

You can verify this by turning on bitmap graphics with the screen memory set to display addresses from 0 to 8192. You will see that the bottom portion of the screen shows all of the text character shapes stored in the ROM. For more information on the format of text character data storage, see the description of the Character ROM at 53248 ($D000).

32768 $8000

Autostart ROM Cartridge

An 8K or 16K autostart ROM cartridge designed to use this as a starting memory address may be plugged into the Expansion Port on the back. If the cartridge ROM at locations 32772-32776 ($8004-$8008) contains the numbers 195, 194, 205, 56, 48 ($C3, $C2, $CD, $38, $30) when the computer powers up, it will start the program pointed to by the vector at locations 32768-32769 ($8000-$8001), and will use 32770-32771 ($8002-$8003) for a warm start vector when the RESTORE key is pressed. These characters are PETASCII for the inverse letters CBM, followed by the digits 80. An autostart cartridge may also be addressed at 40960 ($A000), where it would replace BASIC, or at 61440 ($F000), where it would replace the Kernal.

It is possible to have a 16K cartridge sitting at 32768 ($8000), such as Simon's BASIC, which can be turned on and off so that the BASIC ROM underneath can also be used. Finally, it is even possible to have bank-selected cartridges, which turn banks of memory in the cartridge on and off alternately, so that a 32K program could fit into only 16K of addressing space.

36864-40959 $9000-$9FFF

Character ROM Image for VIC-II Chip When Using Memory Bank 2

When the VIC-II chip is set up to use the third 16K block of memory for graphics (as would be the case when the 64 is set up to emulate the PET, which has its text screen memory at 32768, $8000, it sees the character generator ROM at this address (see entry at 4096 ($1000) above for more details).

It should be noted that the character ROM is available only when the VIC-II chip is using banks 0 or 2. When using one of the other two banks, the user must supply all of the character shape data in a RAM table.

Chapter 5

8K BASIC ROM and 4K Free RAM

8K BASIC ROM and 4K Free RAM

Locations 40960 to 49151 ($A000 to $BFFF) are used by the BASIC ROM when it is selected (which is the default condition). BASIC is the 64's main program, which is always run if there is no autostart cartridge inserted at power-up time. When the 64 tells you READY, that's BASIC talking.

The BASIC interpreter that comes with the 64 is, aside from being located in a different memory space, almost identical to the Microsoft BASIC interpreter found on the VIC-20. Both of these interpreters are slightly modified versions of PET BASIC 2.0, also known as PET BASIC 3.0 or Upgrade BASIC, because it was an upgraded version of the BASIC found on the original PET.

This is a somewhat mixed blessing, because while PET BASIC was, in its day, quite an advanced language for use with an eight-bit microprocessor, it lacks several of the features (such as error trapping) which are now standard on most home computers. And, of course, it makes no provision whatever for easy use of the many graphics and sound capabilities made available by the new dedicated video and sound support chips.

On the other hand, its faithfulness to the original Commodore BASIC allows a large body of software to be translated for the 64 with little change (in most cases, the PET Emulator program from Commodore will allow you to run PET programs with no changes). Programming aids and tricks developed for the PET and VIC will, for the most part, carry over quite nicely to the 64. Although there is no official source code listing of the ROM available from Commodore, this version of BASIC has been around long enough that it has been thoroughly disassembled, dissected, and documented by PET users.

The labels used here correspond to those used by Jim Butterfield in his PET memory maps, which are well-known among PET BASIC users. They should, therefore, provide some assistance in locating equivalent routines on the two machines. A good description of the

workings of PET BASIC can be found in *Programming the PET/CBM* by Raeto West.

It is beyond the scope of this book to detail the inner workings of each routine in the BASIC interpreter. However, the following summary of routines and their functions should aid the user who is interested in calling BASIC routines from his or her own program, or in modifying the BASIC.

Please keep in mind that the entry and exit points listed for routines that perform a particular function are to be used as guideposts, and not absolutes. In fact, BASIC enters many of these routines from slightly different places to accomplish different tasks. Some subroutines are called by so many commands that it is hard to say which they belong to. You will even find that some whole commands are part of other commands. Where it is important for you to know the details of a particular routine, you will need to obtain a disassembly of that section and look at the machine language program itself.

It should be noted that when BASIC is not needed, it can be switched out and replaced by a machine language program in RAM, by an 8K ROM cartridge, or by the last half of a 16K ROM cartridge that starts at 32768 ($8000). See location 1 for more information on the different memory configurations.

Also, it should be remembered that even when the BASIC ROM is switched in, the RAM underneath can be accessed by the VIC-II chip and used for screen graphics. See location 56576 ($DD00) for more information.

40960-40961 $A000-$A001

Cold Start Vector

This vector points to the address of the routine used to initialize BASIC. After the Operating System finishes its power-on activities, it enters the BASIC program through this vector. The most visible effect of the BASIC initialization routine is that the screen is cleared, and the words

**** COMMODORE 64 BASIC V2 ****

are printed along with the BYTES FREE message. For details of the steps taken during the initialization of BASIC, see the entry for 58260 ($E394), the current cold start entry point.

40962-40963 $A002-$A003

Warm Start Vector

The warm start vector points to the address of the routine used to reset BASIC after the STOP/RESTORE key combination is pressed. This is the same address to which the BRK instruction is vectored.

When BASIC is entered through this vector, the program in memory is not disturbed. For more information, see the entry for 58235 ($E37B), the current warm start entry point.

40964-40971 $A004-$A00B

ASCII Text Characters CBMBASIC

The ASCII characters for the letters CBMBASIC are located here. Possibly an identifier, this text is not referenced elsewhere in the program.

40972-41041 $A00C-$A051 STMDSP

Statement Dispatch Vector Table

This table contains two-byte vectors, each of which points to an address which is one byte before the address of one of the routines that perform a BASIC statement.

The statements are in token number order. When it comes time to execute a statement, the NEWSTT routine at 42926 ($A7AE) places this address–1 on the stack and jumps to the CHRGET routine. The RTS instruction at the end of that routine causes the statement address to be pulled off the stack, incremented, and placed in the Program Counter, just as if it were the actual return address.

This table is handy for locating the address of the routine that performs a BASIC statement, so that the routine can be disassembled and examined. To aid in this purpose, the table is reproduced below with the actual target addresses, and not in the address–1 format used by BASIC.

Token #	Statement	Routine Address
128 $80	END	43057 $A831
129 $81	FOR	42818 $A742
130 $82	NEXT	44318 $AD1E
131 $83	DATA	43256 $A8F8
132 $84	INPUT#	43941 $ABA5
133 $85	INPUT	43967 $ABBF
134 $86	DIM	45185 $BO81
135 $87	READ	44038 $ACO6
136 $88	LET	43429 $A9A5
137 $89	GOTO	43168 $A8A0
138 $8A	RUN	43121 $A871
139 $8B	IF	43304 $A928
140 $8C	RESTORE	43037 $A8D
141 $8D	GOSUB	43139 $A883
142 $8E	RETURN	43218 $A8D2
143 $8F	REM	43323 $A93B
144 $90	STOP	43055 $A82F
145 $92	ON	43339 $A94B

Token #	Statement	Routine Address
146 $92	WAIT	47149 $B82D
147 $93	LOAD	57704 $E168
148 $94	SAVE	57686 $E156
149 $95	VERIFY	57701 $E165
150 $96	DEF	46003 $B3B3
151 $97	POKE	47140 $B824
152 $98	PRINT#	43648 $AA80
153 $99	PRINT	43680 $AAA0
154 $9A	CONT	43095 $A857
155 $9B	LIST	42652 $A69C
156 $9C	CLR	42590 $A65E
157 $9D	CMD	43654 $AA86
158 $9E	SYS	57642 $E12A
159 $9F	OPEN	57790 $E1BE
160 $A0	CLOSE	57799 $E1C7
161 $A1	GET	43899 $AB7B
162 $A2	NEW	42562 $A642

41042-41087 $A052-$A07F FUNDSP TABLE

Function Dispatch Vector Table

This table contains two-byte vectors, each of which points to the address of one of the routines that perform a BASIC function.

A function is distinguished by a following argument, in parentheses. The expression in the parentheses is first evaluated by the routines which begin at 44446 ($AD9E). Then this table is used to find the address of the function that corresponds to the token number of the function to be executed.

The substance of this table, which can be used for locating the addresses of these routines, is reproduced below. Note that the address for the USR function is 784 ($310), which is the address of the JMP instruction which precedes the user-supplied vector.

Token #	Function	Routine Address
180 $B4	SGN	48185 $BC39
181 $B5	INT	48332 $BCCC
182 $B6	ABS	48216 $BC58
183 $B7	USR	784 $0310
184 $B8	FRE	45949 $B37D
185 $B9	POS	45982 $B39E
186 $BA	SQR	49009 $BF71
187 $BB	RND	57495 $E097
188 $BC	LOG	47594 $B9EA
189 $BD	EXP	49133 $BFED
190 $BE	COS	57956 $E264

191 $BF	SIN	57963 $E26B
192 $C0	TAN	58036 $E2B4
193 $C1	ATN	58126 $E30E
194 $C2	PEEK	47117 $B80D
195 $C3	LEN	46972 $B77C
196 $C4	STR$	46181 $B465
197 $C5	VAL	47021 $B7AD
198 $C6	ASC	46987 $B78B
199 $C7	CHR$	46828 $B6EC
200 $C8	LEFT$	46848 $B700
201 $C9	RIGHT$	46892 $B72C
202 $CA	MID$	46903 $B737

41088-41117 $A080-$A09D OPTAB

Operator Dispatch Vector Table

This table contains two-byte vectors, each of which points to an address which is one byte before the address of one of the routines that perform a BASIC math operation.

For the reasoning behind the one-byte offset to the true address, see the entry for location 40972 ($A00C). In addition, each entry has a one-byte number which indicates the degree of precedence that operation takes. Operations with a higher degree of precedence are performed before operations of a lower degree (for example, in the expression A=3+4*6, the 4*6 operation is performed first, and 3 is added to the total). The order in which they are performed is:

1. Expressions in parentheses
2. Exponentiation (raising to a power, using the up-arrow symbol)
3. Negation of an expression (–5,–A)
4. Multiplication and division
5. Addition and subtraction
6. Relation tests (=, <>, <, >, <=, >= all have the same precedence)
7. NOT (logical operation)
8. AND (logical operation)
9. OR (logical operation)

The substance of this table, which can be used to locate the addresses of the math routines, is given below. Note that the less than, equal, and greater than operators all use the same routines, though they have different token numbers.

Token #	Operator	Routine Address
170 $AA	+ (ADD)	47210 $B86A
171 $AB	–(SUBTRACT)	47187 $B853
172 $AC	* (MULTIPLY)	47659 $BA2B

Token #	Operator	Routine Address
173 $AD	/ (DIVIDE)	47890 $BB12
174 $AE	↑ (EXPONENTIATE)	49019 $BF7B
175 $AF	AND (LOGICAL AND)	45033 $AFE9
176 $B0	OR (LOGICAL OR)	45030 $AFE6
177 $B1	> (GREATER THAN)	49076 $BFB4
178 $B2	= (EQUAL TO)	44756 $AED4
179 $B3	< (LESS THAN)	45078 $B016

41118-41373 $A09E-$A19D RESLST

List of Keywords

This table contains a complete list of the reserved BASIC keywords (those combinations of ASCII text characters that cause BASIC to do something). The ASCII text characters of these words are stored in token number order. Bit 7 of the last letter of each word is set to indicate the end of the word (the last letter has 128 added to its true ASCII value).

When the BASIC program text is stored, this list of words is used to reduce any keywords to a single-byte value called a token. The command PRINT, for example, is not stored in a program as five ASCII bytes, but rather as the single token 153 ($99).

When the BASIC program is listed, this table is used to convert these tokens back to ASCII text. The entries in this table consist of the following:

1. The statements found in STMDSP at 40972 ($A00C), in the token number order indicated (token numbers 128-162).
2. Some miscellaneous keywords which never begin a BASIC statement:

Token #	Keyword
163 $A3	TAB(
164 $A4	TO
165 $A5	FN
166 $A6	SPC(
167 $A7	THEN
168 $A8	NOT
169 $A9	STEP

3. The math operators found in OPTAB at 41088 ($A080), in the token number order indicated (token numbers 170-179).
4. The functions found in FUNDSP at 41042 ($A052), in the token number order indicated (token numbers 180-202).
5. The word GO (token number 203, $CB). This word was added to the table to make the statement GO TO legal, to afford some compatibility with the very first PET BASIC, which allowed spaces within keywords.

41374-41767 $A19E-$A327 ERRTAB

ASCII Text of BASIC Error Messages

This table contains the ASCII text of all of the BASIC error messages. As in the keyword table, Bit 7 of the last letter of each message is set to indicate the end of the message. Although we've all seen some of them at one time or another, it's somewhat daunting to see the whole list at once. The possible errors you can make include:

1. TOO MANY FILES
2. FILE OPEN
3. FILE NOT OPEN
4. FILE NOT FOUND
5. DEVICE NOT PRESENT
6. NOT INPUT FILE
7. NOT OUTPUT FILE
8. MISSING FILENAME
9. ILLEGAL DEVICE NUMBER
10. NEXT WITHOUT FOR
11. SYNTAX
12. RETURN WITHOUT GOSUB
13. OUT OF DATA
14. ILLEGAL QUANTITY
15. OVERFLOW
16. OUT OF MEMORY
17. UNDEF'D STATEMENT
18. BAD SUBSCRIPT
19. REDIM'D ARRAY
20. DIVISION BY ZERO
21. ILLEGAL DIRECT
22. TYPE MISMATCH
23. STRING TOO LONG
24. FILE DATA
25. FORMULA TOO COMPLEX
26. CAN'T CONTINUE
27. UNDEF'D FUNCTION
28. VERIFY
29. LOAD

Message number 30, BREAK, is located in the Miscellaneous Messages table below.

41768-41828 $A328-$A364

Error Message Vector Table

This table contains the two-byte address of the first letter of each of the 30 error messages.

41829-41865 $A365-$A389

Miscellaneous Messages

The text of some of the other messages that BASIC can give you is stored here. This text includes cursor movement characters, and each message ends with a 0 character. The messages are:

1) Carriage return, OK, carriage return
2) Space, space, ERROR
3) Space, IN, space
4) Carriage return, linefeed, READY., carriage return, linefeed
5) Carriage return, linefeed, BREAK

41866-41911 $A38A-$A3B7 FNDFOR

Find FOR on Stack

This routine searches the stack for the blocks of data entries which are stored by each FOR command. For more information on the data that FOR places on the stack, see location 256 ($100).

41912 $A3B8 BLTU

Open a Space in Memory for a New Program Line or Variable

When a new nonarray variable is being created, or when a BASIC program line is being added or replaced, this routine is used to make room for the addition. It first checks to see if space is available, and then moves the program text and/or variables to make room.

41979-41991 $A3FB-$A407 GETSTK

Check for Space on Stack

Before undertaking an operation that requires stack space, this routine is used to check if there is enough room on the stack. If there is not, an OUT OF MEMORY error is issued.

41992-42036 $A408-$A434 REASON

Check for Space in Memory

This is the subroutine that checks to see if there is enough space in free memory for proposed additions such as new lines of program text. If not, it calls for garbage collection, and if this still does not produce enough space, an OUT OF MEMORY error is issued.

42037-42088 $A435-$A468 OMERR

OUT OF MEMORY Error Handler

This routine just sets the error message code, and falls through to the general error handler.

42039-42088 $A437-$A468 ERROR

General Error Handler

The error number is passed to this routine in the .X register, and it displays the appropriate error message. Since this routine is vectored through RAM at 768 ($300), you can divert this vector to the address of your own routine, which would allow error trapping, or the addition of new commands.

42089-42099 $A469-$A473

Display ERROR or Other Message

This portion of the routine tacks on the word ERROR to the message. This entry point is also used to print BREAK.

42100-42111 $A474-$A47F READY

Print READY

This routine displays the word READY, sets the Kernal message flag to show that direct mode is operative, and falls through to the main BASIC loop.

42112-42139 $A480-$A49B MAIN

Main Loop, Receives Input and Executes Immediately or Stores as Program Line

This is the main BASIC program loop. It jumps through the RAM vector at 770 ($302), so this routine can be diverted. The routine gets a line of input from the keyboard, and checks for a line number. If there is a line number, the program branches to the routine that stores a line of program text. If there is no line number, it branches to the routine that executes statements.

42140 $A49C MAIN1

Add or Replace a Line of Program Text

This routine calls subroutines to get the line number, tokenize keywords, and then looks for a line with the same number.

If it finds a line with the same number, the routine deletes that line by moving all higher program text and variables down to where it started. The new line is then added. Since the CLR routine is called, the value of all current program variables is lost.

42291 $A533 LINKPRG

Relink Lines of Tokenized Program Text

Each line of program text starts with a pointer to the address of the next line (link address). This routine scans each line to the end (which is marked with a 0), and calculates the new link address by adding the offset to the address of the current statement.

42336 $A560 INLIN

Input a Line to Buffer from Keyboard

This subroutine calls the Kernal CHRIN routine (61783, $F157) to obtain a line of input from the current input device (usually the keyboard). It stores the characters in the BASIC text input buffer at 512 ($200) until a carriage return or 89 characters have been received. The keyboard device will never return more than 80 characters before a carriage return, but other devices can output a longer line. An error will occur if the line goes over 88 characters.

42361 $A579 CRUNCH

Tokenize Line in Input Buffer

When a line of program text has been input into the BASIC text buffer at 512 ($200), this routine goes through the line and changes any keywords or their abbreviations, which do not appear in quotes, into the corresponding token. This command is vectored through RAM at 772 ($304), so it can be diverted in order to add new commands.

42515 $A613 FNDLIN

Search for Line Number

This routine searches through the program text, trying to match the two-byte integer line number that is stored in 20-21 ($14-$15). If it is found, 95-96 ($5F-$60) will be set as a pointer to the address of the link field for that line, and the Carry flag will be set. If it is not found, the Carry flag will be cleared.

42562 $A642 SCRTCH

Perform NEW

The NEW command stores two zeros in the link address of the first program line to indicate the end of program, and sets the end of program pointer at 45-46 ($2D-$2E) to point to the byte past those zeros. It continues through to the CLR command code.

42590 $A65E CLEAR

Perform CLR

The CLR command closes all I/O files with the Kernal CLALL routine (62255, $F32F). It eliminates string variables by copying the end of memory pointer at 55-56 ($37-$38) to the bottom of strings pointer at 51-52 ($33-$34). It also copies the pointer to end of BASIC program text at 49-50 ($31-$32) to the pointer to the start of nonarray variables at 45-46 ($2D-$2E), the start of array variables at 47-48 ($2F-$30), and the end of array variables at 49-50 ($31-$32). This makes these variables unusable (although the contents of these areas are not actually erased). RESTORE is called to set the data pointer back to the beginning, and the stack is cleared.

42638 $A68E RUNC

Reset Pointer to Current Text Character to the Beginning of Program Text

This routine resets the CHRGET pointer TXTPTR (122–123, $7A–$7B) so that the next byte of text that the interpreter will read comes from the beginning of program text.

42652 $A69C LIST

Perform LIST

This routine saves the range of lines to be printed in pointers at 95-96 ($5F- $60) and 20-21 ($14-$15), and then prints them out, translating any tokens back to their ASCII equivalent.

42775 $A717 QPLOP

Print BASIC Tokens as ASCII Characters

This is the part of the LIST routine that changes one-byte program tokens back to their ASCII text characters. The routine is vectored through RAM at 774 ($306), so it is possible to list out new command words that you have added by changing this vector to detour through your own routine.

42818 $A742 FOR

Perform FOR

FOR is performed mostly by saving the needed information for the NEXT part of the command on the stack (see the entry for 256 ($100) for details). This includes the TO termination value, so if the upper limit is a variable, the current value of the variable will be stored, and you cannot end the loop early by decreasing the value of the TO variable within the loop (although you can end it early by increasing the value of the FOR variable within the loop).

Also, since the TO expression is evaluated only once, at the time FOR is performed, a statement such as FOR I=1 TO I+100 is valid. The terminating value is not checked until NEXT is executed, so the loop statements always execute at least once. The variable used by FOR must be a nonarray floating point variable. Reusing the same FOR variable used in a loop that is still active will cause the previous FOR loop and all intervening loops to be cancelled.

42926 $A7AE NEWSTT

Set Up Next Statement for Execution

This routine tests for the STOP key, updates the pointer to the current line number, and positions the text pointer to read the beginning of the statement.

42980 $A7E4 GONE

Read and Execute Next Statement

This is the routine which gets the next token and executes the statement. It is vectored through RAM at 776 ($308) to allow the addition and execution of new statement tokens.

Since a statement must always start with a token or an implied LET statement, this routine checks to see if the first character is a valid token. If it is, the address is placed on the stack, so that a call to CHRGET will return to the address of the code that executes the statement (see the table of statement tokens at 40972 ($A00C)).

An invalid token will cause a SYNTAX ERROR. A character whose ASCII value is less than 128 will cause LET to be executed.

43037 $A81D RESTOR

Perform RESTORE

The RESTORE command simply resets the DATA pointer at 65-66 ($41-$42) from the start of BASIC pointer at 43-44 ($2B-$2C).

43052 $A82C

Test STOP Key for Break in Program

The Kernal STOP routine is called from here, and if the key is pressed, the STOP (63213, $F6ED) command, below, is executed.

43055 $A82F STOP

Perform STOP

This entry point preserves the Carry flag (which is set to 1) and enters the END routine, which also performs STOP.

43057 $A831 END

Perform END

The current line number and text pointers are preserved for a possible CONT command, and the READY prompt is printed. If a STOP key break occurred, the BREAK message is printed first.

43095 $A857 CONT

Perform CONT

The CONT statement is performed by moving the saved pointers back to the current statement and current text character pointers. If the saved pointers cannot be retrieved, the CAN'T CONTINUE error statement is printed.

43121 $A871 RUN

Perform RUN

RUN is executed by calling the Kernal SETMSG (65048, $FE18)

routine to set the message flag for RUN mode and performing a CLR to start the program. If a line followed RUN, a GOTO is executed after the CLR.

43139 $A883 GOSUB

Perform GOSUB

This statement pushes the pointers to the current text character and current line onto the stack, along with a constant 141 ($8D) which identifies the block as saved GOSUB information to be used by RETURN. The GOTO is called.

43168 $A8A0 GOTO

Perform GOTO

This statement scans BASIC for the target line number (the scan starts with the current line if the target line number is higher, otherwise it starts with the first line). When the line is found, the pointers to the current statement and text character are changed, so that the target statement will be executed next.

43218 $A8D2 RETURN

Perform RETURN

The RETURN statement finds the saved GOSUB data on the stack, and uses it to restore the pointers to the current line and current character. This will cause execution to continue where it left off when GOSUB was executed.

43256 $A8F8 DATA

Perform DATA

DATA uses the next subroutine to find the offset to the next statement, and adds the offset to the current pointers so that the next statement will be executed. In effect, it skips the statement, much like REM.

43270 $A906 DATAN

Search Program Text for the End of the Current BASIC Statement

This routine starts at the current byte of program text and searches until it finds a zero character (line delimiter) or a colon character that is not in quotes (statement delimiter).

43304 $A928 IF

Perform IF

IF uses the FRMEVL routine at 44446 ($AD9E) to reduce the expression which follows to a single term. If the expression evaluates to 0 (false), the routine falls through to REM. If it is not 0, GOTO or the statement following THEN is executed.

43323 $A93B REM

Perform REM

The REM statement is executed by skipping all program text until the beginning of the next statement. It is actually a part of the IF statement, which continues for a few bytes after the REM part.

43339 $A94B ONGOTO

Perform ON GOTO or ON GOSUB

ON is performed by converting the argument to an integer, and then skipping a number between commas each time that the integer is decremented until the argument reaches 0. If a GOTO or GOSUB is the next token, the current number between commas is used to execute one of those statements. If the numbers between commas are used up before the argument reaches 0, the statement has no effect, and the next statement is executed.

43371 $A96B LINGET

Convert an ASCII Decimal Number to a Two-Byte Binary Line Number

This subroutine is used by several statements to read a decimal number, convert it to a two-byte integer line number (in low-byte, high-byte format), and check that it is in the correct range of 0-63999.

43429 $A9A5 LET

Perform LET

The LET command causes variables to be created and initialized, or to have a new value assigned. It handles all types of array or nonarray variables: strings, floating point, integer, ST, TI, and TI$. The routine is composed of several subroutines that evaluate the variable, evaluate the assigned expression, check that the assigned value is suitable for a variable of that type, and then assign a value to the existing variable, or create a new variable.

43648 $AA80 PRINTN

Perform PRINT#

The PRINT# statement calls CMD and then closes the output channel with the Kernal CLRCHN routine (62259, $F333).

43654 $AA86 CMD

Perform CMD

This routine calls the Kernal CHKOUT routine (62032, $F250), and calls PRINT to send any included text to the device. Unlike PRINT# it leaves the output channel open, so that output continues to go to that device.

43680 $AAA0 PRINT

Perform PRINT

The PRINT routine has many segments, which are required for the various options which can be added to it: TAB, SPC, comma, semicolon, variables, PI, ST, TI, and TI$. Eventually, all output is converted to strings, and the Kernal CHROUT routine is called to print each character.

43806 $AB1E STROUT

Print Message from a String Whose Address Is in the .Y and .A Registers

This part of the PRINT routine outputs a string whose address is in the Accumulator (low byte) and .Y register (high byte), and which ends in a zero byte.

43853 $AB4D DOAGIN

Error Message Formatting Routines for GET, INPUT, and READ

43899 $AB7B GET

Perform GET and GET#

The GET routine first makes sure that the program is not in direct mode. It opens an input channel using the Kernal CHKIN routine (61966, $F20E) if a number sign was added to make GET#. Then it calls the common I/O routines in READ to get a single character, and causes the input channel to be closed if one was opened.

43941 $ABA5 INPUTN

Perform INPUT#

This routine opens an input channel with the Kernal CHKIN routine, calls INPUT, and then closes the channel with a CHKOUT routine (62032, $F250). Extra data is discarded without an EXTRA IGNORED message, and a FILE DATA ERROR message is issued when the data type is not suitable for the type of variable used.

43967 $ABBF INPUT

Perform INPUT

The INPUT routine checks to make sure that direct mode is not active, prints prompts, receives a line of input from the device, and jumps to the common code in READ that assigns the input to the variables which were named.

44038 $AC06 READ

Perform READ

This routine includes the READ command and common code for

GET and INPUT. The READ command locates the next piece of DATA, reads the text, and converts it to the appropriate type of data to be assigned to a numeric or string variable.

44284 $ACFC EXIGNT

ASCII Text for Input Error Messages

The text stored here is ?EXTRA IGNORED and ?REDO FROM START, each followed by a carriage return and a zero byte.

44318 $AD1E NEXT

Perform NEXT

NEXT is executed by finding the appropriate FOR data on the stack, adding the STEP value to the FOR variable, and comparing the result to the TO value. If the loop is done, the stack entries for that FOR command are removed from the stack. If the loop hasn't reached its limit, the pointers to the current statement and text character are updated from the FOR stack entry, which causes execution to continue with the statement after the FOR statement.

44426 $AD8A FRMNUM

Evaluate a Numeric Expression and/or Check for Data Type Mismatch

This routine can be called from different entry points to check the current data against the desired data type (string or numeric) to see if they match. If they don't, a TYPE MISMATCH error will result.

44446 $AD9E FRMEVL

Evaluate Expression

This is the beginning point of a very powerful group of subroutines which are used extensively by BASIC.

The main purpose of these routines is to read in the ASCII text of BASIC expressions, separate the operators and terms of the expression, check them for errors, combine the individual terms by performing the indicated operations, and obtain a single value which the BASIC program can use.

This can be a very complex task, as expressions can be of the string or numeric type, and can contain any type of variable, as well as constants.

At the end, the flag which shows whether the resulting value is string or numeric at 13 ($D) is set, and if the value is numeric, the flag at 14 ($E) is set as well, to show if it is an integer or floating point number.

44675 $AE83 EVAL

Convert a Single Numeric Term from ASCII Text to a Floating Point Number

This routine reduces a single arithmetic term which is part of an expression from ASCII text to its floating point equivalent.

If the term is a constant, the routine sets the data type flag to number, sets the text pointer to the first ASCII numeric character, and jumps to the routine which converts the ASCII string to a floating point number.

If the term is a variable, the variable value is retrieved. If it is the PI character, the value of PI is moved into the Floating Point Accumulator.

This routine is vectored through RAM at 778 ($30A).

44712 $AEA8 PIVAL

PI Expressed as a Five-Byte Floating Point Number

The value of PI is stored here as a five-byte floating point number.

44785 $AEF1 PARCHK

Evaluate Expression Within Parentheses

This routine evaluates an expression within parentheses by calling the syntax checking routines that look for opening and closing parentheses, and then calling FRMEVL 44446 ($AD9E) for each level of parentheses.

44791 $AEF7 CHKCLS

Check for and Skip Closing Parentheses

44794 $AEFA CHKOPN

Check for and Skip Opening Parentheses

44799 $AEFF CHKCOM

Check for and Skip Comma

This syntax checking device is the same in substance as the two checking routines above. It is used when the next character should be a comma. If it is not, a SYNTAX ERROR results: If it is, the character is skipped and the next character is read. Any character can be checked for and skipped this way by loading the character into the Accumulator and entering this routine from SYNCHR at 44799 ($AEFF).

44808 $AF08 SNERR

Print Syntax Error Message

44843 $AF2B ISVAR

Get the Value of a Variable

44967 $AFA7 ISFUN

Dispatch and Evaluate a Function

If a BASIC function (like ASC("A")) is part of an expression, this routine will use the function dispatch table at 42242 ($A502) to set up the address of the proper function routine, and then branch to that routine.

45030 $AFE6 OROP

Perform OR

The OR routine sets the .Y register as a flag, and falls through to the AND routine, which also performs OR.

45033 $AFE9 ANDOP

Perform AND

The AND routine changes the parameters to two-byte integer values, and performs the appropriate logical operation (AND or OR). A result of 0 signifies false, while a result of –1 signifies true.

45078 $B016 DORE1

Perform Comparisons

This routine does the greater than (>), less than (<), and equal (=) comparisons for floating point numbers and strings. The result in the Floating Point Accumulator will be 0 if the comparison is false, and –1 if it is true.

45185 $B081 DIM

Perform DIM

This command calls the next routine to create an array for every variable dimensioned (since a statement can take the form DIM A(12), B(13), C(14)...). If an array element is referenced before a DIM statement (for example, A(3)=4), the array will be dimensioned to 10 (as if DIM A(10) were executed). Remember, DIMensioning an array to 10 really creates 11 elements (0-10). The 0 element should always be considered in calculating the size to DIMension your array.

45195 $B08B PTRGET

Search for a Variable and Set It Up If It Is Not Found

This routine attempts to locate a variable by searching for its name in the variable area. If an existing variable of that name cannot be found, one is created with the NOTFNS routine below.

45331 $B113

Check If .A Register Holds Alphabetic ASCII Character

This is part of the check for a valid variable name (it must start with an alphabetic character).

45341 $B11D NOTFNS

Create a New BASIC Variable

This routine makes space for a seven-byte descriptor by moving the variable storage area seven bytes higher in memory, and then creates the descriptor.

45445 $B185 FINPTR

Return the Address of the Variable That Was Found or Created

This routine stores the address of the variable that was found or created by the preceding routines in a pointer at 71-72 ($47-$48).

45460 $B194 ARYGET

Allocate Space for Array Descriptors

This routine allocates five bytes plus two bytes for every dimension specified for the array descriptor.

45477 $B1A5 N32768

The Constant -32768 in Five-Byte Floating Point Format

This constant is used for range checking in the conversion of a floating point number to a signed integer (the minimum integer value is -32768).

45482 $B1AA

Convert a Floating Point Number to a Signed Integer in .A and .Y Registers

This subroutine calls AYINT, below, which checks to make sure that the number in the Floating Point Accumulator is between 32767 and -32768, and converts it to a 16-bit signed interger in 100-101 ($64-$65), high byte first. It leaves the high byte of the integer in the Accumulator, and the low byte in the .Y register.

Although this routine does not appear to be referenced anywhere in BASIC, the vector at locations 3-4 points to its address. Presumably, it is provided for the benefit of the user who wishes to pass parameters in a USR call, or the like.

45490 $B1B2 INTIDX

Input and Convert a Floating Point Subscript to a Positive Integer

This routine converts a floating point subscript value to an integer, making sure first that it is positive.

45503 $B1BF AYINT

Convert a Floating Point Number to a Signed Integer

This subroutine first checks to make sure that the number in the Floating Point Accumulator is between 32767 and -32768. If it is not, an ILLEGAL QUANTITY error results. If it is, the routine converts it to a 16-bit signed integer with the high byte in location 100 ($64), and the low byte in location 101 ($65).

45521 $B1D1 ISARY

Find Array Element or Create New Array in RAM

This routine searches for an array. If it is found, the subscript value is checked to see if it is valid, and pointers to the array and element of the array are set. If it is not found, the array is created, and the pointers set.

45637 $B245 BSERR

Print BAD SUBSCRIPT Error Message

45640 $B248 FCERR

Print ILLEGAL QUANTITY Error Message

45900 $B34C UMULT

Compute the Size of a Multidimensional Array

This routine calculates the size of a multidimensional array by multiplying the dimensions.

45949 $B37D FRE

Perform FRE

The FRE function calls the garbage collection routine at 46374 ($B526) to get rid of unused string text, and calculates the difference between the bottom of string text and the top of array storage. It then drops through to the following routine, which assumes that the free memory value is a signed 16-bit integer, and converts it to floating point accordingly.

Of course, while the free memory space on the PET might have always been 32767 or less (the maximum value of a signed integer), such is definitely not the case on the 64. Because the conversion is from a signed integer, any memory value over 32767 will be regarded as negative (the high bit is treated as a sign bit). Therefore, for these higher values you must add twice the bit value of the high bit (65536) in order to come up with the correct value. The expression FRE (0)–65536*(FRE(0)<0) will always return the correct amount of free memory.

45969 $B391 GIVAYF

Convert 16-Bit Signed Integer to Floating Point

This routine treats the value in the Accumulator as the high byte of a 16-bit signed integer, and the value in the .Y register as the low byte, and converts the signed integer into a floating point number in the Floating Point Accumulator.

The address of this routine is pointed to by the RAM vector at 5-6, and the routine can be used to return an argument from the USR call in the Floating Point Accumulator.

45982 $B39E POS

Perform POS

The POS command calls the Kernal PLOT routine (58634, $E50A) to get the position of the cursor on the logical line. What it really does is an equivalent of PEEK (211). Remember, since we are dealing with a logical line, this number can be over 39. The statement "THIS SENTENCE IS LONGER THAN ONE PHYSICAL LINE";POS(X) will return a value of 48 for the POS(X).

45990 $B3A6 ERRDIR

Check If the Program Is Running in Direct Mode, and If So Issue an Error

This routine is called by statements that prohibit execution in direct mode. It checks a flag that is set when a line without a line number is entered, and causes an ILLEGAL DIRECT error if the flag is set.

46003 $B3B3 DEF

Perform DEF

DEF performs some syntax checking, and pushes five bytes onto the stack: the first byte of the function statement, a two-byte pointer to the dependent variable (the X in FN(X)), and the address of the first character of the definition itself, where it resides in program text.

The DEF statement must fit on one line, but functions can be extended by nesting them (having one function call another).

46049 $B3E1 GETFNM

Check DEF and FN Syntax

This routine checks to make sure that FN follows DEF, and that the dependent variable has a valid floating point variable name. It calls the routine to find or create a variable to get the pointer to its address.

46068 $B3F4 FNDOER

Perform FN

The FN evaluation is done by evaluating the FN argument (for example, FN(A+B*C/D)) and then getting the rest of the expression from the text of the function definition statement. The function variable descriptor area is used as a work area, and the dependent variable is not disturbed (so that if the definition used FN(X), the value of X will not be changed by the function call).

46181 $B465 STRD

Perform STR$

STR$ first checks to make sure that the parameter is a number, and then calls the routines that convert floating point to ASCII and create the pointers to a string constant.

46215 $B487 STRLIT

Scan and Set Up Pointers to a String in Memory

This routine calculates the length of the string, and calls the routine that allocates space in memory. It then saves the string, or creates a pointer to its location in the BASIC text input buffer at 512 ($200).

46324 $B4F4 GETSPA

Allocate Space in Memory for String

The amount of space needed for a string is passed to this routine, and the routine checks if there is that amount of space available in free memory. If not, it does a garbage collection and tries again.

46374 $B526 GARBAG

String Garbage Collection

Whenever a string is changed in any way, the revised version of the text is added to the bottom of the string text area, leaving the old version higher up in memory, wasting space.

In order to reclaim that space, the descriptor for every string whose text is in the string text area (rather than in the program text area) must be searched to find the valid text that is highest in memory. If that string is not as high as it could be, it is moved up to replace any string that is no longer valid. Then all of the string descriptors must be searched again to find the next highest string and move it up. This continues until every string that is in use has been covered. After all have been moved up, the pointer to the bottom of string text at 51-52 ($33-$34) is changed to show the new bottom location.

If there are more than a few strings whose text is in the string text storage area, rather than in the body of the program, scanning every string as many times as there are strings can take an awful lot

of time. The computer may seem as if it had died (the STOP key is not even checked during the procedure).

The collection will take about as long whether there is any spare space or not; the full collection will be done even if it is done immediately after the last collection. Although the increased memory capacity of the 64 helps to forestall the need for garbage collection, a large program with many string arrays may still experience lengthy collection delays.

46525 $B5BD

Check for Most Eligible String to Collect

This part of the garbage collection routine checks to see if the current string is the highest in memory.

46598 $B606

Collect a String

This part of the garbage collection routine moves the string to high memory and updates the descriptor to point to its new location.

46653 $B63D CAT

Concatenate Two Strings

This routine is used to add the text of one string onto the end of another (A$+B$). Error checking is done to see if the length of the combined string is within range, the allocation routine is called to allocate space, and the new string is built at the bottom of the string text area.

46714 $B67A MOVINS

Move a String In Memory

This is the routine which is used to move a string to the bottom of the string text area for the above routine. It is generally used as a utility routine to move strings.

46755 $B6A3 FRESTR

Discard a Temporary String

This routine calls the following routine which clears an entry from the temporary descriptor stack. If the descriptor was on the stack, it exits after setting pointers to the string and its length. If it wasn't on the temporary stack and is at the bottom of string text storage, the pointer to the bottom is moved up to deallocate the string.

46811 $B6DB FRETMS

Remove an Entry from the String Descriptor Stack

If the descriptor of a currently valid string is the same as one of the entries on the temporary string descriptor stack, the stack entry is removed.

46828 $B6EC CHRD

Perform CHR$

The CHR$ routine creates a descriptor on the temporary string stack for the one-byte string whose value is specified in the command, and sets a pointer to that string.

46848 $B700 LEFTD

Perform LEFT$

LEFT$ creates a temporary string descriptor for a new string which contains the number of characters from the left side of the string that is specified in the command.

46892 $B72C RIGHTD

Perform RIGHT$

RIGHT$ manipulates its parameters so that the tail end of LEFT$ can be used to create a temporary string descriptor for a new string. This new string contains the number of characters from the right side of the string that is specified in the command.

46903 $B737 MIDD

Perform MID$

MID$ manipulates its parameters so that the tail end of LEFT$ can be used to create a temporary string descriptor for a new string. This new string contains the number of characters from the position in the middle of the string that is specified in the command.

46945 $B761 PREAM

Pull String Function Parameters from Stack for LEFT$, RIGHT$, and MID$

This routine is used to obtain the first two parameters for all three of these commands.

46972 $B77C LEN

Perform LEN

The LEN function is performed by obtaining the string length from the descriptor and converting it to a floating point number.

46987 $B78B ASC

Perform ASC

This routine gets the first character of the string in the .Y register (if it's not a null string). Then it calls the part of POS that converts a one-byte integer in .Y to a floating point number.

47003 $B79B GETBYTC

Input a Parameter Whose Value Is Between 0 and 255

This routine reads numeric ASCII program text, converts it to an integer, checks that it is in the range 0-255, and stores it in the .X register. This routine can be useful for reading parameters from a USR statement or new commands.

47021 $B7AD VAL

Perform VAL

The VAL routine obtains the string pointer, and reads the string one character at a time until an invalid character is found (ASCII numbers, sign character, a single decimal point, exponent, and spaces are all valid). Then the string is changed to floating point. If no valid characters are found, a 0 is returned.

47083 $B7EB GETNUM

Get a 16-Bit Address Parameter and an 8-Bit Parameter (for POKE and WAIT)

This routine gets the next numeric parameter from the current place in program text. The routine evaluates it, checks that it is a positive integer within the range 0-65535, and changes it from floating point to a two-byte integer in 20-21 ($14-$15). It checks for and skips a comma, then gets a one-byte integer parameter in the .X register. The routine is used to get the parameters for POKE and WAIT.

47095 $B7F7 GETADR

Convert a Floating Point Number to an Unsigned Two-Byte Integer

This routine checks the number in the Floating Point Accumulator to make sure that it is a positive number less than 65536, and then calls the subroutine which converts floating point to integer. It is used to get address parameters, for commands such as PEEK.

47117 $B80D PEEK

Perform PEEK

PEEK reads the address parameter and converts it to a pointer. Then it gets the byte pointed to into the .Y register, and calls the part of POS that converts a single integer in .Y to a floating point number.

47140 $B824 POKE

Perform POKE

POKE gets a pointer to the address parameter, and stores the next parameter there.

47149 $B82D FUWAIT

Perform WAIT

WAIT gets an address parameter and an integer parameter to use as a mask. WAIT then looks for an optional parameter to use as a pattern for the exclusive OR. Then, the address location is read, its value is exclusive ORed with the optional pattern value (or 0 if there is none). This value is ANDed with the mask value. The command loops continuously until the result is not-zero.

The purpose of this command is to allow the program to watch a location which can be changed by the system or by outside hardware (such as the software clock or keycode value locations).

The AND function lets you check if a bit changes from 0 to 1, while the EOR function allows you to check if a bit changes from 1 to 0. For more information see the article "All About the Wait Instruction," by Louis Sander and Doug Ferguson, in *COMPUTE!'s First Book of Commodore 64.*

47177 $B849 FADDH

Add .5 to Contents of Floating Point Accumulator #1

47184 $B850 FSUB

Subtract FAC1 from a Number in Memory

This routine is used to subtract the Floating Point Accumulator from a number in memory. It moves the number in memory into FAC2, and falls through to the next routine.

47187 $B853 FSUBT

BASIC's Subtraction Operation

This routine subtracts the contents of FAC2 from FAC1 by complementing its sign and adding.

47207 $B867 FADD

Add FAC1 to a Number in Memory

This routine is used to add the contents of the Floating Point Accumulator (FAC1) to a number in memory, by moving that number into FAC2, and falling through to the next routine.

47210 $B86A FADDT

Perform BASIC's Addition Operation

This routine adds the contents of FAC1 and FAC2 and stores the results in FAC1.

47271 $B8A7 FADD4

Make the Result Negative If a Borrow Was Done

47358 $B8FE NORMAL
Normalize Floating Point Accumulator #1

47431 $B947 NEGFAC
Replace FAC1 with Its 2's Complement

47486 $B97E OVERR
Print Overflow Error Message

47491 $B983 MULSHF
SHIFT Routine

47548 $B9BC FONE
Floating Point Constant with a Value of 1
The five-byte floating point representation of the number 1 is stored here for use by the floating point routines. It is also used as the default STEP value for the FOR statement.

47553 $B9C1 LOGCN2
Table of Floating Point Constants for the LOG function
This table of eight numeric constants in five-byte floating point representation is used by the LOG function.

47594 $B9EA LOG
Perform LOG to Base E
The LOG to the base e of the number in FAC1 is performed here, and the result left in FAC1.

47656 $BA28 FMULT
Multiply FAC1 by Value in Memory
This routine is used to multiply the Floating Point Accumulator (FAC1) by a number in memory. It moves the number in memory into FAC2, and falls through to the next routine.

47667 $BA33 FMULT
Multiply FAC1 with FAC2
This routine multiplies the contents of FAC1 by the contents of FAC2 and stores the result in FAC1.

47705 $BA59 MLTPLY
Multiply a Byte Subroutine
This subroutine is used to repetitively add a mantissa byte of FAC2 to FAC1 the number of times specified in the .A register.

47756 $BA8C CONUPK

Move a Floating Point Number from Memory into FAC2

This subroutine loads FAC2 from the four-byte number (three mantissa and one sign) pointed to by the .A and .Y registers.

47799 $BAB7 MULDIV

Add Exponent of FAC1 to Exponent of FAC2

47828 $BAD4 MLDVEX

Handle Underflow or Overflow

47842 $BAE2 MUL10

Multiply FAC1 by 10

This subroutine is called to help convert a floating point number to a series of ASCII numerals.

47865 $BAF9 TENC

The Constant 10 in Five-Byte Floating Format

47870 $BAFE DIV10

Divide FAC1 by 10

47887 $BB0F FDIV

Divide a Number in Memory by FAC1

This number in memory is stored to FAC2, and this routine falls through to the next.

47890 $BB12 FDIVT

Divide FAC2 by FAC1

This routine is used to divide the contents of FAC2 by the contents of FAC1, with the result being stored in FAC1. A check for division by 0 is made before dividing.

48034 $BBA2 MOVFM

Move a Floating Point Number from Memory to FAC1

This routine loads FAC1 with the five-byte floating point number pointed to by the address stored in the Accumulator (low byte) and the .Y register (high byte).

48071 $BBC7 MOV2F

Move a Floating Point Number from FAC1 to Memory

This routine is used to move a number from the Floating Point Accumulator (FAC1) to memory at either 92-96 ($5C-$60) or 87-91 ($57-$5B), depending on the entry point to the routine.

48124 $BBFC MOVFA

Move a Floating Point Number from FAC2 to FAC1

48140 $BC0C MOVAF

Round and Move a Floating Point Number from FAC1 to FAC2

48143 $BC0F MOVEF

Copy FAC1 to FAC2 Without Rounding

48155 $BCIB ROUND

Round Accumulator #1 by Adjusting the Rounding Byte

If doubling the rounding byte at location 112 ($70) makes it greater than 128, the value of FAC1 is increased by 1.

48171 $BC2B SIGN

Put the Sign of Accumulator #1 into .A Register

On exit from this routine the Accumulator will hold a 0 if FAC1 is 0, a 1 if it is positive, and a value of 255 ($FF) if it is negative.

48185 $BC39 SGN

Perform SGN

The SGN routine calls the above routine to put the sign of FAC1 into .A, and then converts that value to a floating point number in FAC1.

48216 $BC58 ABS

Perform ABS

The FAC1 sign byte at 102 ($66) is shifted right by this command, so that the top bit is a 0 (positive).

48219 $BC5B FCOMP

Compare FAC1 to Memory

On entry to this routine, .A and .Y point to a five-byte floating point number to be compared to FAC1. After the comparison, .A holds a 0 if the two are equal, a 1 if the value of FAC1 is greater than that in the memory location, and 255 ($FF) if the value in FAC1 is less than that in the memory location.

48283 $BC9B QINT

Convert FAC1 into Integer Within FAC1

This routine converts the value in FAC1 into a four-byte signed integer in 98-101 ($62-$65), with the most significant byte first.

48332 $BCCC INT

Perform INT

This routine removes the fractional part of a floating point number by calling the routine above to change it to an integer, and then changing the integer back to floating point format.

48371 $BCF3 FIN

Convert an ASCII String to a Floating Point Number FAC1

This routine is called by VAL to evaluate and convert an ASCII string to a floating point number.

48510 $BD7E FINLOG

Add Signed Integer to FAC1

This routine is used to add an ASCII digit that has been converted to a signed integer to FAC1.

48563 $BDB3 N0999

This table of three floating point constants holds the values 99,999,999.9, 999,999,999.5 and 1,000,000,000. These values are used in converting strings to floating point numbers.

48576 $BDC0 INPRT

Print IN Followed by a Line Number

48589 $BDCD LINPRT

Output a Number in ASCII Decimal Digits

This routine is used to output the line number for the routine above. It converts the number whose high byte is in .A and whose low byte is in .X to a floating point number. It also calls the routine below, which converts the floating point number to an ASCII string.

48605 $BDDD FOUT

Convert Contents of FAC1 to ASCII String

This routine converts a floating point number to a string of ASCII digits, and sets a pointer to the string in .A and .Y.

48913 $BF11 FHALF

The Constant Value 1/2 in Five-Byte Floating Point Notation

This constant is used for rounding and SQR.

48924 $BF1C FOUTBL

Powers of Minus Ten Constants Table

This table contains the powers of -10 expressed as four-byte floating point numbers (that is, -1; +10; -100; +1000; -10,000; +100,000; -1,000,000; +10,000,000; and -100,000,000).

48954 $BF3A FDCEND

Table of Constants for TI$ Conversion

This table contains the floating point representation of powers of -60 multiplied by 1 or 10. These constants are used for converting TI$ to ASCII.

48978 $BF52

Unused area

This unused area is filled with bytes of 170 ($AA).

49009 $BF71 SQR

Perform SQR

This routine moves the contents of FAC1 to FAC2, moves the constant 0.5 to FAC1, and falls through to the exponentiation routine.

49019 $BF7B FPWRT

Performs Exponentiation (Power Calculation Called for by UPARROW)

This routine raises the value in FAC2 to the power in FAC1 and leaves the result in FAC1.

49076 $BFB4 NEGOP

Perform NOT and >

This negates the Floating Point Accumulator by exclusive ORing the sign byte with a constant of 255 ($FF). Zero is left unchanged. The results of this command follow from the formula NOT X=-(X+1). Therefore, if you NOT a statement that is true (-1), you get 0 (false).

49087 $BFBF EXPCON

Table of Constants for EXP and LOG in Five-Byte Floating Point Format

These tables are used to calculate 2 to the N power.

49133 $BFED EXP

Perform EXP

This routine calculates the natural logarithm e (2.718281828...) raised to the power in FAC1. The result is left in FAC1.

This routine is split between the BASIC ROM which ends at 49151 ($BFFF) and the Kernal ROM which begins at 57344 ($E000). Therefore, a JMP $E000 instruction is tacked on to the end, which makes the BASIC routines in the 64 Kernal ROM three bytes higher in memory than the corresponding VIC-20 routines.

4K Free RAM

49152-53247 ($C000-$CFFF)

Locations 49152 to 53247 ($C000 to $CFFF) are free RAM. Since this area is not contiguous with the BASIC program text RAM area, it is not available for BASIC program or variable storage (it is not counted in the FRE(0) total).

This area is fully available for any other use, however, such as storing machine language subroutines for use with BASIC, alternate I/O drivers for parallel or IEEE devices, character graphics or sprite data, etc.

This large free area is such a tempting spot for system additions that many such applications may be competing for the same RAM space. For example, the Universal Wedge DOS Support program that adds easy access to the disk communications channel is usually loaded at 52224 ($CC00). Programs that use that part of RAM will therefore overwrite the DOS support program, with the result that they may not run correctly, or even at all. Likewise, Simon's BASIC, the extended language which Commodore has released on cartridge, uses several locations in this range. Be aware of this potential problem when you buy hardware additions that use this spot to hook into the system.

Chapter 6

VIC-II, SID, I/O Devices, Color RAM, and Character ROM

VIC-II, SID, I/O Devices, Color RAM, and Character ROM

53248-57343 ($D000-$DFFF)

This 4K block of memory is used for several key functions. Normally, the 6510 microprocessor addresses the two Complex Interface Adapter (CIA) Input/Output chips here, along with the VIC-II video controller chip, the Sound Interface Device (SID) music synthesizer, and the Color RAM.

Alternatively, the 6510 can address the character ROM here (though normally only the VIC-II chip has access to it). Finally, there is also 4K of RAM here, although to use it may require banking it in only when necessary, as the I/O devices are needed for such niceties as reading the keyboard, and updating the screen display.

It will appear from the map of the I/O devices below that many of the locations are not accounted for. That is because these devices tie up more addressing space than they actually use. Each of them uses only a few addresses, mostly on the bit level.

The missing addresses either consist of images of the hardware registers, or cannot be addressed in this configuration. In addition, some address space is left open for the use of future hardware devices which might be plugged into the expansion port, like the CP/M card.

As mentioned above, memory usage by these I/O devices is so intensive that to work with them often requires that you turn individual bits on and off. Here is a quick reminder of how to manipulate bits.

The bit values for each bit are:

Bit 0=1, Bit 1=2, Bit 2=4, Bit 3=8, Bit 4=16, Bit 5=32, Bit 6=64, Bit 7=128.

To set a bit to 1 from BASIC, POKE address, PEEK (address) OR Bitvalue. To reset a bit to 0 from BASIC, POKE address, PEEK (address) AND 255–Bitvalue).

53248-53294 ($D000-$D02E)

VIC-II Chip Registers

The Video Interface Controller (VIC-II chip) is a specially designed processor that is in charge of the 64's video display. It is this chip which makes possible the 64's wide range of graphics capabilities.

The VIC-II chip's ability to address memory is independent of the 6510 microprocessor. It can address only 16K at a time, and any of the four blocks of 16K can be chosen for video memory. The system default is for it to use the first 16K.

All of the video display memory, character dot data, and sprite shapes must be stored within the chosen 16K block. Locations 53248-53294 ($D000-$D02E) are registers which allow the user to communicate with the VIC-II chip. Although for the most part they can be written to and read like ordinary memory locations, their contents directly control the video display. Since many of these locations work in close conjunction with others, a general overview of some of the different graphics systems on the 64 is in order.

The most familiar type of graphics display is the ordinary text that appears when you turn the machine on. The area of RAM which is displayed on the screen is determined by the Video Matrix Base Address Nybble of the VIC-II Memory Control Register (53272, $D018). The address of the dot-data which is used to assign a shape to each text character based on an 8 by 8 matrix of lit or unlit dots is determined by the other half of the Memory Control Register at 53272 ($D018). More information on how the data is used to represent the character shapes may be found at the alternate entry for 53248 ($D000), the Character Generator ROM.

Text character graphics may employ one of the two sets of text and graphics characters in the Character Generator ROM, or the user may substitute a completely different set of graphics or text characters in RAM.

Normally, the text graphics screen uses a background color which is common to all text characters, and that value is stored in Background Color Register 0 (53281, $D021). The color of the frame around the screen is determined by the Border Color Register at 53280 ($D020).

The color of each character is determined by one nybble of the Color RAM which starts at 55296 ($D800). There are, however, two variations which alter this scheme somewhat.

The first is called multicolor text mode, and is set by Bit 4 of 53270 ($D016). Instead of each bit selecting either the foreground or the background color for each dot in the character, bit-pairs are used to select one of four colors for each double-width dot. This results in the horizontal resolution being cut to four dots across per character, but allows two extra colors to be introduced from Background Color Registers 1 and 2 (53282-53283, $D022-$D023).

The other text mode is called Extended Background Color Mode. In this mode, the foreground color is always selected by the Color RAM. The background color depends on the actual screen code of the character. In this mode, only the first 64 character shapes are available, but each can have one of four different background colors.

The background color for each character is determined by its screen code as follows:

1. If the screen code is from 0–63 (this includes the normal alphanumerics), the value in Background Color Register 0 (53281, #D021) will determine the background color, as is usual.
2. Characters with codes 63-255 will have the same shape as the corresponding character in the group with codes 0-63.
3. For characters with codes 63-127 (SHIFTed characters), the background colors are determined by the value in Background Color Register 1 (53282, $D022).
4. The value in Background Color Register 2 (53283, $D023) is used for characters with codes 128-191 (reversed alphanumerics).
5. For characters with codes 192-255, the value in Background Color Register 3 (53284, $D024) is used to determine the background color.

In place of the normal text mode, a bitmap graphics mode is also available by setting Bit 5 of location 53265 ($D011). In this mode, each bit of data determines whether one dot on the screen will be set to either the background color or foreground color. Within an 8 by 8 dot area, the foreground and background colors may be individually selected.

The bitmap area is 320 dots wide and 200 dots high. The area which contains the graphics data, the *bitmap* is determined by the Character Dot Data Base Address in the lower nybble of the VIC-II Memory Control Register (53272, $D018). The Video Matrix Base Address in the upper nybble, which normally determines which area of memory will be displayed, instead determines where the color memory for each 8 by 8 group of dots will be located.

The Color RAM is not used for high-resolution bitmap graphics. But multicolor mode is also available for bitmap graphics, and it uses the Color RAM to determine the foreground color of each dot.

As with multicolor text mode, the horizontal resolution is cut in half (to 160 dots across), so that in addition to the foreground and background colors, each dot can be one of two other colors as well. This mode gets the value for the two extra colors from the two nybbles of each byte of bitmap color memory, the location of which is determined by the Video Matrix Base Address.

Multicolor text mode offers four colors, three of which will be common to all characters, and one of which can be selected individually. Multicolor bitmap mode offers a choice of four colors, three of

which can be individually selected within an 8 by 8 dot area.

The 64 also contains an entirely separate graphics system, whose character shapes, colors, and positions are derived and displayed without any reference to the Video Matrix and Character Dot-Data addresses. Best of all, these characters may be moved quickly and easily to any position on the screen, greatly facilitating games and animated graphics of all types. This system is known as *sprite graphics.*

Sprite graphics takes its name from the graphics characters it displays, each of which is called a sprite. There are eight sprites, known as Sprites 0-7. Each sprite character is 24 dots wide by 21 dots high. This is about eight times as large as a regular text character, which is only 8 dots wide by 8 dots high.

A sprite takes its shape from 63 bytes of data in one of the 256 data blocks, each 64 bytes long, that can fit into the 16K space which the VIC-II chip can address. The block currently assigned to any given sprite is determined by the Sprite Data Pointers, which are located at the last eight bytes of the screen memory area (the default locations are 2040-2047, $7F8-$7FF).

The first Sprite Data Pointer determines the data block used for the shape of Sprite 0, the second for the shape of Sprite 1, etc. The number in the pointer times 64 equals the address of the first byte of the data block within the VIC-II addressing range.

For example, using the default values for VIC-II addressing area and screen memory, a value of 11 in location 2040 ($7F8) would mean that the shape of Sprite 0 is determined by the data in the 63-byte block starting at location 704 (11*64). It should be noted that it is possible for more than one sprite to take its shape data from the same block, so that only 64 bytes of data are required to create eight sprites, each having the same shape.

The dot patterns of each sprite correspond to the bit patterns of the sprite shape data. Each byte of shape data in memory consists of a number from 0 to 255. This number can be represented by eight binary digits of 0 or 1.

Each binary digit has a bit value that is two times greater than the last. If the digit in the zero bit place is a 1, it has a value of 1 (we count bit places from 0 to 7). A 1 in the first bit place has a value of 2, the second bit has a value of 4, the third has a value of 8, the fourth has a value of 16, the fifth a value of 32, the sixth a value of 64, and the seventh a value of 128.

By making all of the possible combinations of 0's and 1's in all eight bit places, and adding the bit values of every bit place that contains a 1, we can represent every number from 0 to 255 as a series of 1's and 0's.

If you think of every 0 as a dot having the same color as the background, and every 1 as a dot which is the color of the sprite,

you can see how a series of bytes could be used to represent the sprite shape.

Since each line of the sprite is 24 dots wide, it takes 3 bytes of memory (24 bits) per line to portray its shape. Let's take a look at a couple of sample sprite lines.

00000000 01111110 00000000 =0,126,0

As you can see, the first and last bytes are all 0's, so nothing will be displayed there. The middle byte has six 1's, so it will be displayed as a line six dots long. By adding the values of these six bits (64+32+16+8+4+2), we get a byte value of 126. Let's try another line.

00011111 11111111 11111000 =31,255,248

The first byte has five bits set to 1, having values of 16, 8, 4, 2, and 1, for a total value of 31. The second byte has all bits set to 1, so it has the maximum value of 255. The third byte also has five bits set to 1, having values of 128, 64, 32, 16, and 8, for a total of 248. The result is that this line of sprite data will display a line that is 18 dots long.

We can put these two kinds of lines together to show how a large cross might be drawn using bytes of sprite data.

```
000000000000000000000000 =0,0,0
000000000000000000000000 =0,0,0
000000000111111000000000 =0,126,0
000000000111111000000000 =0,126,0
000000000111111000000000 =0,126,0
000000000111111000000000 =0,126,0
000000000111111000000000 =0,126,0
000000000111111000000000 =0,126,0
000000000111111000000000 =0,126,0
000111111111111111111000 =31,255,248
000111111111111111111000 =31,255,248
000111111111111111111000 =31,255,248
000000000111111000000000 =0,126,0
000000000111111000000000 =0,126,0
000000000111111000000000 =0,126,0
000000000111111000000000 =0,126,0
000000000111111000000000 =0,126,0
000000000111111000000000 =0,126,0
000000000111111000000000 =0,126,0
000000000000000000000000 =0,0,0
000000000000000000000000 =0,0,0
```

The 63 numbers, displayed three per line opposite the bit patterns they represent, are the values that would have to be put in the sprite shape data area in order to display this cross using sprite graphics.

Even after the sprite shape data has been placed in memory, and the Sprite Data Pointer has been set to display that block of data bytes as the sprite shape, there are still several steps that must be taken in order to display the sprite on the screen.

The proper bit of the Sprite Display Enable Register at 53269 ($D015) must be set to 1 in order to turn on the sprite display. A horizontal and vertical screen position must be selected for the sprite by means of the horizontal and vertical position registers (53248–53264, $D000–$D010). Finally, the color value of the sprite should be placed in the appropriate Sprite Color Register (53287–53294, $D027–$D02E).

Once you have the sprite on the screen, animation is fairly simple to achieve. Moving the sprite is as easy as changing the values in the sprite position registers. Changing the sprite shape can be accomplished by merely changing the Sprite Data Pointer to point to another block of shape data in memory.

There are also some optional sprite graphics features available which enhance its flexibility. Sprite expand registers allow you to make each sprite twice as wide as normal, twice as tall, or both. Collision detection registers let you know when a sprite shape overlaps a regular text character or bitmap dot, or if two sprites are touching.

If a sprite is positioned in the same place on the screen as a text character or bitmap dot, a Priority Register allows you to choose whether the sprite or the normal graphics will be displayed. This enables three-dimensional effects by letting you choose whether the sprite goes in front of or behind other objects on the screen.

Finally, any sprite may be selected for multicolor display, using the register at location 53276 ($D01C). In this mode, as in multicolor text and bitmap modes, pairs of bits are used to determine the color of each double-width dot. The possible color values which may be selected are those of Background Color Register 0, the Sprite Color Register, or the two color values in the Sprite Multicolor Registers at 53285-53286 ($D025-$D026).

Location Range: 53248-53264 ($D000-$D010)

Sprite Horizontal and Vertical Position Registers

These locations determine the horizontal and vertical position at which each sprite will be displayed on the screen. Each sprite has its own horizontal and vertical position register. In addition, all of the sprites share a common register which is used to extend the range of horizontal positions.

Vertical positions for each sprite range from 0 to 255, and these indicate the vertical position of the top line of the sprite's 21-line length. Since there are only 200 visible scan lines in the screen dis-

play, some of these vertical positions will result in the sprite being partially or wholly offscreen.

The visible viewing area starts at line 50 and extends to line 249. Therefore, any sprite whose vertical position is 29 ($1D) or less will be completely above the visible picture. At vertical position 30 ($1E), the bottom line of the sprite display becomes visible at the top of the screen. At vertical position 50 ($32), the entire sprite becomes visible at the top of the screen. At position 230 ($E6), the bottom line of the sprite is lost from view off the bottom of the screen, and at vertical position 250 ($FA), the entire sprite disappears from view off the bottom edge of the screen.

Horizontal positioning is somewhat trickier, because the visible display area is 320 dots wide, and one eight-bit register can hold only 256 position values. Therefore, an additional register is needed to hold the ninth bit of each sprite's horizontal position.

Each sprite is assigned a single bit in the Most Significant Bit of Horizontal Position Register (MSB Register) at 53264 ($D010). If that bit is set to 1, the value 256 is added to the horizontal position. This extends the range of possible horizontal positions to 511.

In order to set a sprite's horizontal position, you must make sure that both the values in the horizontal position register and the MSB Register are correct. For example, if you wish to set the horizontal position of Sprite 5 to a value of 30, you must place a value of 30 in the Sprite 5 Horizontal Position Register (POKE 53258,30 will do it from BASIC), and you must also clear Bit 5 of the MSB Register (POKE 53264,PEEK(53264)AND(255–16)). If you forget the MSB register, and Bit 5 is set to 1, you will end up with position 286 instead of 30.

The horizontal position value indicates the position of the leftmost dot of the sprite's 24-dot width. The visible display is restricted to the 320 dot positions between positions 24 and 344. At position 0 the whole sprite is past the left edge of the visible screen. At position 1 the rightmost dot enters the display area, and at position 24 ($18) the entire sprite is displayed on screen. At position 321 ($141) the rightmost dot goes past the right edge of the visible display area, and at position 344 ($158) the whole sprite has moved out of sight, off the right edge of the screen.

These registers are all initialized to 0 at power-up.

53248 $D000 SP0X
Sprite 0 Horizontal Position

53249 $D001 SP0Y
Sprite 0 Vertical Position

53250 $D002 SP1X
Sprite 1 Horizontal Position

53251 $D003 SP1Y
Sprite 1 Vertical Position

53252 $D004 SP2X
Sprite 2 Horizontal Position

53253 $D005 SP2Y
Sprite 2 Vertical Position

53254 $D006 SP3X
Sprite 3 Horizontal Position

53255 $D007 SP3Y
Sprite 3 Vertical Position

53256 $D008 SP4X
Sprite 4 Horizontal Position

53257 $D009 SP4Y
Sprite 4 Vertical Position

53258 $D00A SP5X
Sprite 5 Horizontal Position

53259 $D00B SP5Y
Sprite 5 Vertical Position

53260 $D00C SP6X
Sprite 6 Horizontal Position

53261 $D00D SP6Y
Sprite 6 Vertical Position

53262 $D00E SP7X
Sprite 7 Horizontal Position

53263 $D00F SP7Y
Sprite 7 Vertical Position

53264 $D010 MSIGX
Most Significant Bits of Sprites 0-7 Horizontal Positions

Bit 0: Most significant bit of Sprite 0 horizontal position
Bit 1: Most significant bit of Sprite 1 horizontal position

Bit 2: Most significant bit of Sprite 2 horizontal position
Bit 3: Most significant bit of Sprite 3 horizontal position
Bit 4: Most significant bit of Sprite 4 horizontal position
Bit 5: Most significant bit of Sprite 5 horizontal position
Bit 6: Most significant bit of Sprite 6 horizontal position
Bit 7: Most significant bit of Sprite 7 horizontal position

Setting one of these bits to 1 adds 256 to the horizontal position of the corresponding sprite. Resetting one of these bits to 0 restricts the horizontal position of the corresponding sprite to a value of 255 or less.

53265 $D011 SCROLY

Vertical Fine Scrolling and Control Register

Bits 0-2: Fine scroll display vertically by X scan lines (0-7)
Bit 3: Select a 24-row or 25-row text display (1=25 rows, 0=24 rows)
Bit 4: Blank the entire screen to the same color as the border (0=blank)
Bit 5: Enable bitmap graphics mode (1=enable)
Bit 6: Enable extended color text mode (1=enable)
Bit 7: High bit (Bit 8) of raster compare register at 53266 ($D012)

This is one of the two important multifunction control registers on the VIC-II chip. Its default value is 155, which sets the high bit of the raster compare to 1, selects a 25-row text display, disables the blanking feature, and uses a vertical scrolling offset of three scan lines.

Bits 0-2. These bits control vertical fine scrolling of the screen display. This feature allows you to move the entire text display smoothly up and down, enabling the display area to act as a window, scrolling over a larger text or character graphics display.

Since each row of text is eight scan lines high, if you simply move each line of text up one row, the characters travel an appreciable distance each time they move, which gives the motion a jerky quality. This is called coarse scrolling, and you can see an example of it when LISTing a program that is too long to fit on the screen all at one time.

By placing a number from 1 to 7 into these three bits, you can move the whole screen display down by from 1 to 7 dot spaces. Stepping through values 1 to 7 allows you to smoothly make the transition from having a character appear in one row on the screen to having it appear in the next row. To demonstrate this, type in the following sample program, LIST it, and RUN.

```
10 FOR I= 1 TO 50:FOR J=0 TO 7
20 POKE 53265, (PEEK(53265)AND248) OR J:NEXTJ,I
30 FOR I= 1 TO 50:FOR J=7 TO 0 STEP-1
40 POKE 53265, (PEEK(53265)AND248) OR J:NEXTJ,I
```

As you can see, after the display has moved seven dot positions up or down, it starts over at its original position. In order to continue the scroll, you must do a coarse scroll every time the value of the scroll bits goes from 7 to 0, or from 0 to 7. This is accomplished by moving the display data for each line by 40 bytes in either direction, overwriting the data for the last line, and introducing a line of data at the opposite end of screen memory to replace it. Obviously, only a machine language program can move all of these lines quickly enough to maintain the effect of smooth motion. The following BASIC program, however, will give you an idea of what vertical fine scrolling is like.

```
10 POKE 53281,0:PRINTCHR$(5);CHR$(147)
20 FORI=1 TO 22:
30 PRINTTAB(15)CHR$(145)"{12 SPACES}":POKE 53265,P
   EEK(53265)AND248
40 WAIT53265,128:PRINTTAB(15)"I'M FALLING"
50 FOR J=1 TO 7
60 POKE53265,(PEEK(53265)AND248)+J
70 FORK=1TO50
80 NEXT K,J,I:RUN
```

Bit 3. This bit register allows you to select either the normal 25-line text display (by setting the bit to 1), or a shortened 24-row display (by resetting that bit to 0). This shortened display is created by extending the border to overlap the top or bottom row. The characters in these rows are still there; they are just covered up.

The shortened display is designed to aid vertical fine scrolling. It covers up the line into which new screen data is introduced, so that the viewer does not see the new data being moved into place.

However, unlike the register at 53270 ($D016) which shortens the screen by one character space on either side to aid horizontal scrolling in either direction, this register can blank only one vertical line at a time. In order to compensate, it blanks the top line when the three scroll bits in this register are set to 0, and shifts the blanking one scan line at a time as the value of these bits increases. Thus the bottom line is totally blanked when these bits are set to 7.

Bit 4. Bit 4 of this register controls the screen blanking feature. When this bit is set to 0, no data can be displayed on the screen. Instead, the whole screen will be filled with the color of the frame (which is controlled by the Border Color Register at 53280, $D020).

Screen blanking is useful because of the way in which the VIC-II chip interacts with the 6510 microprocessor. Since the VIC-II and the 6510 both have to address the same memory, they must share the system data bus. Sharing the data bus means that they must take turns whenever they want to address memory.

The VIC-II chip was designed so that it fetches most of the data it needs during the part of the cycle in which the 6510 is not using

the data bus. But certain operations, such as reading the 40 screen codes needed for each line of text from video memory, or fetching sprite data, require that the VIC-II chip get data at a faster rate than is possible just by using the off half of the 6510 cycle.

Thus, the VIC-II chip must delay the 6510 for a short amount of time while it is using the data bus to gather display information for text or bitmap graphics, and must delay it a little more if sprites are also enabled. When you set the screen blanking bit to 0, these delays are eliminated, and the 6510 processor is allowed to run at its full speed. This speeds up any processing task a little.

To demonstrate this, run the following short program. As you will see, leaving the screen on makes the processor run about 7 percent slower than when you turn it off. If you perform the same timing test on the VIC-20, you will find that it runs at the same speed with its screen on as the 64 does with its screen off. And the same test on a PET will run substantially slower.

```
10 PRINT CHR$(147);TAB(13);"TIMING TEST":PRINT:TI$
   ="000000":GOTO 30
20 FOR I=1 TO 10000:NEXT I:RETURN
30 GOSUB 20:DISPLAY=TI
40 POKE 53265,11:TI$="000000"
50 GOSUB 20:NOSCREEN=TI:POKE 53265,27
60 PRINT "THE LOOP TOOK";DISPLAY;" JIFFIES"
70 PRINT "WITH NO SCREEN BLANKING":PRINT
80 PRINT "THE LOOP TOOK";NOSCREEN;" JIFFIES"
90 PRINT "WITH SCREEN BLANKING":PRINT
100 PRINT "SCREEN BLANKING MADE THE PROCESSOR"
110 PRINT "GO";DISPLAY/NOSCREEN*100-100;"PERCENT F
    ASTER"
```

The above explanation accounts for the screen being turned off during tape read and write operations. The timing of these operations is rather critical, and would be affected by even the relatively small delay caused by the video chip. It also explains why the 64 has difficulty loading programs from an unmodified 1540 Disk Drive, since the 1540 was set up to transfer data from the VIC-20, which does not have to contend with these slight delays.

If you turn off the 64 display with a POKE 53265,PEEK(53265) AND 239, you will be able to load programs correctly from an old 1540 drive. The new 1541 drive transfers data at a slightly slower rate in the default setting, and can be set from software to transfer it at the higher rate for the VIC-20.

Bit 5. Setting Bit 5 of this register to 1 enables the bitmap graphics mode. In this mode, the screen area is broken down into 64,000 separate dots of light, 320 dots across by 200 dots high. Each dot corresponds to one bit of display memory. If the bit is set to 1, the dot will be displayed in the foreground color. If the bit is reset to

0, it will be displayed in the background color. This allows the display of high-resolution graphics images for games, charts, and graphs, etc.

Bitmapping is a common technique for implementing high-resolution graphics on a microcomputer. There are some features of the Commodore system which are unusual, however.

Most systems display screen memory sequentially; that is, the first byte controls the display of the first eight dots in the upper-left corner of the screen, the second byte controls the eight dots to the right of that, etc. In the Commodore system, display memory is laid out more along the lines of how character graphics dot-data is arranged.

The first byte controls the row of eight dots in the top-left corner of the screen, but the next byte controls the eight dots below that, and so on until the ninth byte. The ninth byte controls the eight dots directly to the right of those controlled by the first byte of display memory. It is exactly the same as if the screen were filled with 1000 programmable characters, with display memory taking the place of the character dot-data.

The 64's bitmap graphics mode also resembles character graphics in that the foreground color of the dots is set by a color map (although it does not use the Color RAM for this purpose). Four bits of each byte of this color memory control the foreground color of one of these eight-byte groups of display memory (which form an 8 by 8 grid of 64 dots). Unlike character graphics, however, the other four bits control the background color that will be seen in the eight-byte display group where a bit has a value of 0.

Setting up a bitmap graphics screen is somewhat more complicated than just setting this register bit to 1. You must first choose a location for the display memory area, and for the color memory area. The display memory area will be 8192 bytes long (8000 of which are actually used for the display) and can occupy only the first or the second half of the 16K space which the VIC-II chip can address.

Each byte of bitmap graphics color memory uses four bits for the background color as well as four bits for the foreground color. Therefore, the Color RAM nybbles at 55296 ($D800), which are wired for four bits only, cannot be used. Another RAM location must therefore be found for color memory.

This color memory area will take up 1K (1000 bytes of which are actually used to control the foreground and background colors of the dots), and must be in the opposite half of VIC-II memory as the display data. Since bitmap graphics require so much memory for the display, you may want to select a different 16K bank for VIC-II memory (see the discussion of things to consider in selecting a VIC-II memory bank at location 56576, $DD00).

To keep things simple, however, let's assume that you have

selected to use the default bank of VIC-II memory, which is the first 16K. You would have to select locations 8192-16383 ($2000-$3FFF) for screen memory, because the VIC-II chip sees an image of the character ROM in the first half of the 16K block (at locations 4096-8191, $1000-$1FFF). Color memory could be placed at the default location of text display memory, at 1024-2047 ($400-$7FF). Placement of bitmap display and color memory is controlled by the VIC Memory Control Register at 53272 ($D018).

When in bitmap mode, the lower four bits of this register, which normally control the base address of character dot-data, now control the location of the 8K bitmap. Only Bit 3 is significant. If it is set to 1, the graphics display memory will be in the second 8K of VIC-II memory (in this case, starting at 8192, $2000). If that bit contains a 0, the first 8K will be used for the bitmap. The upper four bits of this register, which normally control the location of the Video Display Matrix, are used in bitmap mode to establish the location of the color map within the VIC-II address space. These four bits can hold a number from 0 to 15, which indicates on which 1K boundary the color map begins. For example, if color memory began at 1024 (1K), the value of these four bits would be 0001.

Once the bitmap mode has been selected, and the screen and color memory areas set up, you must establish a method for turning each individual dot on and off. The conventional method for identifying each dot is to assign it a horizontal (X) position coordinate and a vertical (Y) coordinate.

Horizontal position values will range from 0 to 319, where dot 0 is at the extreme left-hand side of the screen, and dot 319 at the extreme right. Vertical positions will range from 0 to 199, where dot 0 is on the top line, and dot 199 on the bottom line.

Because of the unusual layout of bitmap screen data on the 64, it is fairly easy to transfer text characters to a bitmap screen, but it is somewhat awkward finding the bit which affects the screen dot having a given X-Y coordinate. First, you must find the byte BY in which the bit resides, and then you must POKE a value into that byte which turns the desired bit on or off. Given that the horizontal position of the dot is stored in the variable X, its vertical position is in the variable Y, and the base address of the bitmap area is in the variable BASE, you can find the desired byte with the formula:

```
BY=BASE+40*(Y AND 248)+(Y AND 7)+(X AND 504)
```

To turn on the desired dot,

```
POKE BY, PEEK(BY) OR (2↑(NOTX AND 7))
```

To turn the dot off,

```
POKE BY, PEEK(BY) AND (255-2↑(NOTX AND 7))
```

The exponentiation function takes a lot of time. To speed things up,

an array can be created, each of whose elements corresponds to a power of two.

FOR I=0 TO 7: BIT(I)=2↑I:NEXT

After this is done, the expression 2↑(I) can be replaced by BI(I).

The following sample program illustrates the bit-graphics concepts explained above, and serves as a summary of that information.

```
10 FOR I=0 TO 7:BI(I)=2↑I:NEXT: REM SET UP ARRAY O
   F POWERS OF 2 (BIT VALUES)
20 BASE=2*4096:POKE53272,PEEK(53272)OR8:REM PUT BI
   T MAP AT 8192
30 POKE53265,PEEK(53265)OR32:REM ENTER BIT MAP MOD
   E
40 A$="":FOR I=1 TO 37:A$=A$+"C":NEXT:PRINT CHR$(1
   9);
50 FOR I=1 TO 27:PRINTA$;:NEXT:POKE 2023,PEEK(2022
   ): REM SET COLOR MAP
60 A$="":FOR I=1 TO 128:A$=A$+"@":NEXT:FOR I=32 TO
    63 STEP 2
70 POKE 648,I:PRINT CHR$(19);A$;A$;A$;A$:NEXT:POKE
    648,4:REM CLEAR HI-RES SCREEN
80 FORY=0TO199STEP.5:REM FROM THE TOP OF THE SCREE
   N TO THE BOTTOM
90 X=INT(160+40*SIN(Y/10)): REM SINE WAVE SHAPE
100 BY=BASE+40*(Y AND 248)+(Y AND 7)+(X AND 504):
    {SPACE}REM FIND HI-RES BYTE
110 POKEBY,PEEK(BY)OR(BI(NOT X AND 7)):NEXT Y: REM
     POKE IN BIT VALUE
120 GOTO 120: REM LET IT STAY ON SCREEN
```

As you can see, using BASIC to draw in bit-graphics mode is somewhat slow and tedious. Machine language is much more suitable for bit-graphics plotting. For a program that lets you replace some BASIC commands with hi-res drawing commands, see the article "Hi-Res Graphics Made Simple," by Paul F. Schatz, in *COMPUTE!'s First Book of Commodore 64 Sound and Graphics.*

There is a slightly lower resolution bitmap graphics mode available which offers up to four colors per 8 by 8 dot matrix. To enable this mode, you must set the multicolor bit (Bit 4 of 53270, $D016) while in bitmap graphics mode. For more information on this mode, see the entry for the multicolor enable bit.

Bit 6. This bit of this register enables extended background color mode. This mode lets you select the background color of each text character, as well as its foreground color. It is able to increase the number of background colors displayed, by reducing the number of characters that can be shown on the screen.

Normally, 256 character shapes can be displayed on the screen.

You can see them either by using the PRINT statement or by POKEing a display code from 0 to 255 into screen memory. If the POKEing method is used, you must also POKE a color code from 0 to 15 into color memory (for example, if you POKE 1024,1, and POKE 55296,1, a white A appears in the top-left corner of the screen).

The background color of the screen is determined by Background Color Register 0, and you can change this color by POKEing a new value to that register, which is located at 53281 ($D021). For example, POKE 53281,0 creates a black background.

When extended background color mode is activated, however, only the first 64 shapes found in the table of screen display codes can be displayed on the screen. This group includes the letters of the alphabet, numerals, and punctuation marks. If you try to print on the screen a character having a higher display code, the shape displayed will be from the first group of 64, but that character's background color will no longer be determined by the register at 53281 ($D021). Instead, it will be determined by one of the other background color registers.

When in extended background color mode, characters having display codes 64-127 will take their background color from register 1, at location 53282 ($D022). These characters include various SHIFTed graphics characters. Those with codes 128–191 will have their background colors determined by register 2, at 53283 ($D023). These include the reversed numbers, letters, and punctuation marks. Finally, characters with codes 192–255 will use register 3, at 53284 ($D024). These are the reversed graphics characters.

Let's try an experiment to see just how this works. First, we will put the codes for four different letters in screen memory:

```
FOR I=0 TO 3: POKE 1230+(I*8),I*64+1: POKE 55502+(I*8),1: NEXT
```

Four white letters should appear on the screen, an A, a shifted A, a reversed A, and a reversed, shifted A, all on a blue background. Next, we will put colors in the other background color registers:

```
POKE 53282,0: POKE 53283,2: POKE 53284,5
```

This sets these registers to black, red, and green, respectively. Finally, we will activate extended color mode by setting Bit 6 of the VIC-II register at location 53265 to a 1. The BASIC statement that turns this mode on is:

```
POKE 53265, PEEK(53265) OR 64
```

Notice that two things happened. First, all of the letters took on the same shape, that of the letter A. Second, each took on the background color of a different color register. To get things back to normal, turn off extended color mode with this statement:

POKE 53265, PEEK(53265) AND 191

Extended color mode can be a very useful enhancement for your text displays. It allows the creation of windows. These windows, because of their different background colors, make different bodies of text stand out as visually distinct from one another. For example, a text adventure program could have one window to display the player's current location, one to show an inventory of possessions, and one to accept commands for the next move.

In this mode the background color of these windows can be changed instantly, just by POKEing a new value to the color register. This technique lends itself to some dramatic effects. A window can be flashed to draw attention to a particular message at certain times. And varying the foreground color can make either the window or the message vanish and reappear later.

There are, however, a couple of problems involved in using these windows. The character shape that you want to use might not have a screen code of less than 64. In that case, the only solution is to define your own character set, with the shape you want in one of the first 64 characters.

Another problem is that characters within a PRINT statement in your program listing are not always going to look the same on the screen. Having to figure out what letter to print to get the number 4 with a certain background color can be very inconvenient. The easiest solution to this problem is to have a subroutine do the translation for you. Since letters will appear normally in window 1, and window 3 characters are simply window 1 characters reversed, you will only have problems with characters in windows 2 and 4. To convert these characters, put your message into A$, and use the following subroutine:

```
500 B$="":FOR I=1 TO LEN(A$):B=ASC(MID$(A$,I,1))
510 B=B+32:IF B<96 THEN B=B+96
520 B$=B$+CHR$(B):NEXT I:RETURN
```

This subroutine converts each letter to its ASCII equivalent, adds the proper offset, and converts it back to part of the new string, B$. When the conversion is complete, B$ will hold the characters necessary to PRINT that message in window 2. For window 4, PRINT CHR$(18); B$; CHR$(146). This will turn reverse video on before printing the string, and turn it off afterwards.

A practical demonstration of the technique for setting up windows is given in the sample program below. The program sets up three windows, and shows them flashing, appearing and disappearing.

```
5 DIM RO$(25):RO$(0)=CHR$(19):FOR I=1 TO 24:RO$(I)
  =RO$(I-1)+CHR$(17):NEXT
```

```
10 POKE 53265,PEEK(53265) OR 64
20 POKE 53280,0: POKE 53281,0:POKE 53282,1:POKE 53
   283,2:POKE 53284,13
25 OP$=CHR$(160):FOR I=1 TO 4:OP$=OP$+OP$:NEXTI:PR
   INTCHR$(147);RO$(3);
30 FOR I=1 TO10:PRINTTAB(1);CHR$(18);"{15 SPACES}"
   ;TAB(23);OP$:NEXT
40 PRINT CHR$(146):PRINT:PRINT:FOR I=1 TO 4:PRINTO
   P$;OP$;OP$;OP$;OP$;:NEXTI
50 PRINT RO$(5);CHR$(5);CHR$(18);TAB(2);"A RED WIN
   DOW"
60 PRINT CHR$(18);TAB(2);"COULD BE USED"
70 PRINT CHR$(18);TAB(2);"FOR ERROR"
80 PRINT CHR$(18);TAB(2);"MESSAGES"
100 A$="A GREEN WINDOW":GOSUB 300:PRINT RO$(5);CHR
    $(144);CHR$(18);TAB(24);B$
110 A$="COULD BE USED":GOSUB 300:PRINTTAB(24);CHR$
    (18);B$
120 A$="TO GIVE":GOSUB 300:PRINTTAB(24);CHR$(18);B
    $
130 A$="INSTRUCTIONS":GOSUB 300:PRINTTAB(24);CHR$(
    18);B$
140 PRINT CHR$(31);RO$(19);
150 A$="{2 SPACES}WHILE THE MAIN WINDOW COULD BE U
    SED":GOSUB300:PRINT B$
160 A$="{2 SPACES}FOR ACCEPTING COMMANDS.":GOSUB30
    0:PRINT B$
170 FOR I=1 TO 5000:NEXT I: POKE 53284,0
180 FOR I=1 TO 5:FOR J=1 TO 300:NEXT J:POKE 53282,
    15
190 FOR J=1 TO 300:NEXT J:POKE 53282,1
200 NEXT I: POKE 53283,-2*(PEEK(53283)=240):POKE 5
    3284,-13*(PEEK(53284)=240)
210 GOTO 180
300 B$="":FOR I=1TOLEN(A$):B=ASC(MID$(A$,I,1))
310 B=B+32:IFB<96THENB=B+96
320 B$=B$+CHR$(B):NEXTI:RETURN
```

Bit 7. Bit 7 of this register is the high-order bit (Bit 8) of the Raster Compare register at 53266 ($D012). Even though it is located here, it functions as part of that register (see the description below for more information on the Raster Compare register).

Machine language programmers should note that its position here at Bit 7 allows testing this bit with the Negative flag. Since scan lines above number 256 are all off the screen, this provides an easy way to delay changing the graphics display until the scan is in the vertical blanking interval and the display is no longer being drawn:

```
LOOP    LDA     $D011
        BPL     LOOP
```

Sprites should always be moved when the raster is scanning off-screen, because if they are moved while they are being scanned, their shapes waver slightly.

The BASIC equivalent of the program fragment above is the statement WAIT 53265,128, but BASIC is usually not fast enough to execute the next statement while still in the blanking interval.

53266 $D012 RASTER

Read Current Raster Scan Line/Write Line to Compare for Raster IRQ

The Raster Compare register has two different functions, depending on whether you are reading from it or writing to it. When this register is read, it tells which screen line the electron beam is currently scanning.

There are 262 horizontal lines which make up the American (NTSC) standard display screen (312 lines in the European or PAL standard screen). Every one of these lines is scanned and updated 60 times per second. Only 200 of these lines (numbers 50-249) are part of the visible display.

It is sometimes helpful to know just what line is being scanned, because changing screen graphics on a particular line while that line is being scanned may cause a slight disruption on the screen. By reading this register, it is possible for a machine language program to wait until the scan is off the bottom of the screen before changing the graphics display.

It is even possible for a machine language program to read this register, and change the screen display when a certain scan line is reached. The program below uses this technique to change the background color in midscreen, in order to show all 256 combinations of foreground and background text colors at once.

```
40 FOR I=49152 TO 49188: READ A: POKE I,A: NEXT:PO
   KE 53280,11
50 PRINT CHR$(147):FOR I=1024 TO I+1000: POKE I,16
   0: POKE I+54272,11:NEXTI
60 FOR I=0 TO 15: FOR J=0 TO 15
70 P=1196+(40*I)+J: POKE P,J+1: POKE P+54272,J: NE
   XT J,I
80 PRINT TAB(15)CHR$(5)"COLOR CHART":FOR I=1 TO 19
   :PRINT:NEXT
85 PRINT"THIS CHART SHOWS ALL COMBINATIONS OF
   {3 SPACES}"
86 PRINT "FOREGROUND AND BACKGROUND COLORS.
   {6 SPACES}"
87 PRINT "FOREGROUND INCREASES FROM LEFT TO RIGHT"
88 PRINT "BACKGROUND INCREASES FROM TOP TO BOTTOM"
   ;
```

```
90 SYS 12*4096
100 DATA 169,90,133,251,169,0,141,33,208,162,15,12
    0,173,17,208,48
105 DATA 251,173,18,208
110 DATA 197,251,208,249,238,33,208,24,105,8,133,2
    51,202,16,233,48,219
```

Writing to this register designates the comparison value for the Raster Compare Interrupt. When that interrupt is enabled, a maskable interrupt request will be issued every time the electron beam scan reaches the scan line whose number was written here. This is a much more flexible technique for changing the display in midscreen than reading this register as the sample program above does. That technique requires that the program continuously watch the Raster Register, while the interrupt method will call the program when the time is right to act. For more information on raster interrupts, see the entry for the Interrupt Mask Register, 53274 ($D01A).

It is very important to remember that this register requires nine bits, and that this location holds only eight of those bits (the ninth is Bit 7 of 53265, $D011). If you forget to read or write to the ninth bit, your results could be in error by a factor of 256.

For example, some early programs written to demonstrate the raster interrupt took for granted that the ninth bit of this register would be set to 0 on power-up. When a later version of the Kernal changed this initial value to a 1, their interrupt routines, which were supposed to set the raster interrupt to occur at scan line number 150, ended up setting it for line number 406 instead. Since the scan line numbers do not go up that high, no interrupt request was ever issued and the program did not work.

Location Range: 53267-53268 ($D013-$D014)

Light Pen Registers

A light pen is an input device that can be plugged into joystick Con trol Port #1. It is shaped like a pen and has a light-sensitive device at its tip that causes the trigger switch of the joystick port to close at the moment the electron beam that updates the screen display strikes it. The VIC-II chip keeps track of where the beam is when that happens, and records the corresponding horizontal and vertical screen coordinates in the registers at these locations.

A program can read the position at which the light pen is held up to the screen. The values in these registers are updated once every screen frame (60 times per second). Once the switch is closed and a value written to these registers, the registers are latched, and subsequent switch closings during the same screen frame will not be recorded.

A given light pen may not be entirely accurate (and the operator may not have a steady hand). It is probably wise to average the positions returned from a number of samplings, particularly when using a machine language driver.

53267 $D013 LPENX

Light Pen Horizontal Position

This location holds the horizontal position of the light pen. Since there are only eight bits available (which give a range of 256 values) for 320 possible horizontal screen positions, the value here is accurate only to every second dot position. The number here will range from 0 to 160 and must be multiplied by 2 in order to get a close approximation of the actual horizontal dot position of the light pen.

53268 $D014 LPENY

Light Pen Vertical Position

This location holds the vertical position of the light pen. Since there are only 200 visible scan lines on the screen, the value in this register corresponds exactly to the current raster scan line.

53269 $D015 SPENA

Sprite Enable Register

Bit 0: Enable Sprite 0 (1=sprite is on, 0=sprite is off)
Bit 1: Enable Sprite 1 (1=sprite is on, 0=sprite is off)
Bit 2: Enable Sprite 2 (1=sprite is on, 0=sprite is off)
Bit 3: Enable Sprite 3 (1=sprite is on, 0=sprite is off)
Bit 4: Enable Sprite 4 (1=sprite is on, 0=sprite is off)
Bit 5: Enable Sprite 5 (1=sprite is on, 0=sprite is off)
Bit 6: Enable Sprite 6 (1=sprite is on, 0=sprite is off)
Bit 7: Enable Sprite 7 (1=sprite is on, 0=sprite is off)

In order for any sprite to be displayed, the corresponding bit in this register must be set to 1 (the default for this location is 0). Of course, just setting this bit alone will not guarantee that a sprite will be shown on the screen. The Sprite Data Pointer must indicate a data area that holds some values other than 0. The Sprite Color Register must also contain a value other than that of the background color. In addition, the Sprite Horizontal and Vertical Position Registers must be set for positions that lie within the visible screen range in order for a sprite to appear on screen.

53270 $D016 SCROLX

Horizontal Fine Scrolling and Control Register

Bits 0-2: Fine scroll display horizontally by X dot positions (0-7)
Bit 3: Select a 38-column or 40-column text display (1=40 columns, 0=38 columns)

Bit 4: Enable multicolor text or multicolor bitmap mode (1=multicolor on, 0=multicolor off)
Bit 5: Video chip reset (0=normal operation, 1=video completely off)
Bits 6-7: Unused

This is one of the two important multifunction control registers on the VIC-II chip. On power-up, it is set to a default value of 8, which means that the VIC chip Reset line is set for a normal display, Multicolor Mode is disabled, a 40-column text display is selected, and no horizontal fine-scroll offset is used.

Bits 0-2. The first three bits of this chip control vertical fine scrolling of the screen display. This feature allows you to smoothly move the entire text display back and forth, enabling the display area to act as a window, scrolling over a larger text or character graphics display.

Since each text character is eight dots wide, moving each character over one whole character position (known as *coarse scrolling*) is a relatively big jump, and the motion looks jerky. By placing a number from 1 to 7 into these three bits, you can move the whole screen display from one to seven dot spaces to the right.

Stepping through values 1 to 7 allows you to smoothly make the transition from having a character appear at one screen column to having it appear at the next one over. To demonstrate this, type in the following program, LIST, and RUN it.

```
10 FOR I= 1 TO 50:FOR J=0 TO 7
20 POKE 53270, (PEEK(53270)AND248) OR J:NEXTJ,I
30 FOR I= 1 TO 50:FOR J=7 TO 0 STEP-1
40 POKE 53270, (PEEK(53270)AND248) OR J:NEXTJ,I
```

As you can see, after the display has moved over seven dots, it starts over at its original position. In order to continue the scroll, you must do a coarse scroll every time the value of the scroll bits goes from 7 to 0, or from 0 to 7. This is accomplished by moving each byte of display data on each line over one position, overwriting the last character, and introducing a new byte of data on the opposite end of the screen line to replace it.

Obviously, only a machine language program can move all of these bytes quickly enough to maintain the effect of smooth motion. The following BASIC program, however, will give you an idea of what the combination of fine and coarse scrolling looks like.

```
10 POKE 53281,0:PRINTCHR$(5);CHR$(147):FOR I=1 TO
   {SPACE}5:PRINTCHR$(17):NEXT
20 FORI=1 TO 30
30 PRINT TAB(I-1)"{UP}{10 SPACES}{UP}"
40 WAIT53265,128:POKE53270,PEEK(53270)AND248:PRINT
   TAB(I)"AWAY WE GO"
```

```
5Ø FOR J=1 TO 7
6Ø POKE53270,(PEEK(53270)AND248)+J
7Ø FORK=1TO3Ø-I
8Ø NEXT K,J,I:RUN
```

Changing the value of the three horizontal scroll bits will affect the entire screen display. If you wish to scroll only a portion of the screen, you will have to use raster interrupts (see 53274, $D01A below) to establish a scroll zone, change the value of these scroll bits only when that zone is being displayed, and change it back to 0 afterward.

Bit 3. Bit 3 of this register allows you to cover up the first and last columns of the screen display with the border. Since the viewers cannot see the characters there, they will not be able to see you insert a new character on the end of the line when you do coarse scrolling (see explanation of Bits 0-2 above).

Setting this bit to 1 enables the normal 40-column display, while resetting it to 0 changes the display to 38 columns. This is purely a cosmetic aid, and it is not necessary to change the screen to the smaller size to use the scrolling feature.

Bit 4. This bit selects multicolor graphics. The effect of setting this bit to 1 depends on whether or not the bitmap graphics mode is also enabled.

If you are not in bitmap mode, and you select multicolor text character mode by setting this bit to 1, characters with a color nybble whose value is less than 8 are displayed normally. There will be one background color and one foreground color. But each dot of a character with a color nybble whose value is over 7 can have any one of four different colors.

The two colors in Background Color Registers 1 and 2 (53282-3, $D022–3) are available in addition to the colors supplied by the Color RAM. The price of these extra colors is a reduction in horizontal resolution. Instead of each bit controlling one dot, in multicolor mode a pair of bits control the color of a larger dot. Thus, each character is eight dots across; multicolor characters are only four double-width dots across. A bit pattern of 00 will put the color from Background Color Register 0 into that dot. A pattern of 11 will light it with the color from the lower three bits of color RAM. Patterns of 01 and 10 will select the colors from Background Color Registers 1 and 2, respectively, for the double-width dot.

You can see the effect that setting this bit has by typing in the following BASIC command line:

```
POKE 53270,PEEK(53270)OR16: PRINT CHR$(149)"THIS IS
    MULTICOLOR MODE"
```

It is obvious from this example that the normal set of text characters was not made to be used in multicolor mode. In order to take advan-

tage of this mode, you will need to design custom four-color characters. For more information, see the alternate entry for 53248 ($D000), the Character Generator ROM.

If the multicolor and bitmap enable bits are both set to 1, the result is multicolor bitmap mode. As in multicolor text mode, pairs of graphics data bits are used to set each dot in a 4 by 8 matrix to one of four colors. This results in a reduction of the horizontal resolution to 160 double-wide dots across. But while text multicolor mode allows only one of the four colors to be set individually for each 4 by 8 dot area, bitmap multicolor mode allows up to three different colors to be individually selected in each 4 by 8 dot area. The source of the dot color for each bit-pair combination is shown below:

00 Background Color Register 0 (53281, $D021)
01 Upper four bits of Video Matrix
10 Lower four bits of Video Matrix
11 Color RAM nybble (area starts at 55296, $D800)

The fact that bit-pairs are used in this mode changes the strategy for plotting somewhat. In order to find the byte BY in which the desired bit-pair resides, you must multiply the horizontal position X, which has a value of 0-159, by 2, and then use the same formula as for hi-res bitmap mode.

Given that the horizontal position (0-159) of the dot is stored in the variable X, its vertical position is in the variable Y, and the base address of the bitmap area is in the variable BASE. You can find the desired byte with the formula:

BY=BASE+(Y AND 248)*40+(Y AND 7)+(2*X AND 504)

Setting the desired bit-pair will depend on what color you choose. First, you must set up an array of bit masks.

CA(0)=1:CA(1)=4:CA(2)=16:CA(3)=64

To turn on the desired dot, select a color CO from 0 to 3 (representing the color selected by the corresponding bit pattern) and execute the following statement:

BI=(NOT X AND 3): POKE BY, PEEK(BY) AND (NOT 3*CA(BI)) OR (CO*CA(BI))

The following program will demonstrate this technique:

```
10 CA(0)=1:CA(1)=4:CA(2)=16:CA(3)=64:REM ARRAY FOR
    BIT PAIRS
20 BASE=2*4096:POKE53272,PEEK(53272)OR8:REM PUT BI
   T MAP AT 8192
30 POKE53265,PEEK(53265)OR32:POKE 53270,PEEK(53270
   )OR16:REM MULTI-COLOR BIT MAP
40 A$="":FOR I=1 TO 37:A$=A$+"C":NEXT:PRINT CHR$(1
   9);
```

```
50 FOR I=1 TO 27:PRINTA$;:NEXT:POKE 2023,PEEK(2022
   ): REM SET COLOR MAP
60 A$="":FOR I=1 TO 128:A$=A$+"@":NEXT:FOR I=32 TO
    63 STEP 2
70 POKE648,I:PRINTCHR$(19);A$;A$;A$;A$:NEXT:POKE64
   8,4:REM CLR HI-RES SCREEN
80 FOR CO=1 TO3: FORY=0TO199STEP.5:REM FROM THE TO
   P OF THE SCREEN TO THE BOTTOM
90 X=INT(10*CO+50+15*SIN(CO*45+Y/10)): REM SINE WA
   VE SHAPE
100 BY=BASE+40*(Y AND 248)+(Y AND 7)+(X*2 AND 504)
    : REM FIND HI-RES BYTE
110 BI=(NOT X AND 3):POKE BY,PEEK(BY) AND (NOT 3*C
    A(BI)) OR (CO*CA(BI))
120 NEXT Y,CO
130 GOTO 130: REM LET IT STAY ON SCREEN
```

Bit 5. Bit 5 controls the VIC-II chip Reset line. Setting this bit to 1 will completely stop the video chip from operating. On older 64s, the screen will go black. It should always be set to 0 to insure normal operation of the chip.

Bits 6 and 7. These bits are not used.

53271 $D017 YXPAND

Sprite Vertical Expansion Register

Bit 0: Expand Sprite 0 vertically (1=double height, 0=normal height)
Bit 1: Expand Sprite 1 vertically (1=double height, 0=normal height)
Bit 2: Expand Sprite 2 vertically (1=double height, 0=normal height)
Bit 3: Expand Sprite 3 vertically (1=double height, 0=normal height)
Bit 4: Expand Sprite 4 vertically (1=double height, 0=normal height)
Bit 5: Expand Sprite 5 vertically (1=double height, 0=normal height)
Bit 6: Expand Sprite 6 vertically (1=double height, 0=normal height)
Bit 7: Expand Sprite 7 vertically (1=double height, 0=normal height)

This register can be used to double the height of any sprite. When the bit in this register that corresponds to a particular sprite is set to 1, each dot of the 24 by 21 sprite dot matrix will become two raster scan lines high instead of one.

53272 $D018 VMCSB

VIC-II Chip Memory Control Register

Bit 0: Unused
Bits 1-3: Text character dot-data base address within VIC-II address space
Bits 4-7: Video matrix base address within VIC-II address space

This register affects virtually all graphics operations. It determines the base address of two very important data areas, the Video Matrix, and the Character Dot-Data area.

Bits 1-3. These bits are used to set the location of the Character Dot-Data area. This area is where the data that defines the shapes of the characters displayed on the screen is stored (for more information on character shape data, see the alternate entry for location 53248 ($D000), the Character Generator ROM).

Bits 1-3 can represent any even number from 0 to 14. That number stands for the even 1K offset of the character data area from the beginning of VIC-II memory. For example, if these bits are all set to 0, it means that character memory occupies the first 2K of VIC-II memory. If they equal 2, the data area starts 2*1K (2*1024) or 2048 bytes from the beginning of VIC memory.

The default value of this nybble is 4. This sets the address of the Character Dot-Data area to 4096 ($1000), which is the starting address of where the VIC-II chip addresses the Character ROM. The normal character set which contains uppercase and graphics occupies the first 2K of that ROM. The alternate character set which contains both upper- and lowercase letters uses the second 2K. Therefore, to shift to the alternate character set, you must change the value of this nybble to 6, with a POKE 53272,PEEK(53272)OR2. To change it back, POKE 53272,PEEK(53272)AND253.

In bitmap mode, the lower nybble controls the location of the bitmap screen data. Since this data area can start only at an offset of 0 or 8K from the beginning of VIC-II memory, only Bit 3 of the Memory Control Register is significant in bitmap mode. If Bit 3 holds a 0, the offset is 0, and if it holds a 1, the offset is 8192 (8K).

Bits 4-7. This nybble determines the starting address of the Video Matrix area. This is the 1024-byte area of memory which contains the screen codes for the text characters that are displayed on the screen. In addition, the last eight bytes of this area are used as pointers which designate which 64-byte block of VIC-II memory will be used as shape data for each sprite.

These four bits can represent numbers from 0 to 15. These numbers stand for the offset (in 1K increments) from the beginning of VIC-II memory to the Video Matrix.

For example, the default bit pattern is 0001. This indicates that the Video Matrix is offset by 1K from the beginning of VIC-II memory,

the normal starting place for screen memory. Remember, though, the bit value of this number will be 16 times what the bit pattern indicates, because we are dealing with Bits 4-7. Therefore, the 0001 in the upper nybble has a value of 16.

Using this register, we can move the start of screen memory to any 1K boundary within the 16K VIC-II memory area. Just changing this register, however, is not enough if you want to use the BASIC line editor. The editor looks to location 648 ($288) to determine where to print screen characters.

If you just change the location of the Video Matrix without changing the value in 648, BASIC will continue to print characters in the memory area starting at 1024, even though that area is no longer being displayed. The result is that you will not be able to see anything that you type in on the keyboard. To fix this, you must POKE 648 with the page number of the starting address of screen memory (page number=location/256). Remember, the actual starting address of screen memory depends not only on the offset from the beginning of VIC-II memory in the register, but also on which bank of 16K is used for VIC-II memory.

For example, if the screen area starts 1024 bytes from the beginning of VIC-II memory, and the video chip is using Bank 2 (32768-49151), the actual starting address of screen memory is 32768+1024=33792 ($8400). For examples of how to change the video memory area, and of how to relocate the screen, see the entry for 56576 ($DD00).

53273 $D019 VICIRQ

VIC Interrupt Flag Register

Bit 0: Flag: Is the Raster Compare a possible source of an IRQ? (1=yes)
Bit 1: Flag: Is a collision between a sprite and the normal graphics display a possible source of an IRQ? (1=yes)
Bit 2: Flag: Is a collision between two sprites a possible source of an IRQ? (1=yes)
Bit 3: Flag: Is the light pen trigger a possible source of an IRQ? (1=yes)
Bits 4-6: Not used
Bit 7: Flag: Is there any VIC-II chip IRQ source which could cause an IRQ? (1=yes)

The VIC-II chip is capable of generating a maskable request (IRQ) when certain conditions relating to the video display are fulfilled. Briefly, the conditions that can cause a VIC-II chip IRQ are:

1. The line number of the current screen line being scanned by the raster is the same as the line number value written to the Raster Register (53266, $D012).

2. A sprite is positioned at the same location where normal graphics data are being displayed.

3. Two sprites are positioned so that they are touching.

4. The light sensor on the light pen has been struck by the raster beam, causing the fire button switch on joystick Control Port #1 to close (pressing the joystick fire button can have the same effect).

When one of these conditions is met, the corresponding bit in this status register is set to 1 and latched. That means that as long as the corresponding enable bit in the VIC IRQ Mask register is set to 1, an IRQ request will be generated, and any subsequent fulfillment of the same condition will be ignored until the latch is cleared.

This allows you to preserve multiple interrupt requests if more than one of the interrupt conditions is met at a time. In order to keep an IRQ source from generating another request after it has been serviced, and to enable subsequent interrupt conditions to be detected, the interrupt service routine must write a 1 to the corresponding bit. This will clear the latch for that bit. The default value written to this register is 15, which clears all interrupts.

There is only one IRQ vector that points to the address of the routine that will be executed when an IRQ interrupt occurs. The same routine will therefore be executed regardless of the source of the interrupt. This status register provides a method for that routine to check what the source of the IRQ was, so that the routine can take appropriate action. First, the routine can check Bit 7. Anytime that any of the other bits in the status register is set to 1, Bit 7 will also be set. Therefore, if that bit holds a 1, you know that the VIC-II chip requested an IRQ (the two CIA chips which are the other sources of IRQ interrupts can be checked in a similar manner). Once it has been determined that the VIC chip is responsible for the IRQ, the individual bits can be tested to see which of the IRQ conditions have been met.

For more information, and a sample VIC IRQ program, see the following entry.

53274 $D01A IRQMSK

IRQ Mask Register

Bit 0: Enable Raster Compare IRQ (1=interrupt enabled)
Bit 1: Enable IRQ to occur when sprite collides with display of normal graphics data (1=interrupt enabled)
Bit 2: Enable IRQ to occur when two sprites collide (1=interrupt enabled)
Bit 3: Enable light pen to trigger an IRQ (1=interrupt enabled)
Bits 4-7: Not used

This register is used to enable an IRQ request to occur when one of the VIC-II chip interrupt conditions is met. In order to understand

what that means, and how these interrupts can extend the range of options available to a programmer, you must first understand what an interrupt is.

An interrupt is a signal given to the microprocessor (the brains of the computer) that tells it to stop executing its machine language program (for example, BASIC), and to work on another program for a short time, perhaps only a fraction of a second. After finishing the interrupt program, the computer goes back to executing the main program, just as if there had never been a detour.

Bit 0. This bit enables the Raster Compare IRQ. The conditions for this IRQ are met when the raster scan reaches the video line indicated by the value written to the Raster Register at 53266 ($D012) and Bit 7 of 53265 ($D011). Again, an explanation of the terminology is in order.

In the normal TV display, a beam of electrons (raster) scans the screen, starting in the top-left corner, and moving in a straight line to the right, lighting up appropriate parts of the screen line on the way. When it comes to the right edge, the beam moves down a line, and starts again from the left. There are 262 such lines that are scanned by the 64 display, 200 of which form the visible screen area. This scan updates the complete screen display 60 times every second.

The VIC-II chip keeps track of which line is being scanned, and stores the scan number in the Raster Register at 53266 and 53265 ($D012 and $D011). The Raster Register has two functions. When read, it tells what line is presently being scanned. But when written to, it designates a particular scan line as the place where a raster interrupt will occur.

At the exact moment that the raster beam line number equals the number written to the register, Bit 0 of the status register will be set to 1, showing that the conditions for a Raster Compare Interrupt have been fulfilled. If the raster interrupt is enabled then, simultaneously, the interrupt program will be executed. This allows the user to reset any of the VIC-II registers at any point in the display, and thus change character sets, background color, or graphics mode for only a part of the screen display.

The interrupt routine will first check if the desired condition is the source of the interrupt (see the above entry) and then make the changes to the screen display. Once you have written this interrupt routine, you must take the following steps to install it.

1. Set the interrupt disable flag in the status register with an SEI instruction. This will disable all interrupts and prevent the system from crashing while you are changing the interrupt vectors.

2. Enable the raster interrupt. This is done by setting Bit 0 of the VIC-II chip interrupt enable register at location 53274 ($D01A) to 1.

3. Indicate the scan line on which you want the interrupt to occur by writing to the raster registers. Don't forget that this is a nine-bit value, and you must set both the low byte (in location 53266, $D012) and the high bit (in the register at 53265, $D011) in order to insure that the interrupt will start at the scan line you want it to, and not 256 lines earlier or later.

4. Let the computer know where the machine language routine that you want the interrupt to execute starts. This is done by placing the address in the interrupt vector at locations 788-789 ($314-$315). This address is split into two parts, a low byte and a high byte, with the low byte stored at 788.

To calculate the two values for a given address AD, you may use the formula HIBYTE=INT(AD/256) and LOWBYTE=AD-(HIBYTE*256). The value LOWBYTE would go into location 788, and the value HIBYTE would go into location 789.

5. Reenable interrupts with a CLI instruction, which clears the interrupt disable flag on the status register.

When the computer is first turned on, the interrupt vector is set to point to the normal hardware timer interrupt routine, the one that advances the jiffy clock and reads the keyboard. Since this interrupt routine uses the same vector as the raster interrupt routine, it is best to turn off the hardware timer interrupt by putting a value of 127 in location 56333 ($DC0D).

If you want the keyboard and jiffy clock to function normally while your interrupt is enabled, you must preserve the contents of locations 788 and 789 before you change them to point to your new routine. Then you must have your interrupt routine jump to the old interrupt routine exactly once per screen refresh (every 1/60 second).

Another thing that you should keep in mind is that at least two raster interrupts are required if you want to change only a part of the screen. Not only must the interrupt routine change the display, but it must also set up another raster interrupt that will change it back.

The sample program below uses a raster-scan interrupt to divide the display into three sections. The first 80 scan lines are in high-resolution bitmap mode, the next 40 are regular text, and the last 80 are in multicolor bitmap mode. The screen will split this way as soon as a SYS to the routine that turns on the interrupt occurs. The display will stay split even after the program ends. Only if you hit the STOP and RESTORE keys together will the display return to normal.

The interrupt uses a table of values that are POKEd into four key locations during each of the three interrupts, as well as values to determine at what scan lines the interrupts will occur. The locations affected are Control Register 1 (53265, $D011), Control Register 2 (53270, $D016), the Memory Control Register (53272, $D018), and

Background Color 0 (53281, $D021). The data for the interrupt routine is contained in lines 49152-49276. Each of these line numbers corresponds to the locations where the first data byte in the statement is POKEd into memory.

If you look at lines 49264-49276 of the BASIC program, you will see REMark statements that explain which VIC-II registers are affected by the DATA statements in each line. The numbers in these DATA statements appear in the reverse order in which they are put into the VIC register. For example, line 49273 holds the data that will go into Control Register 2. The last number, 8, is the one that will be placed into Control Register 2 while the top part of the screen is displayed. The first number, 24, is placed into Control Register 2 during the bottom part of the screen display, and changes that portion of the display to multicolor mode.

The only tricky part in determining which data byte affects which interrupt comes in line 49264, which holds the data that determines the scan line at which each interrupt will occur. Each DATA statement entry reflects the scan line at which the *next* interrupt will occur. The first item in line 49264 is 49. Even though this is the entry for the third interrupt, this number corresponds to the top of the screen (only scan lines 50-249 are visible on the display). That is because after the third interrupt, the next to be generated is the first interrupt, which occurs at the top of the screen. Likewise, the last data item of 129 is used during the first interrupt to start the next interrupt at scan line 129.

Try experimenting with these values to see what results you come up with. For example, if you change the number 170 to 210, you will increase the text area by five lines (40 scan lines).

By changing the values in the data tables, you can alter the effect of each interrupt. Change the 20 in line 49276 to 22, and you will get lowercase text in the middle of the screen. Change the first 8 in line 49273 to 24, and you'll get multicolor text in the center window. Each of these table items may be used exactly like you would use the corresponding register, in order to change background color, to obtain text or bitmap graphics, regular or multicolor modes, screen blanking or extended background color mode.

It is even possible to change the table values during a program, by POKEing the new value into the memory location where those table values are stored. In that way, you can, for example, change the background color of any of the screen parts while the program is running.

```
5 FOR I=Ø TO 7:BI(I)=2↑I:NEXT
1Ø FOR I=49152 TO 49278: READ A:POKE I,A:NEXT:SYS1
   2*4Ø96
2Ø PRINT CHR$(147):FOR I=Ø TO 8:PRINT:NEXT
```

```
30 PRINT"THE TOP AREA IS HIGH-RES BIT MAP MODE"
40 PRINT:PRINT"THE MIDDLE AREA IS ORDINARY TEXT "
50 PRINT:PRINT"THE BOTTOM AREA IS MULTI-COLOR BIT
   {SPACE}MAP"
60 FORG=1384 TO 1423:POKE G,6:NEXT
70 FORG=1024 TO 1383:POKEG,114:POKE G+640,234:NEXT
80 A$="":FOR I=1 TO 128:A$=A$+"@":NEXT:FOR I=32 TO
    63 STEP 2
90 POKE 648,I:PRINT CHR$(19)CHR$(153);A$;A$;A$;A$:
   NEXT:POKE 648,4
100 BASE=2*4096:BK=49267
110 H=40:C=0:FORX=0TO319:GOSUB150:NEXT
120 H=160:C=0:FORX=0TO319STEP2:GOSUB150:NEXT:C=40
125 FORX=1TO319STEP2:GOSUB150:NEXT
130 C=80:FOR X=0 TO 319 STEP2:W=0:GOSUB150:W=1:GOS
    UB150:NEXT
140 GOTO 140
150 Y=INT(H+20*SIN(X/10+C)):BY=BASE+40*(Y AND 248)
    +(Y AND 7)+(X AND 504)
160 POKE BY,PEEK(BY) OR (BI(ABS(7-(XAND7)-W))):RET
    URN
49152 DATA 120, 169, 127, 141, 13, 220
49158 DATA 169, 1, 141, 26, 208, 169
49164 DATA 3, 133, 251, 173, 112, 192
49170 DATA 141, 18, 208, 169, 24, 141
49176 DATA 17, 208, 173, 20, 3, 141
49182 DATA 110, 192, 173, 21, 3, 141
49188 DATA 111, 192, 169, 50, 141, 20
49194 DATA 3, 169, 192, 141, 21, 3
49200 DATA 88, 96, 173, 25, 208, 141
49206 DATA 25, 208, 41, 1, 240, 43
49212 DATA 198, 251, 16, 4, 169, 2
49218 DATA 133, 251, 166, 251, 189, 115
49224 DATA 192, 141, 33, 208, 189, 118
49230 DATA 192, 141, 17, 208, 189, 121
49236 DATA 192, 141, 22, 208, 189, 124
49242 DATA 192, 141, 24, 208, 189, 112
49248 DATA 192, 141, 18, 208, 138, 240
49254 DATA 6, 104, 168, 104, 170, 104
49260 DATA 64, 76, 49, 234
49264 DATA 49, 170, 129 :REM SCAN LINES
49267 DATA 0, 6, 0:REM BACKGROUND COLOR
49270 DATA 59, 27,59:REM CONTROL REG. 1
49273 DATA 24, 8, 8:REM CONTROL REG. 2
49276 DATA 24, 20, 24:REM MEMORY CONTROLRUN
```

Besides enabling the creation of mixed graphics-modes screens, the Raster Compare Interrupt is also useful for creating scrolling zones, so that some parts of the screen can be fine-scrolled while the rest remains stationary.

Bit 1 enables the light pen interrupt. This interrupt can occur when the light of the raster beam strikes the light-sensitive device in the pen's tip, causing it to close the fire button switch on joystick Controller Port #1.

The light pen interrupt affords a method of signaling to a program that the pen is being held to the screen, and that its position can be read. Some light pens provide a push-button switch which grounds one of the other lines on the joystick port. This switch can be pressed by the user as an additional signal that the pen is properly positioned. Its location can then be read in the light pen position registers (53267-8, $D013-4).

Bit 2 enables the sprite-foreground collision interrupt. This interrupt can occur if one of the sprite character's dots is touching one of the dots from the foreground display of either text character or bitmap graphics.

Bit 3 enables the sprite-sprite collision interrupt, which can occur if one of the sprite character's dots is touching one of the dots of another sprite character.

These two interrupts are useful for games, where such collisions often require that some action be taken immediately. Once the interrupt signals that a collision has occurred, the interrupt routine can check the Sprite-Foreground Collision Register at 53279 ($D01F), or the Sprite-Sprite Collision Register at 53278 ($D01E), to see which sprite or sprites are involved in the collision. See the entry for those locations for more details on collisions.

53275 $D01B SPBGPR

Sprite to Foreground Display Priority Register

Bit 0: Select display priority of Sprite 0 to foreground (0=sprite appears in front of foreground)
Bit 1: Select display priority of Sprite 1 to foreground (0=sprite appears in front of foreground)
Bit 2: Select display priority of Sprite 2 to foreground (0=sprite appears in front of foreground)
Bit 3: Select display priority of Sprite 3 to foreground (0=sprite appears in front of foreground)
Bit 4: Select display priority of Sprite 4 to foreground (0=sprite appears in front of foreground)
Bit 5: Select display priority of Sprite 5 to foreground (0=sprite appears in front of foreground)
Bit 6: Select display priority of Sprite 6 to foreground (0=sprite appears in front of foreground)
Bit 7: Select display priority of Sprite 7 to foreground (0=sprite appears in front of foreground)

If a sprite is positioned to appear at a spot on the screen that is al-

ready occupied by text or bitmap graphics, a conflict arises. The contents of this register determines which one will be displayed in such a situation. If the bit that corresponds to a particular sprite is set to 0, the sprite will be displayed in front of the foreground graphics data. If that bit is set to 1, the foreground data will be displayed in front of the sprite. The default value that this register is set to at power-on is 0, so all sprites start out with priority over foreground graphics.

Note that for the purpose of priority, the 01 bit-pair of multicolor graphics modes is considered to display a background color, and therefore will be shown behind sprite graphics even if the foreground graphics data takes priority. Also, between the sprites themselves there is a fixed priority. Each sprite has priority over all higher-number sprites, so that Sprite 0 is displayed in front of all the others.

The use of priority can aid in creating three-dimensional effects, by allowing some objects on the screen to pass in front of or behind other objects.

53276 $D01C SPMC

Sprite Multicolor Registers

Bit 0: Select multicolor mode for Sprite 0 (1=multicolor, 0=hi-res)
Bit 1: Select multicolor mode for Sprite 1 (1=multicolor, 0=hi-res)
Bit 2: Select multicolor mode for Sprite 2 (1=multicolor, 0=hi-res)
Bit 3: Select multicolor mode for Sprite 3 (1=multicolor, 0=hi-res)
Bit 4: Select multicolor mode for Sprite 4 (1=multicolor, 0=hi-res)
Bit 5: Select multicolor mode for Sprite 5 (1=multicolor, 0=hi-res)
Bit 6: Select multicolor mode for Sprite 6 (1=multicolor, 0=hi-res)
Bit 7: Select multicolor mode for Sprite 7 (1=multicolor, 0=hi-res)

Sprite multicolor mode is very similar to text and bitmap multicolor modes (see Bit 4 of 53270, $D016). Normally, the color of each dot of the sprite display is controlled by a single bit of sprite shape data. When this mode is enabled for a sprite, by setting the corresponding bit of this register to 1, the bits of sprite shape data are grouped together in pairs, with each pair of bits controlling a double-wide dot of the sprite display. By sacrificing some of the horizontal resolution (the sprite, although the same size, is now only 12 dots wide), you gain the use of two additional colors. The four possible combinations of these bit-pairs display dot colors from the following sources:

00 Background Color Register 0 (transparent)
01 Sprite Multicolor Register 0 (53285, $D025)
10 Sprite Color Registers (53287-94, $D027-E)
11 Sprite Multicolor Register 1 (53286, $D026)

Like multicolor text characters, multicolor sprites all share two color

registers. While each sprite can display three foreground colors, only one of these colors is unique to that sprite. The number of unique colors may be increased by combining more than one sprite into a single character.

53277 $D01D XXPAND

Sprite Horizontal Expansion Register

Bit 0: Expand Sprite 0 horizontally (1=double-width sprite, 0=normal width)
Bit 1: Expand Sprite 1 horizontally (1=double-width sprite, 0=normal width)
Bit 2: Expand Sprite 2 horizontally (1=double-width sprite, 0=normal width)
Bit 3: Expand Sprite 3 horizontally (1=double-width sprite, 0=normal width)
Bit 4: Expand Sprite 4 horizontally (1=double-width sprite, 0=normal width)
Bit 5: Expand Sprite 5 horizontally (1=double-width sprite, 0=normal width)
Bit 6: Expand Sprite 6 horizontally (1=double-width sprite, 0=normal width)
Bit 7: Expand Sprite 7 horizontally (1=double-width sprite, 0=normal width)

This register can be used to double the width of any sprite. Setting any bit of this register to 1 will cause each dot of the corresponding sprite shape to be displayed twice as wide as normal, so that without changing its horizontal resolution, the sprite takes up twice as much space. The horizontal expansion feature can be used alone, or in combination with the vertical expansion register at 53271 ($D017).

Location Range: 53278-53279 ($D01E-$D01F)

Sprite Collision Detection Registers

While Bit 2 of the VIC IRQ Register at 53273 ($D019) is set to 1 anytime two sprites overlap, and Bit 1 is set to 1 when a sprite shape is touching the foreground text or bit-graphics display, these registers specify which sprites were involved in the collision. Every bit that is set to 1 indicates that the corresponding sprite was involved in the collision. Reading these registers clears them so that they can detect the next collision. Therefore, if you plan to make multiple tests on the values stored here, it may be necessary to copy it to a RAM variable for further reference.

Note that while these registers tell you what sprites were involved in a collision, they do not necessarily tell you what objects have collided with each other. It is quite possible to have three sprites lined up in a row, where Sprite A is on the left, Sprite B is in the middle, touching Sprite A, and Sprite C is on the right, touching Sprite B but not touching Sprite A. The Sprite-Sprite Collision register would show that all three are involved. The only way to make absolutely certain which collided with which is to check the position of each sprite, and calculate for each sprite display line if a sprite of that size would touch either of the others. As you can imagine, this is no easy task.

There are a few simple rules concerning what does or does not cause a collision. Though the sprite character consists of 504 dots in a 24 by 21 matrix, dots which represent data bits that are equal to 0 (or multicolor bit-pairs equal to 00), and therefore always displayed in the background color, do not count when it comes to collision.

A collision can occur only if a dot which represents a sprite shape data bit of 1 touches another dot of nonzero graphics data. Consider the case of two invisible sprites. The first sprite is enabled, its color is set to contrast the background, and it is positioned on the screen, but its shape data bytes are all 0. This sprite can never be involved in a collision, because it displays no nonzero data. The second sprite is enabled, positioned on the screen, and its shape pointer set for a data read that is filled with bytes having a value of 255. Even if that sprite's color is set to the same value as the background color, making the sprite invisible, it can still be involved in collisions. The only exception to this rule is the 01 bit-pair of multicolor graphics data. This bit-pair is considered part of the background, and the dot it displays can never be involved in a collision.

The other rule to remember about collisions is that they can occur in areas that are covered by the screen border. Collision between sprites can occur when the sprites are offscreen, and collisions between sprites and foreground display data can occur when that data is in an area that is coverred by the border due to the reduction of the display to 38 columns or 24 rows.

53278 $D01E SPSPCL

Sprite to Sprite Collision Register

Bit 0: Did Sprite 0 collide with another sprite? (1=yes)
Bit 1: Did Sprite 1 collide with another sprite? (1=yes)
Bit 2: Did Sprite 2 collide with another sprite? (1=yes)
Bit 3: Did Sprite 3 collide with another sprite? (1=yes)
Bit 4: Did Sprite 4 collide with another sprite? (1=yes)
Bit 5: Did Sprite 5 collide with another sprite? (1=yes)
Bit 6: Did Sprite 6 collide with another sprite? (1=yes)
Bit 7: Did Sprite 7 collide with another sprite? (1=yes)

53279 $D01F SPBGCL

Sprite to Foreground Collision Register

Bit 0: Did Sprite 0 collide with the foreground display? (1=yes)
Bit 1: Did Sprite 1 collide with the foreground display? (1=yes)
Bit 2: Did Sprite 2 collide with the foreground display? (1=yes)
Bit 3: Did Sprite 3 collide with the foreground display? (1=yes)
Bit 4: Did Sprite 4 collide with the foreground display? (1=yes)
Bit 5: Did Sprite 5 collide with the foreground display? (1=yes)
Bit 6: Did Sprite 6 collide with the foreground display? (1=yes)
Bit 7: Did Sprite 7 collide with the foreground display? (1=yes)

Location Range: 53280-53294 ($D020-$D02E)

VIC-II Color Register

Although these color registers are used for various purposes, all of them have one thing in common. Like the Color RAM Nybbles, only the lower four bits are connected. Therefore, when reading these registers, you must mask out the upper four bits (that is, BORDERCOLOR=PEEK(53280 AND 15)) in order to get a true reading.

53280 $D020 EXTCOL

Border Color Register

The color value here determines the color of the border or frame around the central display area. The entire screen is set to this color when the blanking feature of Bit 4 of 53265 ($D011) is enabled. The default color value is 14 (light blue).

53281 $D021 BGCOL0

Background Color 0

This register sets the background color for all text modes, sprite graphics, and multicolor bitmap graphics. The default color value is 6 (blue).

53282 $D022 BGCOL1

Background Color 1

This register sets the color for the 01 bit-pair of multicolor character graphics, and the background color for characters having screen codes 64-127 in extended background color text mode. The default color value is 1 (white).

53283 $D023 BGCOL2

Background Color 2

This register sets the color for the 10 bit-pair of multicolor character graphics, and the background color for characters having screen

codes 128-191 in extended background color text mode. The default color value is 2 (red).

53284 $D024 BGCOL3

Background Color 3

This register sets the background color for characters having screen codes between 192 and 255 in extended background color text mode. The default color value is 3 (cyan).

53285 $D025 SPMC0

Sprite Multicolor Register 0

This register sets the color that is displayed by the 01 bit-pair in multicolor sprite graphics. The default color value is 4 (purple).

53286 $D026 SPMC1

Sprite Multicolor Register 1

This register sets the color that is displayed by the 11 bit-pair in multicolor sprite graphics. The default color value is 0 (black).

Location Range: 53287-53294 ($D027-$D02E)

Sprite Color Registers

These registers are used to set the color to be displayed by bits of hi-res sprite data having a value of 1, and by bit-pairs of multicolor sprite data having a value of 10. The color of each sprite is determined by its own individual color register.

53287 $D027 SP0COL

Sprite 0 Color Register (the default color value is 1, white)

53288 $D028 SP1COL

Sprite 1 Color Register (the default color value is 2, red)

53289 $D029 SP2COL

Sprite 2 Color Register (the default color value is 3, cyan)

53290 $D02A SP3COL

Sprite 3 Color Register (the default color value is 4, purple)

53291 $D02B SP4COL

Sprite 4 Color Register (the default color value is 5, green)

53292 $D02C SP5COL

Sprite 5 Color Register (the default color value is 6, blue)

53293 $D02D SP6COL
Sprite 6 Color Register (the default color value is 7, yellow)

53294 $D02E SP7COL
Sprite 7 Color Register (the default color value is 12, medium gray)

Location Range: 53295-53311 ($D02F-$D03F)

Not Connected
The VIC-II chip has only 47 registers for 64 bytes of possible address space. Therefore, the remaining 17 addresses do not access any memory. When read, they will always give a value of 255 ($FF). This value will not change after writing to them.

Location Range: 53312-54271 ($D040-$D3FF)

VIC-II Register Images
Since the VIC-II requires only enough addressing lines to handle 64 locations (the minimum possible for its 47 registers), none of the higher bits are decoded when addressing this 1K area. The result is that every 64-byte area in this 1K block is a mirror of every other. POKE53281+64,1 has the same effect as POKE 53281,1 or POKE 53281+10*64,1; they all turn the screen background to white. For the sake of clarity in your programs it is advisable to use the base address of the chip.

Sound Interface Device (SID) Registers

Memory locations 54272-54300 ($D400-$D41C) are used to address the 6581 Sound Interface Device (SID).

SID is a custom music synthesizer and sound effects generator chip that gives the 64 its impressive musical capabilities. It provides three separate music channels, or voices, as they are called. Each voice has 16-bit frequency resolution, waveform control, envelope shaping, oscillator synchronization, and ring modulation. In addition, programmable high-pass, low-pass, and band-pass filters can be set and enabled or disabled for each sound channel.

Since quite a few of these locations must be used in concert to produce sound, a brief summary of the interplay between some of these registers may be helpful.

Often the first step is to select an overall volume level using the

Volume Register. Then, the desired frequency or pitch of the note is chosen by writing to each of the two bytes which make up the 16-bit Frequency Register.

An ADSR envelope setting must be chosen by writing values to the Attack/Decay and Sustain/Release Registers. These determine the rate of the rise and fall of the volume of the note from zero volume to peak volume and back again. These rates have a great influence on the character of the sound.

Finally, the waveform must be selected, and the note started (or the oscillator gated, as we say). This is done by writing certain bits to the Control Register. The waveform control lets you select one of four different waveforms, each of which has varying harmonic content that affects the tone quality of the sound. By writing a 1 to the gate bit, you start the Attack/Decay/Sustain cycle. After rising to a peak and declining to the Sustain volume, the volume will continue at the same level until you write a 0 to the gate bit. Then, the Release cycle will start. Make sure that you keep the same waveform bit set to 1 while you write the 0 to the gate bit, so that the Release cycle starts. Otherwise, the sound will stop entirely, as it also will if the Volume Register or the Frequency Register is set to 0.

It should be noted that except for the last four SID chip registers, these addresses are write-only. That means that their values cannot be determined by PEEKing these locations.

Location Range: 54272-54273 ($D400-$D401)

Voice 1 Frequency Control

Together, these two locations control the frequency or pitch of the musical output of voice 1. Some frequency must be selected in order for voice 1 to be heard. This frequency may be changed in the middle of a note to achieve special effects. The 16-bit range of the Frequency Control Register covers over eight full octaves, and allows you to vary the pitch from 0 (very low) to about 4000 Hz (very high), in 65536 steps. The exact frequency of the output can be determined by the equation

FREQUENCY=(REGISTER VALUE*CLOCK/16777216)Hz

where CLOCK equals the system clock frequency, 1022730 for American (NTSC) systems, 985250 for European (PAL), and REGISTER VALUE is the combined value of these frequency registers. That combined value equals the value of the low byte plus 256 times the value of the high byte. Using the American (NTSC) clock value, the euqation works out to

FREQUENCY=REGISTER VALUE*.060959458 Hz

54272 $D400 FRELO1
Voice 1 Frequency Control (low byte)

54273 $D401 FREHI1
Voice 1 Frequency Control (high byte)

Location Range: 54274-54275($D402-$D403)

Voice 1 Pulse Waveform Width Control

As you will see below under the description of the Control Register at 54276 ($D404), you can select one of four different waveforms for the output of each voice. If the pulse waveform is selected, these registers must be set to establish the pulse width.

The pulse width has a 12-bit resolution, being made up of the value in the first register and the value in the lower nybble of the second register. The pulse width determines the duty cycle, or proportion of time that the rectangular wave will stay at the high part of the cycle.

The following formula shows the relationship between the value in the Pulse Width Register and the proportion of time that the wave stays at the high part of the cycle:

PULSE WIDTH=(REGISTER VALUE/40.95)%

The possible range of register values (0-4095) covers the range of duty cycles from 0 to 100 percent in 4096 steps. Changing the pulse width will vastly change the sound created with the pulse waveform.

54274 $D402 PWLO1
Voice 1 Pulse Waveform Width (low byte)

54275 $D403 PWHI1
Voice 1 Pulse Waveform Width (high nybble)

54276 $D404 VCREG1
Voice 1 Control Register

Bit 0: Gate Bit: 1=Start attack/decay/sustain, 0=Start release
Bit 1: Sync Bit: 1=Synchronize Oscillator with Oscillator 3 frequency
Bit 2: Ring Modulation: 1=Ring modulate Oscillators 1 and 3
Bit 3: Test Bit: 1=Disable Oscillator 1
Bit 4: Select triangle waveform
Bit 5: Select sawtooth waveform
Bit 6: Select pulse waveform
Bit 7: Select random noise waveform

Bit 0. Bit 0 is used to gate the sound. Setting this bit to a 1 while selecting one of the four waveforms will start the attack/decay/sustain part of the cycle. Setting this bit back to a 0 (while keeping the same waveform setting) anytime after a note has started playing will begin the release cycle of the note. Of course, in order for the gate bit to have an effect, the frequency and attack/decay/sustain/release (ADSR) registers must be set, as well as the pulse width, if necessary, and the volume control set to a nonzero value.

Bit 1. This bit is used to synchronize the fundamental frequency of Oscillator 1 with the fundamental frequency of Oscillator 3, allowing you to create a wide range of complex harmonic structures from voice 1. Synchronization occurs when this bit is set to 1. Oscillator 3 must be set to some frequency other than zero, but no other voice 3 parameters will affect the output from voice 1.

Bit 2. When Bit 2 is set to 1, the triangle waveform output of voice 1 is replaced with a ring modulated combination of Oscillators 1 and 3. This ring modulation produces nonharmonic overtone structures that are useful for creating bell or gong effects.

Bit 3. Bit 3 is the test bit. When set to 1, it disables the output of the oscillator. This can be useful in generating very complex waveforms (even speech synthesis) under software control.

Bit 4. When set to 1, Bit 4 selects the triangle waveform output of Oscillator 1. Bit 0 must also be set for the note to be sounded.

Bit 5. This bit selects the sawtooth waveform when set to 1. Bit 0 must also be set for the sound to begin.

Bit 6. Bit 6 chooses the pulse waveform when set to 1. The harmonic content of sound produced using this waveform may be varied using the Pulse Width Registers. Bit 0 must be set to begin the sound.

Bit 7. When Bit 7 is set to 1, the noise output waveform for Oscillator 1 is set. This creates a random sound output whose waveform varies with a frequency proportionate to that of Oscillator 1. It can be used to imitate the sound of explosions, drums, and other unpitched noises.

One of the four waveforms must be chosen in order to create a sound. Setting more than one of these bits will result in a logical ANDing of the waveforms. Particularly, the combination of the noise waveform and another is not recommended.

Location Range: 54277-54278 ($D405-$D406)

Voice 1 Envelope (ADSR) Control

When a note is played on a musical instrument, the volume does not suddenly rise to a peak and then cut off to zero. Rather, the volume builds to a peak, levels off to an intermediate value, and then fades

away. This creates what is known as a volume *envelope.*

The first phase of the envelope, in which the volume builds to a peak, is known as the attack phase. The second, in which it declines to an intermediate level, is called the decay phase. The third, in which the intermediate level of volume is held, is known as the sustain period. The final interval, in which the sound fades away, is called the release part of the cycle.

The SID chip allows the volume envelope of each voice to be controlled, so that specific instruments may be imitated, or new sounds created. This is done via the attack/decay and sustain/release registers. Each register devotes four bits (which can store a number from 0 to 15) to each phase of the cycle. When a note is gated by writing a 1 to a waveform bit and to Bit 0 of the Control Register, the attack cycle begins.

The volume of the sound builds to a peak over the period of time specified by the high nybble of the attack/decay register. Once it has reached the peak volume, it falls to the intermediate level during the period indicated by the low nybble of the attack/decay register (this is the decay phase). The volume of this intermediate or sustain level is selected by placing a value in the high nybble of the sustain/release register. This volume level is held until a 0 is written to the gate bit of the control register (while leaving the waveform bit set). When that happens, the release phase begins, and the volume of the sound begins to taper off during the period indicated by the low nybble of the sustain/release register.

You may notice the volume of the sound does not quite get to 0 at the end of the release cycle, and you may need to turn off the sound to get rid of the residual noise. You can do this either by setting the waveform bit back to 0, changing the frequency to 0, or setting the volume to 0.

54277 $D405 ATDCY1

Voice 1 Attack/Decay Register

Bits 0-3: Select decay cycle duration (0-15)
Bits 4-7: Select attack cycle duration (0-15)

Bits 4-7 control the duration of the attack cycle. This is the period of time over which the volume will rise from 0 to its peak amplitude. There are 16 durations which may be selected. The way in which the number placed here corresponds to the elapsed time of this cycle is as follows:

0=2 milliseconds
1=8 milliseconds
2=16 milliseconds
3=24 milliseconds
4=38 milliseconds

5=56 milliseconds
6=68 milliseconds
7=80 milliseconds
8=100 milliseconds
9=250 milliseconds
10=500 milliseconds
11=800 milliseconds
12=1 second
13=3 seconds
14=5 seconds
15=8 seconds

Bits 0-3 control the length of the decay phase, in which the volume of the note declines from the peak reached in the attack phase to the sustain level. The number selected corresponds to the length of this phase as shown below:

0=6 milliseconds
1=24 milliseconds
2=48 milliseconds
3=72 milliseconds
4=114 milliseconds
5=168 milliseconds
6=204 milliseconds
7=240 milliseconds
8=300 milliseconds
9=750 milliseconds
10=1.5 seconds
11=2.4 seconds
12=3 seconds
13=9 seconds
14=15 seconds
15=24 seconds

Since the two functions share one register, you must multiply the attack value by 16 and add it to the decay value in order to come up with the number to be placed in the register:

REGISTER VALUE=(ATTACK*16)+DECAY

54278 $D406 SUREL1

Voice 1 Sustain/Release Control Register

Bits 0-3: Select release cycle duration (0-15)
Bits 4-7: Select sustain volume level (0-15)

Bits 4-7 select the volume level at which the note is sustained. Following the decay cycle, the volume of the output of voice 1 will remain at the selected sustain level as long as the gate bit of the Control Register is set to 1. The sustain values range from 0, which

chooses no volume, to 15, which sets the output of voice 1 equal to the peak volume achieved during the attack cycle.

Bits 0-3 determine the length of the release cycle. This phase, in which the volume fades from the sustain level to near zero volume, begins when the gate bit of the Control Register is set to 0 (while leaving the waveform setting that was previously chosen). The duration of this decline in volume corresponds to the number (0-15) selected in the same way as for the decay value:

0=6 milliseconds
1=24 milliseconds
2=48 milliseconds
3=72 milliseconds
4=114 milliseconds
5=168 milliseconds
6=204 milliseconds
7=240 milliseconds
8=300 milliseconds
9=750 milliseconds
10=1.5 seconds
11=2.4 seconds
12=3 seconds
13=9 seconds
14=15 seconds
15=24 seconds

Location Range: 54279-54292 ($D407-$D414)

Voice 2 and Voice 3 Controls

The various control registers for these two voices correspond almost exactly to those of voice 1. The one exception is that the sync and ring-modulation bits of voice 2 operate on Oscillators 1 and 2, while the same bits of the Control Register for voice 3 use Oscillators 2 and 3.

54279 $D407 FRELO2
Voice 2 Frequency Control (low byte)

54280 $D408 FREHI2
Voice 2 Frequency Control (high byte)

54281 $D409 PWLO2
Voice 2 Pulse Waveform Width (low byte)

54282 $D40A PWHI2
Voice 2 Pulse Waveform Width (high nybble)

54283 $D40B VCREG2

Voice 2 Control Register

Bit 0: Gate Bit: 1=Start attack/decay/sustain, 0=Start release
Bit 1: Sync Bit: 1=Synchronize oscillator with Oscillator 1 frequency
Bit 2: Ring Modulation: 1=Ring modulate Oscillators 2 and 1
Bit 3: Test Bit: 1=Disable Oscillator 2
Bit 4: Select triangle waveform
Bit 5: Select sawtooth waveform
Bit 6: Select pulse waveform
Bit 7: Select noise waveform

54284 $D40C ATDCY2

Voice 2 Attack/Decay Register

Bits 0-3: Select decay cycle duration (0-15)
Bits 4-7: Select attack cycle duration (0-15)

54285 $D40D SUREL2

Voice 2 Sustain/Release Control Register

Bits 0-3: Select release cycle duration (0-15)
Bits 4-7: Select sustain volume level (0-15)

54286 $D40E FRELO3

Voice 3 Frequency Control (low byte)

54287 $D40F FREHI3

Voice 3 Frequency Control (high byte)

54288 $D410 PWLO3

Voice 3 Pulse Waveform Width (low byte)

54289 $D411 PWHI3

Voice 3 Pulse Waveform Width (high nybble)

54290 $D412 VCREG3

Voice 3 Control Register

Bit 0: Gate Bit: 1=Start attack/decay/sustain, 0=Start release
Bit 1: Sync Bit: 1=Synchronize oscillator with Oscillator 2 frequency
Bit 2: Ring modulation: 1=Ring modulate Oscillators 3 and 2
Bit 3: Test Bit: 1=Disable Oscillator 3
Bit 4: Select triangle waveform
Bit 5: Select sawtooth waveform
Bit 6: Select pulse waveform
Bit 7: Select noise waveform

54291 $D413 ATDCY3
Voice 3 Attack/Decay Register

Bits 0-3: Select decay cycle duration (0-15)
Bits 4-7: Select attack cycle duration (0-15)

54292 $D414 SUREL3
Voice 3 Sustain/Release Control Register

Bits 0-3: Select release cycle duration (0-15)
Bits 4-7: Select sustain volume level (0-15)

Location Range: 54293-54296 ($D415-$D418)

Filter Controls

In addition to the controls detailed above for each voice, the SID chip also provides a filtering capability which allows you to attenuate (make quieter) certain ranges of frequencies. Any one or all three voices can be filtered, and there is even a provision for filtering an external signal that is input through pin 5 of the monitor jack.

A low-pass filter is available, which suppresses the volume of those frequency components that are above a designated cutoff level. The high-pass filter reduces the volume of frequency components that are below a certain level. The band-pass filter reduces the volume of frequency components on both sides of the chosen frequency, thereby enhancing that frequency. Finally, the high-pass and low-pass filters can be combined to form a notch reject filter, which reduces the volume of the frequency components nearest the selected frequency. These various filters can dramatically change the quality of the sound produced.

The first two registers are used to select the filter cutoff frequency. This is the frequency above or below which any sounds will be made quieter. The further away from this level any frequency components are, the more their output volume will be suppressed (high- and low-pass filters reduce the volume of those components by 12 dB per octave away from the center frequency, while the band-pass filter attenuates them by 6 dB per octave).

The cutoff frequency has an 11-bit range (which corresponds to numbers from 0 to 2047). This is made up of a high byte and three low bits. Therefore, to compute the frequency represented by the value in these registers, you must multiply the value in the high byte by 8, and add the value of the low three bits. The range of cutoff frequencies represented by these 2048 values stretches from 30 Hz to about 12,000 Hz. The exact frequency may be calculated with the formula:

FREQUENCY=(REGISTER VALUE*5.8)+30Hz

An additional element in filtering is the resonance control. This allows you to peak the volume of the frequency elements nearest the cutoff frequency.

54293 $D415 CUTLO

Bits 0-2: Low portion of filter cutoff frequency
Bits 5-7: Unused

54294 $D416 CUTHI

Filter Cutoff Frequency (high byte)

54295 $D417 RESON

Filter Resonance Control Register

Bit 0: Filter the output of voice 1? 1=yes
Bit 1: Filter the output of voice 2? 1=yes
Bit 2: Filter the output of voice 3? 1=yes
Bit 3: Filter the output from the external input? 1=yes
Bits 4-7: Select filter resonance 0-15

Bits 0-3 are used to control which of the voices will be altered by the filters. If one of these bits is set to 1, the corresponding voice will be processed through the filter, and its harmonic content will be changed accordingly. If the bit is set to 0, the voice will pass directly to the audio output. Note that there is also a provision for processing an external audio signal which is brought in through pin 5 of the Audio/Video Port.

Bits 4-7 control the resonance of the filter. By placing a number from 0 to 15 in these four bits, you may peak the volume of those frequencies nearest the cutoff. This creates an even sharper filtering effect. A setting of 0 causes no resonance, while a setting of 15 gives maximum resonance.

54296 $D418 SIGVOL

Volume and Filter Select Register

Bits 0-3: Select output volume (0-15)
Bit 4: Select low-pass filter, 1=low-pass on
Bit 5: Select band-pass filter, 1=band-pass on
Bit 6: Select high-pass filter, 1=high-pass on
Bit 7: Disconnect output of voice 3, 1=voice 3 off

Bits 0-3 control the volume of all outputs. The possible volume levels range from 0 (no volume) to 15 (maximum volume). Some level of volume must be set here before any sound can be heard.

Bits 4-6 control the selection of the low-pass, band-pass, or high-pass filter. A 1 in any of these bits turns the corresponding filter on. These filters can be combined, although only one cutoff

frequency can be chosen. In order for the filter to have any effect, at least one of the voices must be routed through it using the Filter Resonance Control Register at 54295 ($D417).

When Bit 7 is set to 1, it disconnects the output of voice 3. This allows you to use the output of the oscillator for modulating the frequency of the other voices, or for generating random numbers, without any undesired audio output.

Location Range: 54297-54298 ($D419-$D41A)

Game Paddle Inputs

These registers allow you to read the game paddles that plug into joystick Controller Ports 1 and 2. Each paddle uses a variable resistor (also known as a potentiometer or pot), whose resistance is controlled by turning a knob. The varying resistance is used to vary the voltage to two pins of the SID chip between 0 and +5 volts. Analog-to-digital (A/D) converters in the chip interpret these voltage levels as binary values and store the values in these registers. These registers return a number from 0 (minimum resistance) to 255 (maximum resistance) for each paddle in either of the ports, depending on the position of the paddle knob.

Since these registers will read the paddle values for only one controller port, there is a switching mechanism which allows you to select which of the two ports to read. By writing a bit-pair of 01 (bit value of 64) to the last two bits of CIA #1 Data Port A (56320, $DC00), you select the paddles on joystick Controller Port 1. By writing a bit-pair of 10 (bit value of 128), you select the paddles on Controller Port 2.

If you look at the description of Data Port A (56320, $DC00), however, you will notice that it is also used in the keyboard scanning process. By writing to this port, you determine which keyboard column will be read.

Since the IRQ interrupt keyboard scan routine and the routine that checks for the STOP key are putting values into this location 60 times per second, you cannot reliably select the pair of paddles you wish to read from BASIC without first turning off the keyboard IRQ. This can be done with a POKE 56333,127. You can then read the paddles with the statements A=PEEK(54297) and B=PEEK(54298). The IRQ can be restored after the paddle read with a POKE 56333,129. It may, however, be easier and more accurate in the long run to use a machine language paddle read subroutine such as that presented on page 347 of the *Commodore 64 Programmer's Reference Guide.*

The paddle fire buttons are read as Bits 2 and 3 of the Data Ports A (56320, $DC00) and B (56321, $DC01). On Port A, if Bit 2 is set to 0, button 1 is pushed, and if Bit 3 is set to 0, button 2 is

pushed. On Port B, if Bit 2 is set to 0, button 3 is pushed, and if Bit 3 is set to 0, button 4 is pushed.

The BASIC statements to test these buttons, therefore, are:

```
PB(1)=(PEEK(56321)AND4)/4
PB(2)=(PEEK(56321)AND8)/8
PB(3)=(PEEK(56320)AND4)/4
PB(4)=(PEEK(56320)AND8)/8
```

If a 0 is returned by the PEEK statement, the button is pushed, and if a 1 is returned, it is not.

54297 $D419 POTX

Read Game Paddle 1 (or 3) Position

54298 $D41A POTY

Read Game Paddle 2 (or 4) Position

54299 $D41B RANDOM

Read Oscillator 3/Random Number Generator

This register lets you read the upper eight bits of the waveform output of Oscillator 3. The kinds of numbers generated by this output depend on the type of waveform selected.

If the sawtooth waveform is chosen, the output read by this register will be a series of numbers which start at 0 and increase by 1 to a maximum of 255, at which time they start over at 0.

When the triangle waveform is chosen, they increase from 0 to 255, at which time they decrease to 0 again. The rate at which these numbers change is determined by the frequency of Oscillator 3.

If the pulse waveform is selected, the output here will be either 255 or 0.

Finally, selecting the noise waveform will produce a random series of numbers between 0 and 255. This allows you to use the register as a random number generator for games.

There are many other uses for reading Oscillator 3, however, particularly for modulation of the other voices through machine language software. For example, the output of this register could be added to the frequency of another voice. If the triangle waveform were selected for this purpose, it would cause the frequency of the other voice to rise and fall, at the frequency of Oscillator 3 (perhaps for vibrato effects). This output can also be combined with the Filter Frequency or Pulse Width Registers to vary the values in these registers quickly over a short period of time.

Normally, when using Oscillator 3 for modulation, the audio output of voice 3 should be turned off by setting Bit 7 of the Volume and Filter Select Register at 54296 ($D418) to 1. It is not necessary to gate Bit 0 of Control Register 3 to use the oscillator, however, as its output is not affected by the ADSR envelope cycle.

54300 $D41C ENV3

Envelope Generator 3 Output

This register allows you to read the output of the voice 3 Envelope generator, in much the same way that the preceding register lets you read the output of Oscillator 3. This output can also be added to another oscillator's Frequency Control Registers, Pulse Width Registers, or the Filter Frequency Register. In order to produce any output from this register, however, the gate bit in Control Register 3 must be set to 1. Just as in the production of sound, setting the gate bit to 1 starts the attack/decay/sustain cycle, and setting it back to 0 starts the release cycle.

Location Range: 54301-54303 ($D41D-$D41F)

Not Connected

The SID chip has been provided with enough addresses for 32 different registers, but as it has only 29, the remaining three addresses are not used. Reading them will always return a value of 255 ($FF), and writing to them will have no effect.

Location Range: 54304-55295 ($D420-$D7FF)

SID Register Images

Since the SID chip requires enough addressing lines for only 32 locations (the minimum possible for its 29 registers), none of the higher bits are decoded when addressing the 1K area that has been assigned to it. The result is that every 32-byte area in this 1K block is a mirror of every other. For the sake of clarity in your programs, it is advisable not to use these addresses at all.

55296-56319 ($D800-$DBFF) Color RAM

The normal Commodore 64 text graphics system uses a screen RAM area to keep track of the character shapes that are to be displayed. But since each character can be displayed in any of 16 foreground colors, there must also be a parallel area which keeps track of the foreground color. This 1024-byte area is used for that purpose (actually, since there are only 1000 screen positions, only 1000 bytes actually affect screen color).

These 1000 bytes each control the foreground color of one character, with the first byte controlling the foreground color of the character in the upper-left corner, and subsequent bytes controlling the characters to the right and below that character.

Because only four bits are needed to represent the 16 colors available, only the low four bits of each Color RAM location are

connected (this is why they are sometimes referred to as Color RAM Nybbles). Writing to the high bits will not affect them, and these four bits will usually return a random value when read (a small number of 64s return a constant value).

Therefore, in order to read Color RAM correctly, you must mask out the top bits by using the logical AND function. In BASIC, you can read the first byte of Color RAM with the statement CR=PEEK(55296)AND15. This will always return a color value between 0 and 15. These color values correspond to the following colors:

0	= BLACK
1	= WHITE
2	= RED
3	= CYAN (LIGHT BLUE-GREEN)
4	= PURPLE
5	= GREEN
6	= BLUE
7	= YELLOW
8	= ORANGE
9	= BROWN
10	= LIGHT RED
11	= DARK GRAY
12	= MEDIUM GRAY
13	= LIGHT GREEN
14	= LIGHT BLUE
15	= LIGHT GRAY

Color mapping affords a convenient method of changing the color of the text display without changing the letters. By POKEing the appropriate section of Color RAM, you can change the color of a whole section of text on the screen without affecting the content of the text. You can even use this method to make letters disappear by changing their foreground colors to match the background color (or by changing the background to match the foreground), and later make them reappear by changing them back, or by changing the background to a contrasting color. An interesting example program which changes Color RAM quickly in BASIC can be found under the entry for 648 ($288).

A change in the Operating System causes newer 64s to set all of the Color RAM locations to the same value as the current background color whenever the screen is cleared. Therefore, POKEing character codes to the Screen RAM area will not appear to have any effect, because the letters will be the same color as the background. This can easily be turned to your advantage, however, because it means that all you have to do to set all of Color RAM to a particular value is to set the background color to that value (using the register

at 53281, $D021), clear the screen, and return the background color to the desired value.

The various graphics modes use this area differently than does the regular text mode. In high-resolution bitmap mode, this area is not used at all, but in multicolor bitmap mode it is used to determine the color of the 11 bit-pair for a given 8 dot by 8 dot area.

In multicolor text mode, only the lowest three bits are used, so only colors 0-7 may be selected. The fourth bit is used to determine whether a character will be displayed in regular text or multicolor text. Characters with a color value over 7 are displayed as multicolor characters, with the color of the 11 bit-pair determined by the color value minus 8. Characters with a color value under 8 are displayed normally.

It should be noted that unlike the Screen RAM area, which can be moved to any RAM location, the Color RAM area is fixed, and will function normally regardless of where screen memory is located.

Complex Interface Adapter (CIA) #1 Registers

Locations 56320-56335 ($DC00–$DC0F) are used to communicate with the Complex Interface Adapter chip #1 (CIA #1). This chip is a successor to the earlier VIA and PIA devices used on the VIC-20 and PET. This chip functions the same way as the VIA and PIA: It allows the 6510 microprocessor to communicate with peripheral input and output devices. The specific devices that CIA #1 reads data from and sends data to are the joystick controllers, the paddle fire buttons, and the keyboard.

In addition to its two data ports, CIA #1 has two timers, each of which can count an interval from a millionth of a second to a fifteenth of a second. Or the timers can be hooked together to count much longer intervals. CIA #1 has an interrupt line which is connected to the 6510 IRQ line. These two timers can be used to generate interrupts at specified intervals (such as the 1/60 second interrupt used for keyboard scanning, or the more complexly timed interrupts that drive the tape read and write routines). As you will see below, the CIA chip has a host of other features to aid in Input/Output functions.

Location Range: 56320-56321 ($DC00-$DC01)

CIA #1 Data Ports A and B

These registers are where the actual communication with outside devices takes place. Bits of data written to these registers can be sent to external devices, while bits of data that those devices send can be read here.

The keyboard is so necessary to the computer's operation that you may have a hard time thinking of it as a peripheral device. Nonetheless, it cannot be directly read by the 6510 microprocessor. Instead, the keys are connected in a matrix of eight rows by eight columns to CIA #1 Ports A and B. The layout of this matrix is shown below.

READ PORT B (56321, $DC01)

WRITE TO PORT A 56320/$DC00	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Bit 7	STOP	Q	C=	SPACE	2	CTRL	←	1
Bit 6	/	↑	=	RIGHT SHIFT	HOME	;	*	£
Bit 5	,	@	:	.	-	L	P	+
Bit 4	N	O	K	M	0	J	I	9
Bit 3	V	U	H	B	8	G	Y	7
Bit 2	X	T	F	C	6	D	R	5
Bit 1	LEFT SHIFT	E	S	Z	4	A	W	3
Bit 0	CRSR DOWN	f5	f3	f1	f7	CRSR RIGHT	RETURN	DELETE

As you can see, there are two keys which do not appear in the matrix. The SHIFT LOCK key is not read as a separate key, but rather is a mechanical device which holds the left SHIFT key switch in a closed position. The RESTORE key is not read like the other keys either. It is directly connected to the NMI interrupt line of the 6510 microprocessor, and causes an NMI interrupt to occur whenever it is pressed (not just when it is pressed with the STOP key).

In order to read the individual keys in the matrix, you must first set Port A for all outputs (255, $FF), and Port B for all inputs (0), using the Data Direction Registers. Note that this is the default condition. Next, you must write a 0 in the bit of Data Port A that corresponds to the column that you wish to read, and a 1 to the bits that correspond to columns you wish to ignore. You will then be able to read Data Port B to see which keys in that column are being pushed.

A 0 in any bit position signifies that the key in the corresponding row of the selected column is being pressed, while a 1 indicates that the key is not being pressed. A value of 255 ($FF) means that no keys in that column are being pressed.

Fortunately for us all, an interrupt routine causes the keyboard to be read, and the results are made available to the Operating System automatically every 1/60 second. And even when the normal interrupt routine cannot be used, you can use the Kernal SCNKEY routine at 65439 ($FF9F) to read the keyboard.

These same data ports are also used to read the joystick controllers. Although common sense might lead you to believe that you could read the joystick that is plugged into the port marked Controller Port 1 from Data Port A, and the second joystick from Data Port B, there is nothing common about the Commodore 64. Controller Port 1 is read from Data Port B, and Controller Port 2 is read from CIA #1 Data Port A.

Joysticks consist of five switches, one each for the up, down, right, and left directions, and another for the fire button. The switches are read like the key switches—if the switch is pressed, the corresponding bit will read 0, and if it is not pressed, the bit will be set to 1. From BASIC, you can PEEK the ports and use the AND and NOT operators to mask the unused bits and invert the logic for easier comprehension. For example, to read the joystick in Controller Port 1, you could use the statement:

```
S1=NOT PEEK(56321)AND15
```

The meaning of the possible numbers returned are:

0=none pressed
1=up
2=down
4=left
5=up left
6=down left

8=right
9=up right
10=down right

The same technique can be used for joystick 2, by substituting 56320 as the number to PEEK. By the way, the 3 and 7 aren't listed because they represent impossible combinations like up-down.

To read the fire buttons, you can PEEK the appropriate port and use the AND operator to mask all but Bit 4:

```
T1=(PEEK(56321)AND16)/16
```

The above will return a 0 if the button is pressed, and a 1 if it is not. Substitute location 56320 as the location to PEEK for Trigger Button 2.

Since CIA #1 Data Port B is used for reading the keyboard as well as joystick 1, some confusion can result. The routine that checks the keyboard has no way of telling whether a particular bit was set to 0 by a keypress or one of the joystick switches. For example, if you plug the joystick into Controller Port 1 and push the stick to the right, the routine will interpret this as the 2 key being pressed, because both set the same bit to 0. Likewise, when you read the joystick, it will register as being pushed to the right if the 2 key is being pressed.

The problem of mistaking the keyboard for the joystick can be solved by turning off the keyscan momentarily when reading the stick with a POKE 56333,127:POKE 56320,255, and restoring it after the read with a POKE 56333,129. Sometimes you can use the simpler solution of clearing the keyboard buffer after reading the joystick, with a POKE 198,0.

The problem of mistaking the joystick for a keypress is much more difficult—there is no real way to turn off the joystick. Many commercially available games just use Controller Port 2 to avoid the conflict. So, if you can't beat them, sit back and press your joystick to the left in order to slow down a program listing (the keyscan routine thinks that it is the CTRL key).

As if all of the above were not enough, Port A is also used to control which set of paddles is read by the SID chip, and to read the paddle fire buttons. Since there are two paddles per joystick Controller Port, and only two SID registers for reading paddle positions, there has to be a method for switching the paddle read from joystick Port 1 to joystick Port 2.

When Bit 7 of Port A is set to 1 and Bit 6 is cleared to 0, the SID registers read the paddles on Port 1. When Bit 7 is set to 0 and Bit 6 is set to 1, the paddles on Port 2 are read by the SID chip registers. Note that this also conflicts with the keyscan routine, which is constantly writing different values to CIA #1 Data Port A in order to select the keyboard column to read (most of the time, the value for the last column is written to this port, which coincides with the

selection of paddles on joystick Port 1). Therefore, in order to get an accurate reading, you must turn off the keyscan IRQ and select which joystick port you want to read. See POTX at 54297 ($D419), which is the SID register where the paddles are read, for the exact technique.

Although the SID chip is used to read the paddle settings, the fire buttons are read at CIA #1 Data Ports A and B. The fire buttons for the paddles plugged into Controller Port 1 are read at Data Port B (56321, $DC01), while those for the paddles plugged into Controller Port 2 are read from Data Port A (56320, $DC00). The fire buttons are read at Bit 2 and Bit 3 of each port (the same as the joystick left and joystick right switches), and as usual, the bit will read 0 if the corresponding button is pushed, and 1 if it is not.

Although only two of the four paddle values can be read at any one time, you can always read all four paddle buttons. See the game paddle input description at 54297 ($D419) for the BASIC statements used to read these buttons.

Finally, Data Port B can also be used as an output by either Timer A or B. It is possible to set a mode in which the timers do not cause an interrupt when they run down (see the descriptions of Control Registers A and B at 56334-5 $DC0E-F). Instead, they cause the output on Bit 6 or 7 of Data Port B to change. Timer A can be set either to pulse the output of Bit 6 for one machine cycle, or to toggle that bit from 1 to 0 or 0 to 1. Timer B can use Bit 7 of this register for the same purpose.

56320 $DC00 CIAPRA

Data Port Register A

Bit 0: Select to read keyboard column 0
Read joystick 2 up direction
Bit 1: Select to read keyboard column 1
Read joystick 2 down direction
Bit 2: Select to read keyboard column 2
Read joystick 2 left direction
Read paddle 1 fire button
Bit 3: Select to read keyboard column 3
Read joystick 2 right direction
Read paddle 2 fire button
Bit 4: Select to read keyboard column 4
Read joystick 2 fire button
Bit 5: Select to read keyboard column 5
Bit 6: Select to read keyboard column 6
Select to read paddles on Port A or B
Bit 7: Select to read keyboard column 7
Select to read paddles on Port A or B

56321 $DC01 CIAPRB

Data Port Register B

Bit 0: Read keyboard row 0
Read joystick 1 up direction
Bit 1: Read keyboard row 1
Read joystick 1 down direction
Bit 2: Read keyboard row 2
Read joystick 1 left direction
Read paddle 1 fire button
Bit 3: Read keyboard row 3
Read joystick 1 right direction
Read paddle 2 fire button
Bit 4: Read keyboard row 4
Read joystick 1 fire button
Bit 5: Read keyboard row 5
Bit 6: Read keyboard row 6
Toggle or pulse data output for Timer A
Bit 7: Select to read keyboard column 7
Toggle or pulse data output for Timer B

Location Range: 56322-56323 ($DC02-$DC03)

CIA #1 Data Direction Registers A and B

These Data Direction Registers control the direction of data flow over Data Ports A and B. Each bit controls the direction of the data on the corresponding bit of the port. If the bit of the Direction Register is set to a 1, the corresponding Data Port bit will be used for data output. If the bit is set to a 0, the corresponding Data Port bit will be used for data input. For example, Bit 7 of Data Direction Register A controls Bit 7 of Data Port A, and if that direction bit is set to 0, Bit 7 of Data Port A will be used for data input. If the direction bit is set to 1, however, data Bit 7 on Port A will be used for data output.

The default setting for Data Direction Register A is 255 (all outputs), and for Data Direction Register B it is 0 (all inputs). This corresponds to the setting used when reading the keyboard (the keyboard column number is written to Data Port A, and the row number is then read in Data Port B).

56322 $DC02 CIDDRA

Data Direction Register A

Bit 0: Select Bit 0 of Data Port A for input or output (0=input, 1=output)
Bit 1: Select Bit 1 of Data Port A for input or output (0=input, 1=output)

Bit 2: Select Bit 2 of Data Port A for input or output (0=input, 1=output)
Bit 3: Select Bit 3 of Data Port A for input or output (0=input, 1=output)
Bit 4: Select Bit 4 of Data Port A for input or output (0=input, 1=output)
Bit 5: Select Bit 5 of Data Port A for input or output (0=input, 1=output)
Bit 6: Select Bit 6 of Data Port A for input or output (0=input, 1=output)
Bit 7: Select Bit 7 of Data Port A for input or output (0=input, 1=output)

56323 $DC03 CIDDRB

Data Direction Register B

Bit 0: Select Bit 0 of Data Port B for input or output (0=input, 1=output)
Bit 1: Select Bit 1 of Data Port B for input or output (0=input, 1=output)
Bit 2: Select Bit 2 of Data Port B for input or output (0=input, 1=output)
Bit 3: Select Bit 3 of Data Port B for input or output (0=input, 1=output)
Bit 4: Select Bit 4 of Data Port B for input or output (0=input, 1=output)
Bit 5: Select Bit 5 of Data Port B for input or output (0=input, 1=output)
Bit 6: Select Bit 6 of Data Port B for input or output (0=input, 1=output)
Bit 7: Select Bit 7 of Data Port B for input or output (0=input, 1=output)

Location Range: 56324-56327 ($DC04-$DC07)

Timers A and B Low and High Bytes

These four timer registers (two for each timer) have different functions depending on whether you are reading from them or writing to them. When you read from these registers, you get the present value of the Timer Counter (which counts down from its initial value to 0). When you write data to these registers, it is stored in the Timer Latch, and from there it can be used to load the Timer Counter using the Force Load bit of Control Register A or B (see 56334-5, $DC0E-F below).

These interval timers can hold a 16-bit number from 0 to 65535, in normal 6510 low-byte, high-byte format (VALUE=LOW BYTE+256*HIGH BYTE). Once the Timer Counter is set to an initial

value, and the timer is started, the timer will count down one number every microprocessor clock cycle. Since the clock speed of the 64 (using the American NTSC television standard) is 1,022,730 cycles per second, every count takes approximately a millionth of a second. The formula for calculating the amount of time it will take for the timer to count down from its latch value to 0 is:

TIME=LATCH VALUE/CLOCK SPEED

where LATCH VALUE is the value written to the low and high timer registers (LATCH VALUE=TIMER LOW+256*TIMER HIGH), and CLOCK SPEED is 1,022,730 cycles per second for American (NTSC) standard television monitors, or 985,250 for European (PAL) monitors.

When Timer Counter A or B gets to 0, it will set Bit 0 or 1 in the Interrupt Control Register at 56333 ($DC0D). If the timer interrupt has been enabled (see 56333, $DC0D), an IRQ will take place, and the high bit of the Interrupt Control Register will be set to 1. Alternately, if the Port B output bit is set, the timer will write data to Bit 6 or 7 of Port B. After the timer gets to 0, it will reload the Timer Latch Value, and either stop or count down again, depending on whether it is in one-shot or continuous mode (determined by Bit 3 of the Control Register).

Although usually a timer will be used to count the microprocessor clock cycles, Timer A can count either the microprocessor clock cycles or external pulses on the CTN line, which is connected to pin 4 of the User Port.

Timer B is even more versatile. In addition to these two sources, Timer B can count the number of times that Timer A goes to 0. By setting Timer A to count the microprocessor clock, and setting Timer B to count the number of times that Timer A zeros, you effectively link the two timers into one 32-bit timer that can count up to 70 minutes with accuracy within 1/15 second.

In the 64, CIA #1 Timer A is used to generate the interrupt which drives the routine for reading the keyboard and updating the software clock. Both Timers A and B are also used for the timing of the routines that read and write tape data. Normally, Timer A is set for continuous operation, and latched with a value of 149 in the low byte and 66 in the high byte, for a total Latch Value of 17045. This means that it is set to count to 0 every 17045/1022730 seconds, or approximately 1/60 second.

For tape reads and writes, the tape routines take over the IRQ vectors. Even though the tape write routines use the on-chip I/O port at location 1 for the actual data output to the cassette, reading and writing to the cassette uses both CIA #1 Timer A and Timer B for timing the I/O routines.

56324 Timer A (low byte)	**$DC04**	**TIMALO**
56325 Timer A (high byte)	**$DC05**	**TIMAHI**
56326 Timer B (low byte)	**$DC06**	**TIMBLO**
56327 Timer B (high byte)	**$DC07**	**TIMBHI**

Location Range: 56328-56331 ($DC08-$DC0B)

Time of Day Clock (TOD)

In addition to the two general-purpose timers, the 6526 CIA chip has a special-purpose Time of Day Clock, which keeps time in a format that humans can understand a little more easily than microseconds.

This Time of Day Clock even has an alarm, which can cause an interrupt at a specific time. It is organized in four registers, one each for hours, minutes, seconds, and tenths of seconds. Each register reads out in Binary Coded Decimal (BCD) format, for easier conversion to ASCII digits. A BCD byte is divided into two nybbles, each of which represents a single digit in base 10. Even though a four-bit nybble can hold a number from 0 to 15, only the base 10 digits of 0-9 are used. Therefore, 10 o'clock would be represented by a byte in the hours register with the nybbles 0001 and 0000, which stand for the digits 1 and 0. The binary value of this byte would be 16 (16 times the high nybble plus the low nybble). Each of the other registers operates in the same manner. In addition, Bit 7 of the hours register is used as an AM/PM flag. If that bit is set to 1, it indicates PM, and if it is set to 0, the time is AM.

The Time of Day Clock Registers can be used for two purposes, depending on whether you are reading them or writing to them. If you are reading them, you will always be reading the time. There is a latching feature associated with reading the hours register in order to solve the problem of the time changing while you are reading the registers. For example, if you were reading the hours register just as the time was changing from 10:59 to 11:00, it is possible that you would read the 10 in the hours register, and by the time you read the minutes register it would have changed from 59 to 00. Therefore, you would read 10:00 instead of either 10:59 or 11:00.

To prevent this kind of mistake, the Time of Day Clock Registers stop updating as soon as you read the hours register, and do not start again until you read the tenths of seconds register. Of course,

the clock continues to keep time internally even though it does not update the registers. If you want to read only minutes, or seconds or tenths of seconds, there is no problem, and no latching will occur. But anytime you read hours, you must follow it by reading tenths of seconds, even if you don't care about them, or else the registers will not continue to update.

Writing to these registers either sets the time or the alarm, depending on the setting of Bit 7 of Control Register B (56335, $DC0F). If that bit is set to 1, writing to the Time of Day registers sets the alarm. If the bit is set to 0, writing to the Time of Day registers sets the Time of Day clock. In either case, as with reading the registers, there is a latch function. This function stops the clock from updating when you write to the hours register. The clock will not start again until you write to the tenths of seconds registers.

The only apparent use of the Time of Day Clock by the 64's Operating System is in the BASIC RND statement. There, the seconds and tenths of seconds registers are read and their values used as part of the seed value for the RND(0) command.

Nonetheless, this clock can be an invaluable resource for the 64 user. It will keep time more accurately than the software clock maintained at locations 160-162 ($A0-$A2) by the Timer A interrupt routine. And unlike that software clock, the Time of Day Clock will not be disturbed when I/O operations disrupt the Timer A IRQ, or when the IRQ vector is diverted elsewhere. Not even a cold start RESET will disrupt the time. For game timers, just set the time for 00:00:00:0 and it will keep track of elapsed time in hours, minutes, seconds and tenths of seconds format.

The following digital clock program, written in BASIC, will demonstrate the use of these timers:

```
10 PRINT CHR$(147):GOSUB 200
20 H=PEEK(56331):POKE 1238,(H AND 16)/16+48:POKE 1
   239,(H AND 15)+ 48
30 M=PEEK(56330):POKE 1241,(M AND 240)/16+48:POKE
   {SPACE}1242,(M AND 15)+ 48
40 S=PEEK(56329):POKE 1244,(S AND 240)/16+48:POKE
   {SPACE}1245,(S AND 15)+ 48
50 T=PEEK(56328)AND15:POKE 1247,T+48:GOTO 20
200 INPUT"WHAT IS THE HOUR";H$:IF H$=""THEN 200
210 H=0:IF LEN(H$)>1 THEN H=16
220 HH=VAL(RIGHT$(H$,1)):H=H+HH:POKE56331,H
230 INPUT "WHAT IS THE MINUTE";M$:IF M$=""THEN 200
240 M=0:IF LEN(M$)>1 THEN M=16*VAL(LEFT$(M$,1))
250 MM=VAL(RIGHT$(M$,1)):M=M+MM:POKE56330,M
260 INPUT "WHAT IS THE SECOND";S$:IF S$=""THEN 200
270 S=0:IF LEN(S$)>1 THEN S=16*VAL(LEFT$(S$,1))
280 SS=VAL(RIGHT$(S$,1)):S=S+SS:POKE56329,S:POKE 5
    6328,0
```

```
290 POKE 53281,1:PRINTCHR$(147):POKE 53281,6
300 POKE 1240,58:POKE 1243,58:POKE 1246,58:GOTO 20
```

56328 $DC08 TODTEN

Time of Day Clock Tenths of Seconds

Bits 0-3: Time of Day tenths of second digit (BCD)
Bits 4-7: Unused

56329 $DC09 TODSEC

Time of Day Clock Seconds

Bits 0-3: Second digit of Time of Day seconds (BCD)
Bits 4-6: First digit of Time of Day seconds (BCD)
Bit 7: Unused

56330 $DC0A TODMIN

Time of Day Clock Minutes

Bits 0-3: Second digit of Time of Day minutes (BCD)
Bits 4-6: First digit of Time of Day minutes (BCD)
Bit 7: Unused

56331 $DC0B TODHRS

Time of Day Clock Hours

Bits 0-3: Second digit of Time of Day hours (BCD)
Bit 4: First digit of Time of Day hours (BCD)
Bits 5-6: Unused
Bit 7: AM/PM Flag (1=PM, 0=AM)

56332 $DC0C CIASDR

Serial Data Port

The CIA chip has an on-chip serial port, which allows you to send or receive a byte of data one bit at a time, with the most significant bit (Bit 7) being transferred first. Control Register A at 56334 ($DC0E) allows you to choose input or output modes. In input mode, a bit of data is read from the SP line (pin 5 of the User Port) whenever a signal on the CNT line (pin 4) appears to let you know that it is time for a read. After eight bits are received this way, the data is placed in the Serial Port Register, and an interrupt is generated to let you know that the register should be read.

In output mode, you write data to the Serial Port Register, and it is sent out over the SP line (pin 5 of the User Port), using Timer A for the baud rate generator. Whenever a byte of data is written to this register, transmission will start as long as Timer A is running and in continuous mode. Data is sent at half the Timer A rate, and an output will appear on the CNT line (pin 4 of the User Port)

whenever a bit is sent. After all eight bits have been sent, an interrupt is generated to indicate that it is time to load the next byte to send into the Serial Register.

The Serial Data Register is not used by the 64, which does all of its serial I/O through the regular data ports.

56333 $DC0D CIAICR

Interrupt Control Register

Bit 0: Read/ did Timer A count down to 0? (1=yes)
Write/enable or disable Timer A interrupt (1=enable, 0=disable)

Bit 1: Read/ did Timer B count down to 0? (1=yes)
Write/enable or disable Timer B interrupt (1=enable, 0=disable)

Bit 2: Read/ did Time of Day Clock reach the alarm time? (1=yes)
Write/enable or disable TOD clock alarm interrupt (1=enable, 0=disable)

Bit 3: Read/ did the serial shift register finish a byte? (1=yes)
Write/enable or disable serial shift register interrupt (1=enable, 0=disable)

Bit 4: Read/ was a signal sent on the flag line? (1=yes)
Write/enable or disable FLAG line interrupt (1=enable, 0=disable)

Bit 5: Not used

Bit 6: Not used

Bit 7: Read/ did any CIA #1 source cause an interrupt? (1=yes)
Write/set or clear bits of this register (1=bits written with 1 will be set, 0=bits written with 1 will be cleared)

This register is used to control the five interrupt sources on the 6526 CIA chip. These sources are Timer A, Timer B, the Time of Day Clock, the Serial Register, and the FLAG line. Timers A and B cause an interrupt when they count down to 0. The Time of Day Clock generates an interrupt when it reaches the ALARM time. The Serial Shift Register interrupts when it compiles eight bits of input or output. An external signal pulling the CIA hardware line called FLAG low will also cause an interrupt (on CIA #1, this FLAG line is connected to the Cassette Read line of the Cassette Port).

Even if the condition for a particular interrupt is satisfied, the interrupt must still be enabled for an IRQ actually to occur. This is done by writing to the Interrupt Control Register. What happens when you write to this register depends on the way that you set Bit 7. If it is set to 0, any other bit that is written to with a 1 will be cleared, and the corresponding interrupt will be disabled. If you set Bit 7 to 1, any bit written to with a 1 will be set, and the corresponding interrupt will be enabled. In either case, the interrupt enable flags for those bits written to with a 0 will not be affected.

For example, in order to disable all interrupts from BASIC, you could POKE 56333,127. This sets Bit 7 to 0, which clears all of the other bits, since they are all written with 1's. Don't try this from BASIC immediate mode, as it will turn off Timer A which causes the IRQ for reading the keyboard, so that it will in effect turn off the keyboard.

To turn on the Timer A interrupt, a program could POKE 56333,129. Bit 7 is set to 1 and so is Bit 0, so the interrupt which corresponds to Bit 0 (Timer A) is enabled.

When you read this register, you can tell if any of the conditions for a CIA interrupt were satisfied because the corresponding bit will be set to a 1. For example, if Timer A counts down to 0, Bit 0 of this register will be set to 1. If, in addition, the mask bit that corresponds to that interrupt source is set to 1, and an interrupt occurs, Bit 7 will also be set. This allows a multi-interrupt system to read one bit and see if the source of a particular interrupt was CIA #1. You should note, however, that reading this register clears it, so you should preserve its contents in RAM if you want to test more than one bit.

56334 $DC0E CIACRA

Control Register A

Bit 0: Start Timer A (1=start, 0=stop)
Bit 1: Select Timer A output on Port B (1=Timer A output appears on Bit 6 of Port B)
Bit 2: Port B output mode (1=toggle Bit 6, 0=pulse Bit 6 for one cycle)
Bit 3: Timer A run mode (1=one-shot, 0=continuous)
Bit 4: Force latched value to be loaded to Timer A counter (1=force load strobe)
Bit 5: Timer A input mode (1=count microprocessor cycles, 0=count signals on CNT line at pin 4 of User Port)
Bit 6: Serial Port (56332, $DC0C) mode (1=output, 0=input)
Bit 7: Time of Day Clock frequency (1=50 Hz required on TOD pin, 0=60 Hz)

Bits 0-3. This nybble controls Timer A. Bit 0 is set to 1 to start the timer counting down, and set to 0 to stop it. Bit 3 sets the timer for one-shot or continuous mode.

In one-shot mode, the timer counts down to 0, sets the counter value back to the latch value, and then sets Bit 0 back to 0 to stop the timer. In continuous mode, it reloads the latch value and starts all over again.

Bits 1 and 2 allow you to send a signal on Bit 6 of Data Port B when the timer counts. Setting Bit 1 to 1 forces this output (which overrides the Data Direction Register B Bit 6, and the normal Data Port B value). Bit 2 allows you to choose the form this output to Bit 6 of Data Port B will take. Setting Bit 2 to a value of 1 will cause Bit

6 to toggle to the opposite value when the timer runs down (a value of 1 will change to 0, and a value of 0 will change to 1). Setting Bit 2 to a value of 0 will cause a single pulse of a one machine-cycle duration (about a millionth of a second) to occur.

Bit 4. This bit is used to load the Timer A counter with the value that was previously written to the Timer Low and High Byte Registers. Writing a 1 to this bit will force the load (although there is no data stored here, and the bit has no significance on a read).

Bit 5. Bit 5 is used to control just what it is Timer A is counting. If this bit is set to 1, it counts the microprocessor machine cycles (which occur at the rate of 1,022,730 cycles per second). If the bit is set to 0, the timer counts pulses on the CNT line, which is connected to pin 4 of the User Port. This allows you to use the CIA as a frequency counter or an event counter, or to measure pulse width or delay times of external signals.

Bit 6. Whether the Serial Port Register is currently inputting or outputting data (see the entry for that register at 56332, $DC0C for more information) is controlled by this bit.

Bit 7. This bit allows you to select from software whether the Time of Day Clock will use a 50 Hz or 60 Hz signal on the TOD pin in order to keep accurate time (the 64 uses a 60 Hz signal on that pin).

56335 $DC0F CIACRB

Control Register B

Bit 0: Start Timer B (1=start, 0=stop)
Bit 1: Select Timer B output on Port B (1=Timer B output appears on Bit 7 of Port B)
Bit 2: Port B output mode (1=toggle Bit 7, 0=pulse Bit 7 for one cycle)
Bit 3: Timer B run mode (1=one-shot, 0=continuous)
Bit 4: Force latched value to be loaded to Timer B counter (1=force load strobe)
Bits 5-6: Timer B input mode
- 00 = Timer B counts microprocessor cycles
- 01 = Count signals on CNT line at pin 4 of User Port
- 10 = Count each time that Timer A counts down to 0
- 11 = Count Timer A 0's when CNT pulses are also present

Bit 7: Select Time of Day write (0=writing to TOD registers sets alarm, 1=writing to TOD registers sets clock)

Bits 0-3. This nybble performs the same functions for Timer B that Bits 0-3 of Control Register A perform for Timer A, except that Timer B output on Data Port B appears at Bit 7, and not Bit 6.

Bits 5 and 6. These two bits are used to select what Timer B counts. If both bits are set to 0, Timer B counts the microprocessor

machine cycles (which occur at the rate of 1,022,730 cycles per second). If Bit 6 is set to 0 and Bit 5 is set to 1, Timer B counts pulses on the CNT line, which is connected to pin 4 of the User Port. If Bit 6 is set to 1 and Bit 5 is set to 0, Timer B counts Timer A underflow pulses, which is to say that it counts the number of times that Timer A counts down to 0. This is used to link the two timers into one 32-bit timer that can count up to 70 minutes with accuracy to within 1/15 second. Finally, if both bits are set to 1, Timer B counts the number of times that Timer A counts down to 0 *and* there is a signal on the CNT line (pin 4 of the User Port).

Bit 7. Bit 7 controls what happens when you write to the Time of Day registers. If this bit is set to 1, writing to the TOD registers sets the ALARM time. If this bit is cleared to 0, writing to the TOD registers sets the TOD clock.

Location Range: 56336-56575 ($DC10-$DCFF)

CIA #1 Register Images

Since the CIA chip requires only enough addressing lines to handle 16 registers, none of the higher bits are decoded when addressing the 256-byte area that has been assigned to it. The result is that every 16-byte area in this 256-byte block is a mirror of every other. Even so, for the sake of clarity in your programs it is advisable to use the base address of the chip, and not use higher addresses to communicate with the chip.

Complex Interface Adapter (CIA) #2 Registers

Locations 56576-56591 ($DD00-$DD0F) are used to address the Complex Interface Adapter chip #2 (CIA #2). Since the chip itself is identical to CIA #1, which is addressed at 56320 ($DC00), the discussion here will be limited to the use which the 64 makes of this particular chip. For more general information on the chip registers, please see the corresponding entries for CIA #1.

One of the significant differences between CIA chips #1 and #2 is the use to which Data Ports A and B are put. The peripheral input and output devices that CIA #2 controls are those on the Serial Bus (such as the 1541 Disk Drive and 1525 printer), the RS-232 device (which is used for telecommunications), and the User Port, an eight-bit parallel port that can be turned to whatever purpose the user desires. In addition, Data Port A has the important task of selecting the 16K bank of memory that will be used by the VIC-II chip for graphics.

Another significant difference between CIA chips #1 and #2 is that the interrupt line of CIA #1 is wired to the 6510 IRQ line, while that of CIA #2 is wired to the NMI line. This means that interrupts from this chip cannot be masked by setting the Interrupt disable flag (SEI). They can be disabled from CIA's Mask Register, though. Be sure to use the NMI vector when setting up routines to be driven by interrupts generated by this chip.

Location Range: 56576-56577 ($DD00-$DD01)

CIA #2 Data Ports A and B

These registers are where the communication with the Serial Bus, RS-232 device, and User Port take place. The Serial Bus is like the IEEE bus which is used by the PET, in that it allows more than one device to be connected to the port at a time, in a daisychain arrangement. Since each byte of data is sent one bit at a time, however, the Serial Bus is at least eight times slower than the IEEE. It is presently used to control the 1541 Disk Drive and 1525 printer, and other devices (such as printer interfaces for Centronics-type parallel printers and stringy floppy wafer tape storage units) can be placed on this bus.

Data Port A is used for communication with the Serial Bus. Bits 5 and 7 are used for Serial Bus Data Output and Input respectively, and Bits 4 and 6 are used for the Serial Bus Clock Pulse Output and Input. Bit 3 of Data Port A is used to send the ATN signal on the Serial Bus.

The 64 has built-in software to handle RS-232 communications through a modem or other device plugged in the RS-232/User Port. The RS-232 device uses Bit 2 of Data Port A for data output (it is the only line from Port A that is connected to the RS-232/User Port jack). It also makes heavy use of Port B, using Bit 7 for the Data Set Ready (DSR) signal, Bit 6 for the Clear to Send (CTS), Bit 4 for the Carrier Detect (DCD), Bit 3 for the Ring Indicator (RI), Bit 2 for Data Terminal Ready (DTR), Bit 1 for Request to Send (RTS), and Bit 0 for data input. See locations 659-660 ($293-$294) for more details on the RS-232 device.

All of the data lines which the RS-232 device uses are also available to the user as part of the User Port. All of the Port B data lines, and Bit 2 of Port A, are brought out to the User Port connector on the back of the 64. These data bits are utilized in the normal way: The port connections are made to TTL-level input or output devices, and the direction of data is determined by the Data Direction registers.

In addition, the User Port has pins connected to the two CIA Serial Ports (whose eight-bit shift registers are well-suited for serial-

to-parallel and parallel-to-serial conversion), and the two CNT lines which aid in the operation of the Serial Ports. The CNT lines can also be used in conjunction with the CIA Timers, and allow them to be used as frequency counters, event counters, interval timers, etc. The advanced features of the CIA chip make almost any type of interfacing application possible, and in the near future we will probably see many interesting applications for the User Port on the 64. A pin description of the User Port connector is provided below:

User Port Pin	CIA Line	RS-232 DB-25 Pin	Description
1			Ground
2			+5 Volts (100 milliamps maximum)
3			RESET (grounding this pin causes a cold start)
4	CNT1		CIA #1 Serial Port and Timer Counter
5	SP1		CIA #1 Serial Data Port
6	CNT2		CIA #2 Serial Port and Timer Counter
7	SP2		CIA #2 Serial Data Port
8	PC2		CIA #2 handshaking line
9			Connected to the ATN line of the Serial Bus
10			9 Volts AC (+phase, 50 milliamps maximum)
11			9 Volts AC (-phase, 50 milliamps maximum)
12			Ground
A		1	Ground
B	FLAG2		CIA#2 handshaking line
C	PB0	3	Port B Bit 0—RS-232 Received Data (SIN)
D	PB1	4	Port B Bit 1—RS-232 Request to Send (RTS)
E	PB2	20	Port B Bit 2—RS-232 Data Terminal Ready (DTR)
F	PB3	22	Port B Bit 3—RS-232 Ring Indicator (RI)
H	PB4	8	Port B Bit 4—RS-232 Carrier Detect (DCD)
J	PB5		Port B Bit 5
K	PB6	5	Port B Bit 6—RS-232 Clear to Send (CTS)
L	PB7	6	Port B Bit 7—RS-232 Data Set Ready (DSR)
M	PA2	2	Port A Bit 2—RS-232 Transmitted Data (Sout)
N		7	Ground

One of the handshaking lines on the above chart, PC2, was not covered in the discussion of CIA #1, because that line of CIA #1 is not connected to anything. The CIA #2 PC line is accessible from the User Port, however. This line will go low for one cycle following a read or write of Port B on CIA #2. This signal lets external devices know when data has been read or written.

Bits 0 and 1 of CIA #2 Port A have an extremely important function. As mentioned in the section on the VIC-II chip (53248, $D000), the video chip can address only 16K of memory at a time, and all graphics data must be stored in that 16K block in order to be displayed. Within this area, sprite graphics data may be placed in any of 256 groups of 64 bytes each. Character data can be stored in any of eight 2K blocks. Text screen memory may be in any of 16 1K areas, and bitmap screen memory may be in either of two 8K sections.

When you turn the power on, the VIC-II uses the bottom 16K of memory for graphics. Unfortunately, this block of memory is also used extensively for other important purposes. Though some means of eliminating these conflicts are discussed above, in many situations you will want to change from the default 16K bank at the low end of memory.

Bits 0 and 1 select the current 16K bank for video memory from the four possible choices using the following bit patterns:

00 (bit value of 0) Bank 3 (49152-65535, $C000-$FFFF)
01 (bit value of 1) Bank 2 (32768-49151, $8000-$BFFF)
10 (bit value of 2) Bank 1 (16384-32767, $4000-$7FFF)
11 (bit value of 3) Bank 0 (0-16383, $0-$3FFF)

The technique for making this change from BASIC is discussed below. But before we go ahead and start changing banks, let's briefly review the contents of these areas, and the considerations for using them for graphics.

Block 0. This is normally used for system variables and BASIC program text. Locations 1024-2048 ($400-$800) are reserved for the default position of screen memory.

There is an additional limitation on memory usage of this block, as the VIC-II sees the character generator ROM at 4096-8191 ($1000-$1FFF), making this portion of memory unavailable for other graphics data. Generally, there is little free space here for graphics display data. Locations 679-767 ($2A7-$2FF) are unused, and could hold one sprite shape (number 11) or data for 11 characters. The area from 820-1023 ($334-$3FF), which includes the cassette I/O buffer, is available for graphics memory, and is large enough to hold three sprite shapes (numbers 13, 14, and 15), or data for 25 characters (numbers 103-127). But getting enough memory for bitmap graphics requires that you either reserve memory after the end of BASIC text

by lowering the end of BASIC pointer at 56 ($38), or raise the start of BASIC pointer at 44 ($2C). See the entries for these pointers for more details.

Block 1. Block 1 is normally used for BASIC program storage. When using this bank, the VIC-II chip does not have access to the character generator ROM. Providing that you lower the top of memory so that BASIC programs do not interfere, this area is wide open for sprite shapes, character graphics, and bitmap graphics.

The drawbacks to using this bank are the unavailability of the character ROM and the limitation on BASIC program space (as little as 14K). The absence of the character ROM is a relatively minor nuisance, because you can always switch in the ROM and copy any or all of the characters to RAM (see the entries for location 1 and the alternate entry for 53248, $D000, the Character ROM, for details). This block may be a good alternate choice to avoid potential conflicts with other applications that use higher memory.

Block 2. The third block (Block 2) consists of 8K of RAM, half of which is seen by the VIC-II chip as character ROM, and the 8K BASIC interpreter ROM. The BASIC ROM area is available for graphics. This is possible because of the 64's special addressing. The VIC-II chip reads only from RAM, and thus sees the RAM underneath the BASIC ROM even if the 6510 has ROM switched in. The 6510, on the other hand, always writes to RAM, even when dealing with memory it reads as ROM. Whatever is written to the RAM underlying the BASIC ROM is displayed normally by the VIC-II chip. This opens up an extra 8K area for sprites and character data under the BASIC ROM.

You should keep in mind that while you can write to this area, you cannot read it from BASIC. This may not be a serious problem when it comes to character sets and sprite data, but it's more of a drawback if you want to use this RAM for screen memory.

For example, the Operating System has to read the text screen to move the cursor properly, and if it reads the ROM value instead of the RAM screen data, it gets hopelessly confused, making it impossible to type in any commands.

Likewise, you would not be able to read the high-resolution screen if it were placed here, without some machine language trickery. With locations 36864-40959 ousted by the character ROM, only 4K of true RAM remains for use as screen memory, not enough for a complete high-resolution screen. Therefore, this block is not recommended for use in bitmap mode if your program needs to check the screen. Otherwise, this is a good place for graphics memory, particularly if you need to emulate the screen configuration of the PET.

Block 3. Normally Block 3 contains 4K of RAM that is completely unused by the system, 4K of I/O registers, and the 8K Operating System Kernal ROM. It is very convenient to use when you need a

lot of memory space for both graphics and a BASIC program. Although the character ROM is not available, it can be copied to RAM. The area under the Kernal ROM can be used as explained above. One possible conflict that you should be aware of is that the current version of the DOS support program is written to reside at 52224 ($CC00). It would be safest to avoid using 52224-53247 for graphics if you plan to use DOS support.

Changing banks. Once you have selected a bank of 16K to use, the procedure for making the change from BASIC is as follows:

1. Set the Data Direction Register if necessary. In order to use Bits 0 and 1 of Port A to change banks, these bits must be set as outputs in Data Direction Register A. Since this is the default condition on powering-up, this step normally will not be needed.
2. Select a bank. Banks 0-3 can be chosen by entering the following lines:

```
POKE 56578, PEEK(56578) OR 3: REM SET FOR OUTPUT IF NOT
    ALREADY

POKE 56576,(PEEK(56576) AND 252) OR (3-BANK): REM BANK IS
    BANK #, MUST BE 0-3
```

3. Set the VIC-II register for character memory. As explained at the entry for location 53272 ($D018), the formula for this is:

```
POKE 53272, (PEEK(53272) AND 240) OR TK: REM TK IS 2 KBYTE
    OFFSET FROM BEGINNING OF BLOCK
```

4. Set the VIC-II register for display memory. As explained at the entry for location 53272 ($D018), the formula for this is:

```
POKE 53272, (PEEK(53272) AND 15) OR K*16: REM K IS KBYTE
    OFFSET FROM BEGINNING OF BLOCK
```

Since steps 2 and 3 operate on the same register, you could combine these steps and just POKE 53272, (16*K+TK).

4. Set the Operating System pointer for display memory at 648 ($288). Even though you have just told the VIC-II chip where to display memory for the screen, the Operating System does not yet know where to write its text characters. Let it know with this statement:

```
POKE 648,AD/256: REM AD IS THE ACTUAL ADDRESS OF
    SCREEN MEMORY
```

After you make this change, you must watch out for the STOP/RESTORE key combination. The BRK initialization changes the screen display default to location 1024 in Bank 0, but not the Operating System pointer at 648 ($288). As a result, what you are typing will not be displayed on the screen. The computer will lock up until you turn the power off and back on again. The simplest way to avoid this problem is to disable the RESTORE key entirely (see the entries

for 792 ($318) and 808 ($328) for more information).

Below is a sample program which switches the screen to Bank 3. It includes a machine language transfer routine to move the ROM character set to RAM, and a short interrupt routine to correct the RESTORE key problem. After the switch is made, a loop is used to POKE characters to the new screen memory area. Next, the character data is slowly erased, to show that the character set is now in RAM. Then, a loop is used to read the locations of the character set, and write to the same locations. This demonstrates that the 6510 reads the Kernal ROM when you PEEK those locations, but POKEs to the RAM which is being displayed. Finally, the machine language move is used again to show how quickly the set is restored.

```
20 FOR I=1 TO 33:READ A:POKE 49151+I,A:NEXT: REM S
   ET UP ML ROUTINE
30 GOSUB 200: REM ML COPY OF ROM CHARACTER SET TO
   {SPACE}RAM
40 POKE 56576,PEEK(56576) AND 252: REM{2 SPACES}ST
   EP 1, ENABLE BANK 3
50 POKE 53272,44: REM STEPS 2-3, POINT{2 SPACES}VI
   C-II TO SCREEN AND CHARACTER MEMORY
60 REM SCREEN OFFSET IS 2*16, CHARACTER{2 SPACES}O
   FFSET IS 1212
70 POKE 648,200: REM STEP 4, POINT OS TO SCREEN AT
    51200 (200*256)
80 PRINT CHR$(147): REM CLEAR SCREEN
90 FOR I=53236 TO 53245: READ A: POKE I,A: NEXT: R
   EM NEW INTERRUPT ROUTINE
100 POKE 53246,PEEK(792):POKE 53247,PEEK(793): REM
     SAVE OLD NMI VECTOR
110 POKE 792,244: POKE 793,207: REM ROUTE THE INTE
    RRUPT THROUGH THE NEW ROUTINE
120 FOR I=0 TO 255: POKE 51400+I,I:POKE 55496+I,1:
    NEXT
125 REM POKE CHARACTERS TO SCREEN
130 FOR J=1 TO 8: FOR I=61439+J TO I+2048 STEP 8
140 POKE I,0:NEXT I,J: REM ERASE CHARACTER SET
150 FOR I=61440 TO I+2048:POKE I,PEEK(I):NEXT: REM
     POKE ROM TO RAM
160 GOSUB 200:END: REM RESTORE CHARACTER SET
200 POKE 56334,PEEK(56334) AND 254: REM DISABLE IN
    TERRUPTS
210 POKE 1,PEEK(1) AND 251: REM SWITCH CHARACTER R
    OM INTO 6510 MEMORY
220 SYS 49152: REM COPY ROM CHARACTER SET TO RAM A
    T 61440
230 POKE 1,PEEK(1) OR 4: REM SWITCH CHARACTER ROM
    {SPACE}OUT OF 6510 MEMORY
240 POKE 56334,PEEK(56334) OR 1: REM ENABLE INTERR
    UPTS
```

```
25Ø RETURN
3ØØ REM DATA FOR ML PROGRAM TO COPY CHARACTER SET
    {SPACE}TO RAM
31Ø DATA169,Ø,133,251,133,253,169,2Ø8,133,252,169,
    24Ø,133,254,162,16
32Ø DATA16Ø,Ø,177,251,145,253,136,2Ø8,249,23Ø,252,
    23Ø,254,2Ø2,2Ø8,24Ø,96
33Ø REM NEXT IS ML PROGRAM TO MAKE THE RESTORE KEY
     RESET OS POINTER TO SCREEN
34Ø DATA 72,169,4,141,136,Ø2,1Ø4,1Ø8,254,2Ø7
```

See also the sample program showing how to configure your 64 like a PET at location 43 ($2B).

56576 $DD00 CI2PRA

Data Port Register A

Bits 0-1: Select the 16K VIC-II chip memory bank (11=bank 0, 00=bank 3)
Bit 2: RS-232 data output (Sout)/Pin M of User Port
Bit 3: Serial bus ATN signal output
Bit 4: Serial bus clock pulse output
Bit 5: Serial bus data output
Bit 6: Serial bus clock pulse input
Bit 7: Serial bus data input

56577 $DD01 CI2PRB

Data Port Register B

Bit 0: RS-232 data input (SIN)/ Pin C of User Port
Bit 1: RS-232 request to send (RTS)/ Pin D of User Port
Bit 2: RS-232 data terminal ready (DTR)/ Pin E of User Port
Bit 3: RS-232 ring indicator (RI)/ Pin F of User Port
Bit 4: RS-232 carrier detect (DCD)/ Pin H of User Port
Bit 5: Pin J of User Port
Bit 6: RS-232 clear to send (CTS)/ Pin K of User Port
Toggle or pulse data output for Timer A
Bit 7: RS-232 data set ready (DSR)/ Pin L of User Port
Toggle or pulse data output for Timer B

Location Range: 56578-56579 ($DD02-$DD03)

CIA #2 Data Direction Registers A and B

These Data Direction registers control the direction of data flow over Data Ports A and B. For more details on the operation of these registers, see the entry for the CIA #1 Data Direction Registers at 56322 ($DC02).

The default setting for Data Direction Register A is 63 (all bits except 6 and 7 are outputs), and for Data Direction Register B the default setting is 0 (all inputs). Bits 1 and 2 of Port B are changed to output when the RS-232 device is opened.

56578 $DD02 C2DDRA

Data Direction Register A

Bit 0: Select Bit 0 of data Port A for input or output (0=input, 1=output)
Bit 1: Select Bit 1 of data Port A for input or output (0=input, 1=output)
Bit 2: Select Bit 2 of data Port A for input or output (0=input, 1=output)
Bit 3: Select Bit 3 of data Port A for input or output (0=input, 1=output)
Bit 4: Select Bit 4 of data Port A for input or output (0=input, 1=output)
Bit 5: Select Bit 5 of data Port A for input or output (0=input, 1=output)
Bit 6: Select Bit 6 of data Port A for input or output (0=input, 1=output)
Bit 7: Select Bit 7 of data Port A for input or output (0=input, 1=output)

56579 $DD03 C2DDRB

Data Direction Register B

Bit 0: Select Bit 0 of data Port B for input or output (0=input, 1=output)
Bit 1: Select Bit 1 of data Port B for input or output (0=input, 1=output)
Bit 2: Select Bit 2 of data Port B for input or output (0=input, 1=output)
Bit 3: Select Bit 3 of data Port B for input or output (0=input, 1=output)
Bit 4: Select Bit 4 of data Port B for input or output (0=input, 1=output)
Bit 5: Select Bit 5 of data Port B for input or output (0=input, 1=output)
Bit 6: Select Bit 6 of data Port B for input or output (0=input, 1=output)
Bit 7: Select Bit 7 of data Port B for input or output (0=input, 1=output)

Location Range: 56580-56583 ($DD04-$DD07)

Timer A and B Low and High Bytes

These four timer registers are used to control Timers A and B. For details on the operation of these timers, see the entry for Location Range 56324-56327 ($DC04-$DC07).

The 64 Operating System uses the CIA #2 Timers A and B mostly for timing RS-232 send and receive operations. Serial Bus timing uses CIA #1 Timer B.

56580 $DD04 TI2ALO

Timer A (low byte)

56581 $DD05 TI2AHI

Timer A (high byte)

56582 $DD06 TI2BLO

Timer B (low byte)

56583 $DD07 TI2BHI

Timer B (high byte)

Location Range: 56584-56587 ($DD08-$DD0B)

Time of Day Clock

In addition to the two general-purpose timers, the 6526 CIA chip has a special-purpose Time of Day Clock, which keeps time in a format that humans can understand a little more easily than microseconds. For more information about this clock, see the entry for Location Range 56328-56331 ($DC08-$DC0B). The 64's Operating System does not make use of these registers.

56584 $DD08 TO2TEN

Time of Day Clock Tenths of Seconds

Bits 0-3: Time of Day tenths of second digit (BCD)
Bits 4-7: Unused

56585 $DD09 TO2SEC

Time of Day Clock Seconds

Bits 0-3: Second digit of Time of Day seconds (BCD)
Bits 4-6: First digit of Time of Day seconds (BCD)
Bit 7: Unused

56586 $DD0A TO2MIN

Time of Day Clock Minutes

Bits 0-3: Second digit of Time of Day minutes (BCD)
Bits 4-6: First digit of Time of Day minutes (BCD)
Bit 7: Unused

56587 $DD0B TO2HRS

Time of Day Clock Hours

Bits 0-3: Second digit of Time of Day hours (BCD)
Bit 4: First digit of Time of Day hours (BCD)
Bits 5-6: Unused
Bit 7: AM/PM flag (1=PM, 0=AM)

56588 $DD0C CI2SDR

Serial Data Port

The CIA chip has an on-chip serial port, which allows you to send or receive a byte of data one bit at a time, with the most significant bit (Bit 7) being transferred first. For more information about its use, see the entry for location 56332 ($DC0C). The 64's Operating System does not use this facility.

56589 $DD0D CI2ICR

Interrupt Control Register

Bit 0: Read/ did Timer A count down to 0? (1=yes)
Write/enable or disable Timer A interrupt (1=enable, 0=disable)
Bit 1: Read/ did Timer B count down to 0? (1=yes)
Write/enable or disable Timer B interrupt (1=enable, 0=disable)
Bit 2: Read/ did Time of Day Clock reach the alarm time? (1=yes)
Write/enable or disable TOD clock alarm interrupt (1=enable, 0=disable)
Bit 3: Read/ did the serial shift register finish a byte? (1=yes)
Write/enable or disable serial shift register interrupt (1=enable, 0=disable)
Bit 4: Read/ was a signal sent on the FLAG line? (1=yes)
Write/enable or disable FLAG line interrupt (1=enable, 0=disable)
Bit 5: Not used
Bit 6: Not used
Bit 7: Read/ did any CIA #2 source cause an interrupt? (1=yes)
Write/set or clear bits of this register (1=bits written with 1 will be set, 0=bits written with 1 will be cleared)

This register is used to control the five interrupt sources on the 6526

CIA chip. For details on its operation, see the entry for 56333 ($DC0D). The main difference between these two chips pertaining to this register is that on CIA #2, the FLAG line is connected to Pin B of the User Port, and thus is available to the user who wishes to take advantage of its ability to cause interrupts for handshaking purposes.

Location Range: 56590-56591 ($DD0E-$DD0F

See locations 56334 and 56335 for details

56590 $DD0E CI2CRA

Control Register A

Bit 0: Start Timer A (1=start, 0=stop)
Bit 1: Select Timer A output on Port B (1=Timer A output appears on Bit 6 of Port B)
Bit 2: Port B output mode (1=toggle Bit 6, 0=pulse Bit 6 for one cycle)
Bit 3: Timer A run mode (1=one-shot, 0=continuous)
Bit 4: Force latched value to be loaded to Timer A counter (1=force load strobe)
Bit 5: Timer A input mode (1=count microprocessor cycles, 0=count signals on CNT line at pin 4 of User Port)
Bit 6: Serial port (56588, $DD0C) mode (1=output, 0=input)
Bit 7: Time of Day Clock frequency (1=50 Hz required on TOD pin, 0=60 Hz)

56591 $DD0F CI2CRB

Control Register B

Bit 0: Start Timer B (1=start, 0=stop)
Bit 1: Select Timer B output on Port B (1=Timer B output appears on Bit 7 of Port B)
Bit 2: Port B output mode (1=toggle Bit 7, 0=pulse Bit 7 for one cycle)
Bit 3: Timer B run mode (1=one-shot, 0=continuous)
Bit 4: Force latched value to be loaded to Timer B counter (1=force load strobe)
Bits 5-6: Timer B input mode
  00 = Timer B counts microprocessor cycles
  01 = Count signals on CNT line at pin 4 of User Port
  10 = Count each time that Timer A counts down to 0
  11 = Count Timer A 0's when CNT pulses are also present
Bit 7: Select Time of Day write (0=writing to TOD registers sets alarm, 1=writing to TOD registers sets clock

Location Range: 56592-56831 ($DD10-$DDFF)

CIA #2 Register Images

Since the CIA chip requires only enough addressing lines to handle 16 registers, none of the higher bits are decoded when addressing the 256-byte area that has been assigned to it. The result is that every 16-byte area in this 256-byte block is a mirror of every other. For the sake of clarity in your programs, it is advisable not to use these addresses.

Location Range: 56832-57087 ($DE00-$DEFF)

Reserved for I/O Expansion

This range of locations is not used directly by the 64's internal hardware. It is, however, accessible via pin 7 of the Expansion Port. It can be used to control cartridges which are connected to this port. For example, the CP/M module uses this space to control which microprocessor is in control of the system. The Z-80 microprocessor is turned on and off by writing to 56832 ($DE00).

Another cartridge which uses this space is Simon's BASIC. This 16K cartridge is addressed at memory locations 32768-49151 ($8000-$BFFF), which means that it overlaps the regular BASIC ROM at 40960-49151 ($A000-$BFFF). But since it contains additions to BASIC, it must use the BASIC ROM as well. This problem is solved by copying the cartridge at 32768-40959 ($8000-$9FFF) to RAM, and turning the cartridge on and off by writing to or reading from location 56832 ($DE00).

Location Range: 57088-57343 ($DF00-$DFFF)

CIA #2 Register Images

This range of locations is not used directly by the 64's internal hardware, but is accessible via pin 10 of the Expansion Port. One possible use for this I/O memory that Commodore has mentioned is an inexpensive parallel disk drive (which presumably would be much faster than the current serial model).

Alternate 53248-57343 ($D000-$DFFF): Character Generator ROM

The character generator ROM supplies the data which is used to form the shapes of the text and graphics characters that are dis-

played on screen. Each character requires eight bytes of shape data, and these eight-byte sequences are arranged in the order in which the characters appear in the screen code chart (see Appendix G). For example, the first eight bytes of data in the ROM hold the shape information for the commercial at sign (@), the next eight hold the shape of the letter A, etc. In all, there are 4096 bytes, representing shape data for two complete character sets of 256 characters each—1K each for uppercase/graphics, reverse uppercase/reverse graphics, lowercase/uppercase, and reverse lowercase/reverse uppercase.

The shape of each character is formed by an 8 by 8 matrix of screen dots. Whether any of the 64 dots is lit up or not is determined by the bit patterns of the character data. Each byte of the Character ROM holds a number from 0 to 255. This number can be represented by eight binary digits of 0 or 1. The leftmost bit of these eight is known as Bit 7, while the rightmost bit is called Bit 0. Each of these binary digits has a bit value that is two times greater than the last. The values of a bit set to 1 in each of the bit places are:

Bit	0	1	2	3	4	5	6	7
Value	1	2	4	8	16	32	64	128

A byte whose value is 255 has every bit set to 1 (128+64+32+16+8+4+2+1=255), while a byte whose value is 0 is made up of all zero bits. Numbers in between are made up of combinations of bits set to 1 and bits set to 0. If you think of every bit that holds a 0 as a dot on the screen which is the color of the screen background, and every bit that holds a 1 as a dot whose color is that of the appropriate nybble in Color RAM, you can begin to get an idea of how the byte values relate to the shape of the character. For example, if you PEEK at the first eight bytes of the character ROM (the technique is explained in the entry for location 1), you will see the numbers 60, 102, 110, 110, 96, 98, 60, 0. Breaking these data bytes down into their bit values gives us a picture that looks like the following:

```
00111100    0 +  0+ 32+ 16+ 8+ 4+ 0+ 0    =  60
01100110    0 + 64+ 32+  0+ 0+ 4+ 2+ 0    = 102
01101110    0 + 64+ 32+  0+ 8+ 4+ 2+ 0    = 110
01101110    0 + 64+ 32+  0+ 8+ 4+ 2+ 0    = 110
01100000    0 + 64+ 32+  0+ 0+ 0+ 0+ 0    =  96
01100010    0 + 64+ 32+  0+ 0+ 0+ 2+ 0    =  98
00111100    0 +  0+ 32+ 16+ 8+ 4+ 0+ 0    =  60
00000000    0 +  0+  0+  0+ 0+ 0+ 0+ 0    =   0
```

If you look closely, you will recognize the shape of the commercial at sign (@) as it's displayed on your screen. The first byte of data is 60, and you can see that Bits 5, 4, 3, and 2 are set to 1. The chart above shows that the bit values for these bits are 32, 16, 8, and 4.

Adding these together, you get 32+16+8+4=60. This should give you an idea of how the byte value corresponds to the patterns of lit dots. For an even more graphic display, type in the following program, which shows the shape of any of the 512 characters in the ROM, along with the number value of each byte of the shape.

```
10 DIM B%(7),T%(7): FOR I=0 TO 7: T%(I)=2↑I: NEXT
20 POKE 53281,2: PRINT CHR$(147): POKE 53281,1: PO
   KE 53280,0: POKE 646,11
30 POKE 214,20: PRINT: INPUT " CHARACTER NUMBER (0
   -511)";C$
40 C=VAL(C$): GOSUB 80: FOR I=0 TO 7
50 POKE 214,6+I:PRINT:PRINT TAB(23);B%(I);CHR$(20)
   ;"{3 SPACES}"
60 FOR J=7 TO 0 STEP-1: POKE 1319+(7-J)+I*40,32-12
   8*((B%(I)ANDT%(J))=T%(J))
70 NEXT J,I: POKE 214,20: PRINT: PRINT TAB(27)"
   {4 SPACES}": GOTO 30
80 POKE 56333,127: POKE 1,51:FOR I=0 TO 7
90 B%(I)=PEEK(53248+C*8+I): NEXT: POKE 1,55: POKE
   {SPACE}56333,129: RETURN
```

If you have read about the VIC-II video chip, you know that it can address only 16K of memory at a time, and that all display data such as screen memory, character shape data, and sprite shape data must be stored within that 16K block.

Since it would be very inconvenient for the VIC-II chip to be able to access the character data only at the 16K block which includes addresses 53248-57343 ($D000–$DFFF), the 64 uses an addressing trick that makes the VIC-II chip *see* an image of the Character ROM at 4096-8191 ($1000-$1FFF) and at 36864-40959 ($9000-$9FFF). It is not available in the other two blocks. To generate characters in these blocks, you must supply your own user-defined character set, or copy the ROM data to RAM. A machine language routine for doing this is included in a sample program at the entry for 56576 ($DD00).

As indicated above, you are by no means limited to using the character data furnished by the ROM. The availability of user-defined characters greatly extends the graphics power of the 64. It allows you to create special text characters, like math or chemistry symbols and foreign language alphabets. You can also develop special graphics characters as a substitute for plotting points in bitmap graphics. You can achieve the same resolution using a custom character as in high-resolution bitmap mode, but with less memory. Once you have defined the character, it is much faster to print it to the screen than it would be to plot out all of the individual points.

To employ user-defined characters, you must first pick a spot to put the shape data. This requires choosing a bank of 16K for video

chip memory (see the entry under Bits 0-1 of 56576, $DD00 for the considerations involved), and setting the pointer to the 2K area of character memory in 53272 ($D018). It is then up to you to supply the shape data for the characters. You can use part of the ROM character set by reading the ROM and transferring the data to your character shape area (see the entry for location 1 for a method of reading the ROM).

Your original characters may be created by darkening squares on an 8 by 8 grid, converting all darkened squares to their bit values, and then adding the bit values for each of the eight bytes. Or, you may use one of the many character graphics editor programs that are available commercially to generate the data interactively by drawing on the screen.

One graphics mode, multicolor text mode, almost requires that you define your own character set in order to use it effectively. Multicolor mode is enabled by Bit 4 of location 53270 ($D016). Instead of using each bit to control whether an individual dot will be foreground color (1) or background color (0), that mode breaks down each byte of shape data in four bit-pairs. These bit-pairs control whether a dot will be the color in Background Color Register #0 (00), the color in Background Color Register #1 (01), the color in Background Color Register #2 (10), or the color in the appropriate nybble of Color RAM (11). Since each pair of bits controls one dot, each character is only four dots across. To make up for this, each dot is twice as wide as a normal high-resolution dot.

Chapter 7

8K Operating System Kernal ROM

8K Operating System Kernal ROM

Locations 57344 to 65535 ($E000 to $FFFF) are used by the 8K Operating System Kernal ROM. This ROM contains the master control program that directs all of the computer's input and output. When you turn the 64's power on, this program takes over and performs the various internal tasks which are necessary to enable you to use the computer. These include such things as clearing RAM, getting the I/O devices and video display chip ready for use, setting various pointers, checking for autostart cartridges, and setting up BASIC if no such cartridges are found.

The Kernal ROM contains many useful I/O routines that you might wish to incorporate in your own programs. However, the Kernal ROM is subject to revision (it was revised twice in the first six months alone), and these routines are not guaranteed to stay in the same place. This means that if your programs make direct jumps into the ROM, there is no telling if your program will work with future versions of the Kernal on later Commodore models. To help avoid these incompatability problems, Commodore provides a jump table at the end of the ROM which points to the current location of each of a number of these routines in any given Kernal.

This jump table has been a feature of all Commodore computers from the beginning, and has been expanded in the 64 to include 39 separate routines. The jump vectors for the original 15 routines which have always been included in the table (such as the one to the routine which outputs a character) are the same as on previous Commodore computers. By having your program jump to the table address rather than directly to the routine desired, you can guarantee that your program will work without modification on future versions of the 64, as well as on future Commodore models.

To the extent that your programs do not require the enhanced graphics capabilities of the 64, they can also be made to work with the PET and VIC-20 computers. So remember: While jumping to undocumented entry points may save time in the short run, in the long run it may cause compatibility problems and should be avoided.

As with the section on the BASIC ROM, this section is not meant to be a complete explanation of the Kernal ROM, but rather a guidepost for further exploration. Where the exact instructions the Kernal ROM routines use are important to your programming, it will be necessary for you to obtain a disassembly listing of those routines and look at the code itself.

Keep in mind that there is 8K of RAM underlying the Kernal ROM that can be used by turning off interrupts and switching out the Kernal ROM temporarily. Even without switching out the Kernal ROM, this RAM may be read by the VIC-II chip if it is banked to use the top 16K of memory, and may be used for graphics data. The Kernal and BASIC ROMs may be copied to RAM, and the RAM versions modified to change existing features or add new ones.

There are some differences between the version of the Kernal found on the first few Commodore 64s and those found on the majority of newer models. Those differences are discussed in the entries for the sections on later Kernal additions (*patches*) at 58541-58623 ($E4AD-$E4FF) and 65371-65407 ($FF5B-$FF7F).

The most obvious change causes the Color RAM at 55296 ($D800) to be initialized to the foreground color when the screen is cleared on newer models, instead of background color white as on the earlier models. Other changes allow the new Kernal software to be used by either U.S. or European 64s. Keep in mind that the Kernal is always subject to change, and that the following discussion, while accurate at the time written (mid-1984), may not pertain to later models. If future changes are like past ones, however, they are likely to be minor ones. The first place to look for these changes would be in the *patch* sections identified above.

57344 $E000

Continuation of EXP Routine

This routine is split, with part on the BASIC ROM and the other part here. Since the two ROMs do not occupy contiguous memory as on most Commodore machines, the BASIC ROM ends with a JMP $E000 instruction. Thus, while the BASIC interpreter on the 64 is for the most part the same as on the VIC, the addresses for routines in this ROM are displaced by three bytes from their location on the VIC.

57411 $E043 POLY1

Function Series Evaluation Subroutine 1

This routine is used to evaluate more complex expressions, and calls the following routine to do the intermediate evaluation.

57433 $E059 POLY2

Function Series Evaluation Subroutine 2

This is the main series evaluation routine, which evaluates expressions by using a table of the various values that must be operated on in sequence to obtain the proper result.

57485 $E08D RMULC

Multiplicative Constant for RND

A five-byte floating point number which is multiplied by the seed value as part of the process of obtaining the next value for RND.

57490 $E092 RADDC

Additive Constant for RND

The five-byte floating point number stored here is added to the seed as part of the process of obtaining the value for RND.

57495 $E097 RND

Perform RND

This routine comes up with a random number in one of three ways, depending on the argument X of RND(X). If the argument is positive, the next RND value is obtained by multiplying the seed value in location 139 ($8B) by one of the constants above, adding the other constant, and scrambling the resulting bytes. This produces the next number in a sequence. So many numbers can be produced in this way before the sequence begins to repeat that it can be considered random.

If the argument is negative, the argument itself is scrambled, and made the new seed. This allows creation of a sequence that can be duplicated.

If the argument is 0, four bytes of the Floating Point Accumulator are loaded from the low and high byte of Timer A, and the tenths of second and second Time of Day Clock registers, all on CIA #1. This provides a somewhat random value determined by the setting of those timers at the moment that the command is executed, which becomes the new seed value. The RND(1) command should then be used to generate further random numbers.

The RND(0) implementation on the 64 has serious problems which make it unsuitable for generating a series of random numbers when used by itself. First of all, the Time of Day Clock on CIA #1 (see 56328-56331, $DC08-$DC0B) does not start running until you write to the tenth of second register. The Operating System never starts this clock, and therefore the two registers used as part of the floating point RND(0) value always have a value of 0. Even if the clock was started, however, these registers keep time in Binary Coded Decimal (BCD) format, which means that they do not produce a full

range of numbers from 0 to 255. In addition, the Timer A high register output ranges only from 0 to 66, which also limits the range of the final floating point value, so that certain numbers are never chosen.

57593 $E0F9

Call Kernal I/O Routines

This section is used when BASIC wants to call the Kernal I/O routines CHROUT, CHRIN, CHKOUT, CHKIN, and GETIN. It handles any errors that result from the call, and creates a 512-byte buffer space at the top of BASIC and executes a CLR if the RS-232 device is opened.

57642 $E12A SYS

Perform SYS

Before executing the machine language subroutine (JSR) at the address indicated, the .A, .X, .Y, and .P registers are loaded from the storage area at 780-783 ($30C-$30F). After the return from subroutine (RTS), the new values of those registers are stored back at 780-783 ($30C-$30F).

57686 $E156 SAVE

Perform SAVE

This routine sets the range of addresses to be saved from the start of BASIC program text and end of BASIC program text pointers at 43 ($2B) and 45 ($2D), and calls the Kernal SAVE routine. This means that any area of memory can be saved by altering these two pointers to point to the starting and ending address of the desired area, and then changing them back.

57701 $E165 VERIFY

Perform VERIFY

This routine sets the load/verify flag at 10 ($A), and falls through to the LOAD routine.

57704 $E168 LOAD

Perform LOAD

This routine sets the load address to the start of BASIC (from pointer at 43, $2B), and calls the Kernal LOAD routine. If the load is successful, it relinks the BASIC program so that the links agree with the address to which it is loaded, and it resets the end of BASIC pointer to reflect the new end of program text. If the LOAD was done while a program was running, the pointers are reset so that the program starts executing all over again from the beginning. A CLR is not performed, so that the variables built so far are retained, and their

values are still accessible. The pointer to the variable area is not changed, but if the new program is longer than the one that loaded it, the variable table will be partially overwritten. This will cause errors when the overwritten variables are referenced. Likewise, strings whose text was referenced at its location within the original program listing will be incorrect.

Since a LOAD from a program causes the program execution to continue at the first line, when loading a machine language routine or data file with a nonrelocating load (for example, LOAD"FILE",8,1) from a program, you should read a flag and GOTO the line after the LOAD if you don't want the program to keep rerunning indefinitely:

```
10 IF FLAG=1 THEN GOTO 30
20 FLAG=1: LOAD "FILE",8,1
30 REM PROGRAM CONTINUES HERE
```

57790 $E1BE OPEN

Perform OPEN

The BASIC OPEN statement calls the Kernal OPEN routine.

57799 $E1C7 CLOSE

Perform CLOSE

The BASIC CLOSE statement calls the Kernal CLOSE routine.

57812 $E1D4

Set Parameters for LOAD, VERIFY, and SAVE

This routine is used in common by LOAD, SAVE, and VERIFY for setting the filename, the logical file, device number, and secondary address, all of which must be done prior to these operations.

57856 $E200

Skip Comma and Get Integer in .X

This subroutine is used to skip the comma between parameters and get the following integer value in the .X register.

57862 $E206

Fetch Current Character and Check for End of Line

This subroutine gets the current character, and if it is a 0 (end of line), it pulls its own return address off the stack and returns. This terminates both its own execution and that of the subroutine which called it.

57870 $E20E
Check for Comma

This subroutine checks for a comma, moves the text pointer past it if found, and returns an error if it is not found.

57881 $E219
Set Parameters for OPEN and CLOSE

This routine is used in common by OPEN and CLOSE for setting the filename, the logical file, device number, and secondary address, all of which must be done prior to these operations.

57956 $E264 COS
Perform COS

COS is executed by adding PI/2 to the contents of FAC1 and dropping through to SIN.

57963 $E26B SIN
Perform SIN

This routine evaluates the SIN of the number in FAC1 (which represents the angle in radians), and leaves the result there.

58036 $E2B4 TAN
Perform TAN

This routine evaluates the tangent of the number in FAC1 (which represents the angle in radians) by dividing its sine by its cosine.

Location Range: 58080-58125 ($E2E0-$E30D)
Table of Constants for Evaluation of SIN, COS, and TAN

58080 $E2E0 PI2
The Five-Byte Floating Point Representation of the Constant PI/2

58085 $E2E5 TWOPI
The Five-Byte Floating Point Representation of the Constant 2*PI

58090 $E2EA FR4
The Five-Byte Floating Point Representation of the Constant 1/4

58095 $E2EF SINCON
Table of Constants for Evaluation of SIN, COS, and TAN

This table starts with a counter byte of 5, indicating that there are six entries in the table. This is followed by the six floating point constants of five bytes each.

58126 $E30E ATN

Perform ATN

The arc tangent of the number in FAC1 (which represents the angle in radians) is evaluated using the 12-term series of operations from the constant table which follows. The answer is left in FAC1.

58174 $E33E ATNCON

Table of Constants for ATN Evaluation

The table begins with a count byte of 11, which is followed by 12 constants in five-byte floating point representation.

58235 $E37B

Warm Start BASIC

This is the entry point into BASIC from the BRK routine at 65126 ($FE66), which is executed when the STOP and RESTORE keys are both pressed. It first executes the Kernal CLRCHN routine which closes all files. It then sets the default devices, resets the stack and BASIC program pointers, and jumps through the vector at 768 ($300) to the next routine to print the READY prompt and enter the main BASIC loop.

58251 $E38B

Error Message Handler

This routine to print error messages is pointed to by the vector at 768 ($300). Using the .X register as an index, it either prints an error message from the table at 41363 ($A193) or the READY prompt, and continues through the vector at 770 ($302) to the main BASIC loop.

58260 $E394

Cold Start BASIC

This initialization routine is executed at the time of power-up. The RAM vectors to important BASIC routines are set up starting at 768 ($300), the interpreter is initialized, the start-up messages are printed, and the main loop entered through the end of the warm start routine.

58274 $E3A2 INITAT

Text of the CHRGET Routine Which Runs at 115 ($73)

The text of the CHRGET routine is stored here, and moved to Page 0 by the BASIC initialization routine. When creating a wedge in CHRGET, it is possible to execute all or part of this code in place of the RAM version.

58298 $E3BA

Initial RND Seed Value

At power-up time, this five-byte floating point constant is transferred to 139 ($8B), where it functions as the starting RND seed number. Thus, if RND is not initialized with a negative or zero argument, it will always return the same sequence of numbers.

58303 $E3BF INIT

Initialize BASIC

This routine is called by the cold start routine to initialize all of the BASIC zero-page locations which have a fixed value. This includes copying the CHRGET routine from the ROM location above, to 115 ($73).

58402 $E422

Print BASIC Start-Up Messages

This routine prints the start-up message "**** COMMODORE 64 BASIC V2 ****", calculates the amount of free memory, and prints the BYTES FREE message.

58439 $E447

Table of Vectors to Important BASIC Routines

This table contains the vectors which point to the addresses of some important BASIC routines. The contents of this table are moved to the RAM table at 768 ($300).

58451 $E453

Copy BASIC Vectors to RAM

The cold start routine calls this subroutine to copy the table of vectors to important BASIC routines to RAM, starting at location 768 ($300).

58464 $E460 WORDS

Power-Up Messages

The ASCII text of the start-up messages "**** COMMODORE 64 BASIC V2 ****" and "BYTES FREE" is stored here.

Location Range: 58541-65535 ($E4AD-$FFFF)

Kernal I/O Routines

After the conclusion of BASIC comes the part of this ROM which can be considered the Kernal proper. This part contains all of the vectored routines found in the jump table starting at 65409 ($FF81).

Location Range: 58541-58623 ($E4AD-$E4FF)

Patches Added to Later Kernal Versions

This area contains code that was not found in the original version of the Kernal. These additions were made to fix some bugs and to increase Kernal compatibility between U.S. and European 64s.

58541 $E4AD

Patch for BASIC Call of CHKOUT

This patch was made to preserve the .A register if there was no error returned from BASIC's call of the Kernal CHKOUT routine. Apparently, the first version could cause a malfunction of the CMD and PRINT# commands.

58551 $E4B7

28 Unused Bytes (all have the value of 170, $AA)

58579 $E4DA

Patch to RS232 Routine

58586 $E4DA

Clear Color RAM to the Color of Text Foreground

This routine is a patch added to the more recent versions of the Kernal. It is called by the routine which clears a screen line (59903, $E9FF), and it places the value of the current foreground color in 646 ($286) into the current byte of Color RAM pointed to by USER (243, $F3).

In the original version of the Kernal, the routine that cleared a screen line set the corresponding Color RAM to a value of 1, which gives text characters a white foreground color. This was changed in version zero because the white color was found to sometimes cause light flashes during screen scrolling. It was that white foreground color, however, that enabled the user to POKE the screen code for a character into screen RAM, and make that character appear on the screen in a color that contrasted the blue background. This change to the Operating System caused colors POKEd to screen RAM to be the same color as the background, and thus made them invisible. By initializing Color RAM to the current foreground color, version 3 of the Kernal solved this problem.

Owners of old 64s can find the solution to the "invisible POKE" problem in the discussion that begins on the bottom of page 81.

58592 $E4E0

Pause after Finding a File on Cassette

This routine is a patch to the routine which finds a file on cassette. After the file is found, the message FILETITLE FOUND appears on the screen. On the original versions of the Kernal, the user would then have to hit the Commodore key to continue the load. On the newer versions, this patch causes a slight pause after a tape file is found, during which time a keypress is looked for. If a key is pressed, the loading process continues immediately. If it is not, the load continues by itself after the end of the pause.

58604 $E4EC

Baud Rate Table for European (PAL) Standard Monitors

This table of prescaler values was added to later Kernal versions to allow the same Kernal software to be used with either U.S. or European 64s. It contains the values which are required to obtain interrupts at the proper frequency for the standard RS-232 baud rates, and corresponds exactly in format to the table of values for the U.S. (NTSC) monitor format at 65218 ($FEC2). Separate tables are required because the prescaler values are derived from dividing the system clock rate by the baud rate, and PAL machines operate with a slightly slower clock frequency.

58624 $E500 IOBASE

Store Base Address of Memory-Mapped I/O Devices in .X and .Y Registers

This is one of the documented Kernal routines for which there is a vector in the jump table at 65523 ($FFF3).

When called, this routine sets the .X register to the low byte of the base address of the memory-mapped I/O devices, and puts the high byte in the .Y register. This allows a user to set up a zero-page pointer to the device, and to load and store indirectly through that pointer. A program which uses this method, rather than directly accessing such devices, could be made to function without change on future Commodore models, even though the I/O chips may be addressed at different locations. This of course assumes that the CIA or a similar chip will be used. This routine is of limited value for creating software that is compatible with both the VIC-20 and the 64 because of the differences in the VIA I/O chip that VIC uses.

The current version of this routine loads the .X register with a 0, and the .Y register with 220 ($DC), thus pointing to CIA #1, which is at 56320 ($DC00).

58629 $E505 SCREEN

Store Number of Screen Rows and Columns in .Y and .X

This is a documented Kernal routine which is vectored in the jump table at 65517 ($FFED), and is provided to allow for program compatibility between computers.

When called, this subroutine returns the number of screen columns in the .X register, and the number of screen rows in .Y. Thus, a program can detect the screen format of the machine on which it is running, and make sure that text output is formatted accordingly.

The present version of this routine loads the .X register with 40 ($28) and the .Y register with 25 ($19).

58634 $E50A PLOT

Read/Set Location of the Cursor

The jump table entry for this documented Kernal is at 65520 ($FFF0).

The routine allows the user to read or set the position of the cursor. If the carry flag is set with a SEC instruction before calling this subroutine, cursor column (X position) will be returned in the .X register, and the cursor row (Y position) will be returned in the .Y register. If the carry flag is cleared with a CLC instruction before entering this routine, and the .Y and .X registers are loaded with the desired row and column positions respectively, this routine will set the cursor position accordingly.

The current read routine loads .X and .Y from locations 214 ($D6) and 211 ($D3) respectively. The cursor set routine stores .X and .Y in these locations, and calls the routine that sets the screen pointers at 58732 ($E56C).

The user can access this routine from BASIC by loading the .X, .Y, and .P register values desired to the save area starting at 780 ($30C).

58648 $E518

Initialize Screen and Keyboard

This is the original CINT Kernal routine, to which additions were made in later versions of the Kernal.

After calling the routine at 58784 ($E5A0) to set up default I/O values, this routine initializes the cursor blink flags, the keyboard decode vector, the key repeat delay and frequency counters, the current color code, and maximum keyboard buffer size. It then falls through to the next routine.

58692 $E544

Initialize the Screen Line Link Table and Clear the Screen

This routine initializes the screen line link table at 217 ($D9), clears

the screen, and clears the Color RAM to the background color. It falls through to the next routine.

58726 $E566

Home the Cursor

This routine sets PNTR (211, $D3) and TBLX (214, $D6) to 0, and falls through to the next routine.

58732 $E56C

Set Pointer to Current Screen Line

This routine sets the pointer PNT (209, $D1) to the address of the first byte of the current logical line. In figuring this address, it takes into account the status of the screen line link table, which can indicate that two physical lines should be joined as one logical line.

58784 $E5A0

Set Default I/O Devices and Set Default Values for VIC-II Chip Registers

This routine sets the keyboard and screen as the current input and output devices. It then writes the default values found in the table at 60601 ($ECB9) to the VIC-II chip.

58804 $E5B4 LP2

Get a Character from the Keyboard Buffer

This routine transfers the first character from the keyboard buffer to the .A register, bumps the rest of the characters one place up in line, and decrements the pointer, showing how many characters are waiting in the buffer.

58826 $E5CA

Wait for a Carriage Return from the Keyboard

This subroutine is called by the portion of the CHRIN routine that handles keyboard input. It turns the cursor on, gets characters, and echoes them to the screen until a carriage return has been entered. It also looks for the shifted RUN/STOP key, and forces the output of the commands LOAD and RUN if it finds it.

58930 $E632

Input a Character from Screen or Keyboard

This routine is the portion of the Kernal CHRIN routine that handles input from the keyboard and screen devices. CHRIN gets one byte at a time from the current screen position, or inputs a whole line from the keyboard and returns it one byte at a time.

59012 $E684

Test for Quote Marks

This subroutine checks if the current character is a quotation mark, and if it is, toggles the quote switch at 212 ($D4).

59025 $E691

Add a Character to the Screen

This is part of the routine that outputs a character to the screen. It puts printable characters into screen memory.

59048 $E6A8

Return from Outputting a Character to the Screen

This is the common exit point for the screen portion of the CHROUT routine.

59062 $E6B6

Advance the Cursor

This routine advances the cursor, and provides for such things as scrolling at the end of the screen, and inserting a blank line in order to add another physical line to the current logical line.

59137 $E701

Move Cursor Back over a 40-Column Line Boundary

59158 $E716

Output to the Screen

This is the main entry point for the part of CHROUT that handles output to the screen device. It takes the ASCII character number, and tests if the character is printable. If it is, it prints it (taking into consideration the reverse flag, if any inserts are left, etc.). If it is a nonprinting character, the routine performs the appropriate cursor movement, color change, screen clearing, or whatever else might be indicated.

59516 $E87C

Move Cursor to Next Line

This subroutine moves the cursor down to the next line if possible, or scrolls the screen if the cursor is on the last line.

59537 $E891

Output a Carriage Return

A carriage return is performed by clearing insert mode, reverse video, and quote mode, and moving the cursor to the next line.

59553 $E8A1

If at the Beginning of a Screen Line, Move Cursor to Previous Line

59571 $E8B3

If at the End of a Screen Line, Move Cursor to the Next Line

59595 $E8CB

Check for a Color Change

This routine is used by the screen CHROUT routine to check if the character to be printed is one that causes the current foreground color to change (such as the CTRL-1 combination).

59610 $E8DA

PETASCII Color Code Equivalent Table

This table gives the PETASCII values of the color change characters for each of the 16 possible colors. These values are:

144 ($90)	Change to color 0 (black)
5 ($05)	Change to color 1 (white)
28 ($1C)	Change to color 2 (red)
159 ($9F)	Change to color 3 (cyan)
156 ($9C)	Change to color 4 (purple)
30 ($1E)	Change to color 5 (green)
31 ($1F)	Change to color 6 (blue)
158 ($9E)	Change to color 7 (yellow)
129 ($81)	Change to color 8 (orange)
149 ($95)	Change to color 9 (brown)
150 ($96)	Change to color 10 (light red)
151 ($97)	Change to color 11 (dark gray)
152 ($98)	Change to color 12 (medium gray)
153 ($99)	Change to color 13 (light green)
154 ($9A)	Change to color 14 (light blue)
155 ($9B)	Change to color 15 (light gray)

59626 $E8EA

Scroll Screen

This subroutine moves all of the screen lines up, so that a blank line is created at the bottom of the screen and the top screen line is lost. If the top logical line is two physical lines long, all lines are moved up two lines. Holding down the CTRL key will cause a brief pause after the scroll.

59749 $E965

Insert a Blank Line on the Screen

This subroutine is used when INSERTing to add a blank physical line to a logical line.

59848 $E9C8

Move Screen Line

This subroutine is used by the scroll routine to move one screen line (and its associated Color RAM) up a line.

59872 $E9E0

Set Temporary Color Pointer for Scrolling

This subroutine sets up a pointer in 174-175 ($AE-$AF) to the Color RAM address that corresponds to the temporary screen line address in 172-173 ($AC-$AD).

59888 $E9F0

Set Pointer to Screen Address of Start of Line

This subroutine puts the address of the first byte of the screen line designated by the .X register into locations 209-210 ($D1-$D2).

59903 $E9FF

Clear Screen Line

This subroutine writes space characters to an entire line of screen memory, and clears the corresponding line of color memory to color in Background Color Register 0 (53281, $D021).

59923 $EA13

Set Cursor Blink Timing and Color Memory Address for Print to Screen

This subroutine sets the cursor blink countdown and sets the pointer to Color RAM. It then falls through to the next routine.

59932 $EA1C

Store to Screen

This routine stores the character in the .A register to the screen address pointed to by 209 ($D1), and stores the color in the .X register to the address pointed to by 243 ($F3).

59940 $EA24

Synchronize Color RAM Pointer to Screen Line Pointer

This subroutine sets the pointer at 243 ($F3) to the address of the beginning of the line of Color RAM which corresponds to the current line of screen RAM (whose pointer is at 209, $D1).

59953 $EA31

IRQ Interrupt Entry

This is the entry point to the standard IRQ interrupt handler. Timer A of CIA #1 is set at power-on to cause an IRQ interrupt to occur

every 1/60 second. When the interrupt occurs, program flow is transferred here via the CINV vector at 788 ($314). This routine updates the software clock at 160-162 ($A0-$A2), handles the cursor flash, and maintains the tape interlock which keeps the cassette motor on if a button is pushed and the interlock flag is on. Finally, it calls the keyboard scan routine, which checks the keyboard and puts any character it finds into the keyboard buffer.

60039 $EA87 SCNKEY

Read the Keyboard

This subroutine is called by the IRQ interrupt handler above to read the keyboard device which is connected to CIA #1 (see the entry for 56320, $DC00 for details on how to read the keyboard).

It is the Kernal routine SCNKEY which can be entered from the jump table at 65439 ($FF9F). This routine returns the keycode of the key currently being pressed in 203 ($CB), sets the shift/control flag if appropriate, and jumps through the vector at 655 ($28F) to the routine that sets up the proper table to translate the keycode to PETASCII. It concludes with the next routine, which places the PETASCII value of the character in the keyboard buffer.

60128 $EAE0

Decode the Keystroke and Place Its ASCII Value in the Keyboard Buffer

This is the continuation of the IRQ keyscan routine. It decodes the keycode with the proper PETASCII table, and compares it with the last keystroke. If it is the same, it checks to see if it is okay to repeat the character without waiting for the key to be let up. If the character should be printed, it is moved to the end of the keyboard buffer at 631 ($277).

60232 $EB48

Set Up the Proper Keyboard Decode Table

This routine is pointed to by the vector at 655 ($28F). Its function is to read the shift/control flag at 653 ($28D), and set the value of the decode table pointer at 245 ($F5) accordingly.

First it checks if the SHIFT/Commodore logo combination was pressed, and if the toggle enable at 657 ($291) will allow a change, the character set will be changed to lowercase/uppercase or uppercase/graphics by changing the VIC Memory Control Register at 53272 ($D018), and no character will be printed.

Next it sets the decode table pointer. There are 64 keys, and each can have four different PETASCII values, depending on whether the key is pressed by itself, or in combination with the SHIFT, CTRL, or Commodore logo keys. Therefore, there are four tables of

64 entries each to translate the keycode to PETASCII: the standard table, the SHIFT table, the Commodore logo table, and the CONTROL table. The routine will set up the pointer for the appropriate table, depending on whether the SHIFT, CTRL, or logo key was pressed. The CTRL key takes precedence, so that if another of these keys is pressed along with the CTRL key, the CONTROL table is used.

60281 $EB79

Keyboard Decode Table Vectors

This table contains the two-byte addresses of the four keyboard decode tables in low-byte, high-byte format.

60289 $EB81

Standard Keyboard Matrix Decode Table

This table contains the 64 PETASCII values for the standard keyboard, one for each key which is struck by itself. The table is in keycode order (see the keycode table in Appendix H for the correspondence of keycode to key). A 65th byte with the value of 255 ($FF) marks the end of the table (this corresponds to a keypress value of 64, no key pressed).

60354 $EBC2

SHIFTed Keyboard Matrix Decode Table

This table contains the 64 PETASCII values for the shifted keyboard, one for each key which is struck while the SHIFT key is pressed. The table is in keycode order (see the keycode table in Appendix H for the correspondence of keycode to key). A 65th byte with the value of 255 ($FF) marks the end of the table (this corresponds to a keypress value of 64, no key pressed).

60419 $EC03

Commodore Logo Keyboard Matrix Decode Table

This table contains the 64 PETASCII values for the logo keyboard, one for each key which is struck while the Commodore logo key is pressed. The table is in keycode order (see the keycode table in Appendix H for the correspondence of keycode to key). A 65th byte with the value of 255 ($FF) marks the end of the table (this corresponds to a keypress value of 64, no key pressed).

60484 $EC44

Set Lowercase/Uppercase or Uppercase/Graphics Character Set

The part of the Kernal CHROUT routine that outputs to the screen uses this subroutine to check for the special nonprinting characters that switch the character set (CHR$(14) and CHR$(142)). If one of

these is the character to be printed, this routine makes the switch by setting location 53272 ($D018) accordingly.

60510 $EC5E

Set Flag to Enable or Disable Switching Character Sets

This subroutine is also used to check for special characters to print. In this case, it checks for the characters that enable or disable the SHIFT/logo combination from toggling the character set currently in use (CHR$(8) and CHR$(9)). If one of these is to be printed, the flag at 657 ($291) is changed.

60536 $EC78

Control Keyboard Matrix Decode Table

This table contains the 64 PETASCII values for the Control keyboard, one for each key which is struck while the CTRL key is pressed. The table is in keycode order (see the keycode table in Appendix H for the correspondence of keycode to key). A 65th byte with the value of 255 ($FF) marks the end of the table (this corresponds to a keypress value of 64, no key pressed).

The only keys generally struck in combination with the CTRL key are the ones that change the colors on the top row of the keyboard, but this doesn't necessarily mean that the other CTRL key combinations don't do anything. On the contrary, looking at the values in this table, you can see that any of the first 32 values in the PETASCII table can be produced by some combination of the CTRL key and another key. CTRL-@ produces CHR$(0). CTRL-A through CTRL-Z produce CHR$(1) through CHR$(26). CTRL-: is the same as CHR$(27), CTRL-Lira (that's the slashed-L British pound sign) produces CHR$(28), CTRL-; equals CHR$(29), CTRL-up arrow produces CHR$(30), and CTRL-= produces CHR$(31).

Any of these combinations produce the same effect as the CHR$(X) statement. For example, CTRL-; moves the cursor over to the right, CTRL-N switches to lowercase, CTRL-R turns on reverse video, and CTRL-E changes the printing to white.

60601 $ECB9

Video Chip Register Default Table

This table contains the default values that are stored in the 47 VIC-II chip registers. It is interesting to note that this table appears to be incomplete. While Sprite Color Registers 0-6 are initialized to values of 1–7, Sprite Color Register 7 is initialized to 76—the ASCII value of the letter L which begins on the next table.

60647 $ECE7

Text for Keyboard Buffer When SHIFT/RUN Is Pressed

When the SHIFT and RUN keys are pressed, the ASCII text stored

here is forced into the keyboard buffer. That text is LOAD, carriage return, RUN, carriage return.

60656 $ECF0

Low Byte Table of Screen Line Addresses

This table holds the low byte of the screen address for lines 0-24. The high byte is derived from combining a value from the screen line link table at 217 ($D9) with the pointer to screen memory at 648 ($288).

60681 $ED09 TALK

Send TALK to a Device on the Serial Bus

This is a documented Kernal routine whose entry in the jump table is 65460 ($FFB4). When called, it ORs the device number in the Accumulator with the TALK code (64, $40) and sends it on the serial bus. This commands the device to TALK.

60684 $ED0C LISTEN

Send LISTEN to a Device on the Serial Bus

This is a documented Kernal routine whose entry in the jump table is 65457 ($FFB1). When called, it ORs the device number in the Accumulator with the LISTEN code (32, $20) and sends it on the serial bus. This commands the device to LISTEN.

60689 $ED11

Send Command Code to a Device on the Serial Bus

This subroutine is used in common by many Kernal routines to send the command code in the Accumulator to a device on the serial bus.

60736 $ED40

Send a Byte on the Serial Bus

This subroutine is used in common by several Kernal routines to send the byte in the serial bus character buffer at 149 ($95) on the serial bus.

60848 $EDB0

Time-Out Error on Serial Bus

This subroutine handles the case when the device does not respond by setting the DEVICE NOT PRESENT error code and exiting.

60857 $EDB9 SECOND

Send a Secondary Address to a Device on the Serial Bus after LISTEN

This is a documented Kernal routine that can be entered from the jump table at 65427 ($FF93). It sends a secondary address from the

Accumulator to the device on the serial bus that has just been commanded to LISTEN. This is usually done to give the device more particular instructions on how the I/O is to be carried out before information is sent.

60871 $EDC7 TKSA

Send a Secondary Address to a Device on the Serial Bus after TALK

This is a documented Kernal routine that can be entered from the jump table at 65430 ($FF96). It sends a secondary address from the Accumulator to the device on the serial bus that has just been commanded to TALK. This is usually done to give the device more particular instructions on how the I/O is to be carried out before information is sent.

60893 $EDDD CIOUT

Send a Byte to an I/O Device over the Serial Bus

This is a documented Kernal routine which can be entered from the jump table at 65448 ($FFA8). Its purpose is to send a byte of data over the serial bus. In order for the data to be received, the serial device must have first been commanded to LISTEN and been given a secondary address if necessary. This routine always buffers the current character, and defers sending it until the next byte is buffered. When the UNLISTEN command is sent, the last byte will be sent with an End or Identify (EOI).

60911 $EDEF UNTLK

Send UNTALK to a Device on the Serial Bus

This is a documented Kernal routine whose entry in the jump table is 65451 ($FFAB). When called, it sends the UNTALK code (95, $5F) on the serial bus. This commands any TALKer on the bus to stop sending data.

60926 $EDFE UNLSN

Send UNLISTEN to a Device on the Serial Bus

This is a documented Kernal routine whose entry in the jump table is 65454 ($FFAE). It sends the UNLISTEN code (63, $3F) on the serial bus. This commands any LISTENers to get off the serial bus, and frees up the bus for other users.

60947 $EE13 ACPTR

Receive a Byte of Data from a Device on the Serial Bus

This is a documented Kernal routine whose entry point in the jump table is 65445 ($FFA5). When called, it will get a byte of data from the current TALKer on the serial bus and store it in the Accumulator.

In order to receive the data, the device must have previously been sent a command to TALK and a secondary address if it needs one.

61061 $EE85

Set the Serial Clock Line Low (Active)

This subroutine clears the serial bus clock pulse output bit (Bit 4 of CIA #2 Data Port A at 56576, $DD00).

61070 $EE8E

Set the Serial Clock Line High (Inactive)

This subroutine sets the serial bus clock pulse output bit to 1 (Bit 4 of CIA #2 Data Port A at 56576, $DD00).

61079 $EE97

Set Serial Bus Data Output Line Low

This subroutine clears the serial bus data output bit to 0 (Bit 5 of CIA #2 Data Port A at 56576, $DD00).

61088 $EEA0

Set Serial Bus Data Output Line High

This subroutine sets the serial bus data output bit to 1 (Bit 5 of CIA #2 Data Port A at 56576, $DD00).

61097 $EEA9

Get Serial Bus Data Input Bit and Clock Pulse Input Bit

This subroutine reads the serial bus data input bit and clock pulse input bit (Bits 7 and 6 of CIA #2 Data Port A at 56576, $DD00), and returns the data bit in the Carry flag and the clock bit in the Negative flag.

61107 $EEB3

Perform a One-Millisecond Delay

61115 $EEBB

Send Next RS-232 Bit (NMI)

This subroutine is called by the NMI interrupt handler routine to send the next bit of data to the RS-232 device.

61230 $EF2E

Handle RS-232 Errors

This subroutine sets the appropriate error bits in the status register at 663 ($297).

61258 $EF4A

Set the Word Length For the Current RS-232 Character

This routine takes the number of data bits to send per RS-232 character from the control register and puts it into the .X register for use by the RS-232 routines.

61273 $EF59

Receive Next RS-232 Bit (NMI)

This routine is called by the NMI interrupt handler routine to receive the next bit of data from the RS-232 device.

61310 $EF7E

Setup to Receive a New Byte from RS-232

61328 $EF90

Test If Start Bit Received from RS-232

61335 #EF97

Put a Byte of Received Data into RS-232 Receive Buffer

This routine checks for a Receive Buffer Overrun, stores the byte just received in the RS-232 receive buffer, and checks for Parity Error, Framing Error, or Break Detected Error. It then sets up to receive the next byte.

61409 #EFE1

CHKOUT for the RS-232 Device

The Kernal CHKOUT routine calls this subroutine to define the RS-232 device's logical file as an output channel. Before this can be done, the logical file must first be OPENed.

61460 $F014

CHROUT for the RS-232 Device

The Kernal CHROUT routine calls this subroutine to output a character to the RS-232 device. After the logical file has been OPENed and set for output using CHKOUT, the CHROUT routine is used to actually send a byte of data.

61517 $F04D

CHKIN for the RS-232 Device

The Kernal CHKIN routine calls this subroutine to define the RS-232 device's logical file as an input channel. A prerequisite for this is that the logical file first be OPENed.

61574 $F086

GETIN for the RS-232 Device

The Kernal GETIN routine calls this subroutine to remove the next byte of data from the RS-232 receive buffer and return it in the Accumulator. The routine checks for the Receive Buffer Empty Error. It is also called by the Kernal CHRIN routine, which essentially does the same thing as GETIN for the RS-232 device.

61604 $F0A4

Stop CIA #2 RS-232 NMIs for Serial/Cassette Routines

This subroutine turns off the NMIs that drive the RS-232 routines before any I/O is done using the serial bus or cassette device. Such interrupts could throw off the timing of those I/O routines, and interfere with the transmission of data.

61629 $F0BD

Kernal Control Messages

The ASCII text of the Kernal I/O control messages is stored here. The last byte of every message has Bit 7 set to 1 (ASCII value+128). The messages are:

I/O ERROR
SEARCHING
FOR
PRESS PLAY ON TAPE
PRESS RECORD & PLAY ON TAPE
LOADING
SAVING
VERIFYING
FOUND
OK

61739 $F12B

Print Kernal Error Message If in Direct Mode

This routine first checks location 157 ($9D) to see if the messages are enabled. If they are, it prints the message indexed by the .Y register.

61758 $F13E GETIN

Get One Byte from the Input Device

This is a documented Kernal routine whose jump table entry point is at 65508 ($FFE4). The routine jumps through a RAM vector at 810 ($32A). Its function is to get a character from the current input device (whose device number is stored at 153, $99). In practice, it operates identically to the CHRIN routine below for all devices except for the keyboard. If the keyboard is the current input device, this routine

gets one character from the keyboard buffer at 631 ($277). It depends on the IRQ interrupt routine to read the keyboard and put characters into the buffer.

61783 $F157 CHRIN

Input a Character from the Current Device

This is a documented Kernal routine whose jump table entry point is at 65487 ($FFCF).

The routine jumps through a RAM vector at 804 ($324). Its function is to get a character from the current input device (whose device number is stored at 153, $99). This device must first have been OPENed and then designated as the input channel by the CHKIN routine.

When this routine is called, the next byte of data available from this device is returned in the Accumulator. The only exception is the routine for the keyboard device (which is the default input device). If the keyboard is the current input device, this routine blinks the cursor, fetches characters from the keyboard buffer, and echoes them to the screen until a carriage return is encountered. When a carriage return is found, the routine sets a flag to indicate the length of the last logical line before the return character, and reads the first character of this logical line from the screen.

Subsequent calls to this routine will cause the next character in the line to be read from the screen and returned in the Accumulator, until the carriage return character is returned to indicate the end of the line. Any call after this character is received will start the whole process over again.

Note that only the last logical line before the carriage return is used. Any time you type in more than 80 characters, a new logical line is started. This routine will ignore any characters on the old logical line, and process only the most recent 80-character group.

61898 $F1CA CHROUT

Output a Byte

This is a documented Kernal routine whose jump table entry point is at 65490 ($FFD2). The routine jumps through a RAM vector at 806 ($326). It is probably one of the best known and most used Kernal routines, because it sends the character in the Accumulator to the current output device. Unless a device has been OPENed and designated as the current output channel using the CHKOUT routine, the character is printed to the screen, which is the default output device. If the cassette is the current device, outputting a byte will only add it to the buffer. No actual transmission of data will occur until the 192-byte buffer is full.

61966 $F20E CHKIN

Designate a Logical File As the Current Input Channel

This is a documented Kernal routine which can be entered from the jump table at 65478 ($FFC6).

The routine jumps through a RAM vector at 798 ($31E). If you wish to get data from any device other than the keyboard, this routine must be called after OPENing the device, before you can get a data byte with the CHRIN or GETIN routine. When called, the routine will designate the logical file whose file number is in the .X register as the current file, its device as the current device, and its secondary address as the current secondary address. If the device on the channel is a serial device, which requires a TALK command and sometimes a secondary address, this routine will send them over the serial bus.

62032 $F250 CHKOUT

Designate a Logical File As the Current Output Channel

This is a documented Kernal routine which can be entered from the jump table at 65481 ($FFC9).

The routine jumps through a RAM vector at 800 ($320). If you wish to output data to any device other than the screen, this routine must be called after OPENing the device, and before you output a data byte with the CHROUT routine. When called, the routine will designate the logical file whose file number is in the .X register as the current file, its device as the current device, and its secondary address as the current secondary address. If the device on the channel uses the serial bus, and therefore requires a LISTEN command and possibly a secondary address, this information will be sent on the bus.

62097 $F291 CLOSE

Close a Logical I/O File

CLOSE is a documented Kernal routine which can be entered via the jump table at 65475 ($FFC3).

The routine jumps through a RAM vector at 796 ($31C). It is used to close a logical file after all I/O operations involving that file have been completed. This is accomplished by loading the Accumulator with the logical file number of the file to be closed, and calling this routine.

Closing an RS-232 file will de-allocate space at the top of memory for the receiving and transmit buffers. Closing a cassette file that was opened for writing will force the last block to be written to cassette, even if it is not a full 192 bytes. Closing a serial bus device will send an UNLISTEN command on the bus. Remember, it is

necessary to properly CLOSE a cassette or disk data file in order to retrieve the file later.

For all types of files, CLOSE removes the file's entry from the tables of logical files, device, and secondary address at 601, 611, and 621 ($259, $263, $26D), and moves all higher entries in the table down one space.

62223 $F30F

Find the File in the Logical File Table

This subroutine is used by many Kernal routines to find the position of the logical file in the logical file table at 601 ($259).

62239 $F31F

Set Current Logical File, Current Device, and Current Secondary Address

This subroutine is used to update the Kernal variables at 184-186 ($B8-$BA) which hold the current logical file number, current device number, and current secondary address number.

62255 $F32F CLALL

Close All Logical I/O Files

CLALL is a documented Kernal routine whose entry point in the jump table is 65511 ($FFE7).

The routine jumps through a RAM vector at 812 ($32C). It closes all open files, by resetting the index into open files at 152 ($98) to zero. It then falls through to the next routine, which restores the default I/O devices.

62259 $F333 CLRCHN

Restore Current Input and Output Devices to the Default Devices

This is a documented Kernal Routine which can be entered at location 65484 ($FFCC) in the jump table.

The routine jumps through a RAM vector at 802 ($322). It sets the current input device to the keyboard, and the current output device to the screen. Also, if the current input device was formerly a serial device, the routine sends it an UNTALK command on the serial bus, and if a serial device was formerly the current output device, the routine sends it an UNLISTEN command.

62282 $F34A OPEN

Open a Logical I/O File

OPEN is a documented Kernal I/O routine. It can be entered from the jump table at 65472 ($FFC0).

The routine jumps through a RAM vector at 794 ($31A). This routine assigns a logical file to a device, so that it can be used for

Input/Output operations. In order to specify the logical file number, the device number, and the secondary address if any, the SETLFS routine must first be called. Likewise, in order to designate the filename, the SETNAM routine must be used first. After these two routines are called, OPEN is then called.

62622 $F49E LOAD

Load RAM from a Device

This is a documented Kernal routine, whose entry in the jump table appears at 65493 ($FFD5).

The routine jumps through a RAM vector at 816 ($330). LOAD is used to transfer data from a device directly to RAM. It can also be used to verify RAM, comparing its contents to those of a disk or tape file. To choose between these operations you must set the Accumulator with a 0 for a LOAD, or a 1 for a VERIFY.

Since the LOAD routine performs an OPEN, it must be preceded by a call to the SETLFS routine to specify the logical file number, device number, and secondary address, and a call to the SETNAM routine to specify the filename (a LOAD from tape can be performed without a filename being specified). Then the .X and .Y registers should be set with the starting address for the load, and the LOAD routine called. If the secondary address specified was a 1, this starting address will be ignored, and the header information will be used to supply the load address. If the secondary address was a 0, the address supplied by the call will be used. In either case, upon return from the routine, the .X and .Y registers will contain the address of the highest RAM location that was loaded.

62885 $F5A5

Print SEARCHING Message If in Direct Mode

62930 $F5D2

Print LOADING or VERIFYING

62941 $F5DD SAVE

Save RAM to a Device

This is a documented Kernal routine, whose entry in the jump table appears at 65496 ($FFD8).

The routine jumps through a RAM vector at 818 ($332). SAVE is used to transfer data directly from RAM to an I/O device. Since the SAVE routine performs an OPEN, it must be preceded by a call to the SETLFS routine to specify the logical file number, device number, and secondary address, and a call to the SETNAM routine to specify the filename (although a SAVE to the cassette can be performed without giving a filename). A Page 0 pointer to the starting

address of the area to be saved should be set up, with the low byte of the address first. The accumulator should be loaded with the Page 0 offset of that pointer, then the .X and .Y registers should be set with the ending address for the save, and the SAVE routine called.

63119 $F68F

If in Direct Mode, Print SAVING and Filename

63131 $F69B UDTIM

Update the Software Clock and Check for the STOP Key

UDTIM is a documented Kernal routine which can be entered through the jump table at 65514 ($FFEA).

It is normally called by the IRQ interrupt handler once every sixtieth of a second. It adds one to the value in the three-byte software jiffy clock at 160-162 ($A0-$A2), and sets the clock back to zero when it reaches the 24-hour point. In addition, it scans the keyboard row in which the STOP key is located, and stores the current value of that key in location 145 ($91). This variable is used by the STOP routine which checks for the STOP key.

63197 $F6DD RDTIM

Read the Time From the Software Clock into the .A, .X, and .Y Registers

This is a documented Kernal routine whose entry point in the jump table is 65502 ($FFDE).

It reads the software clock (which counts sixtieths of a second) into the internal registers. The .Y register contains the most significant byte (from location 160, $A0), the .X register contains the middle byte (from location 161, $A1), and the Accumulator contains the least significant byte (from location 162, $A2).

63204 $F6E4 SETTIM

Set the Software Clock from the .A, .X, and .Y Registers

This documented Kernal routine can be entered from location 65499 ($FFDB).

It performs the reverse operation from UDTIM, storing the value in the .Y register into location 160 ($A0), the .X register into 161 ($A1), and the Accumulator into 162 ($A2). Interrupts are first disabled, to make sure that the clock will not be updated while being set.

63213 $F6ED STOP

Test STOP Key

STOP is a documented Kernal routine which can be entered from the jump table at location 65505 ($FFE1).

It is vectored through RAM at 808 ($328). The routine checks to see if the STOP key was pressed during the last UDTIM call. If it was, the Zero flag is set to 1, the CLRCHN routine is called to set the input and output devices back to the keyboard and screen, and the keyboard queue is emptied.

63227 $F6FB

Set Kernal I/O Error Message

This subroutine is used to handle I/O errors from Kernal I/O routines. It calls CLRCHN to restore default I/O devices. If Bit 6 of the flag at 157 ($9D) is set, it prints I/O ERROR followed by the error number, and then sets the Carry flag to indicate an error, with the error number in the Accumulator. The Kernal error messages are not used by BASIC, but may be used by machine language monitors and other applications.

63276 $F72C

Get Next Tape File Header from Cassette

This routine reads in tape blocks until it finds a file header block. It then prints the FOUND message along with the first 16 characters of the filename.

63338 $F76A

Write Tape File Header Block

63440 $F7D0

Put Pointer to Tape Buffer in .X and .Y Registers

63447 $F7D7

Set I/O Area Start and End Pointers to Tape Buffer Start and End Address

63466 $F7EA

Search Tape for a Filename

63511 $F817

Test Cassette Buttons and Handle Messages for Tape Read

This routine tests the sense switch, and if no buttons are depressed it prints the PRESS PLAY ON TAPE message, and loops until a cassette button is pressed, or until the STOP key is pressed. If a button is pressed, it prints the message OK.

Since the message printing routine is entered after the test for direct mode, these messages cannot be suppressed by changing the flag at 157 ($9D). You could have them harmlessly printed to ROM, however, by changing the value of HIBASE at 648 ($288) temporarily to 160, and then back to 4.

63534 $F82E

Check Cassette Switch

This subroutine is used to check if a button on the recorder has been pressed.

63544 $F838

Test Cassette Buttons and Handle Messages for Tape Write

This routine tests the sense switch, and if no buttons are depressed it prints the PRESS PLAY & RECORD message, and loops until a cassette button is pressed, or until the STOP key is pressed. If a button is pressed, it prints the message OK. These messages cannot be suppressed by changing the flag at 157 ($9D). See the entry for 63511 ($F817) for more information.

63553 $F841

Start Reading a Block of Data from the Cassette

This subroutine tests the cassette switch and initializes various flags for reading a block of data from cassette.

63588 $F864

Start Writing a Block of Data to the Cassette

This subroutine tests the cassette switch and initializes various flags for writing a block of data to cassette.

63605 $F875

Common Code for Reading a Data Block from Tape and Writing a Block to Tape

This routine sets the actual reading or writing of a block of data. It sets CIA #1 Timer B to call the IRQ which drives the actual reading or writing routine, saves the old IRQ vector, and sets the new IRQ vector to the read or write routine. It also blanks the screen so that the video chip's memory addressing (which normally takes away some of the 6510 microprocessor's addressing time) will not interfere with the timing of the routines.

63696 $F8D0

Test the STOP Key during Cassette I/O Operations

This subroutine is used to test the STOP key during tape I/O operations, and to stop I/O if it is pressed.

63714 $F8E2

Adjust CIA #1 Timer A for Tape Bit Timing

63788 $F92C

Read Tape Data (IRQ)

This is the IRQ handler routine that is used for reading data from the cassette. At the end of the read, the IRQ vector is restored to the normal IRQ routine.

64096 $FA60

Receive and Store the Next Character from Cassette

This is the part of the cassette read IRQ routine that actually gets the next byte of data from the cassette.

64398 $FB8E

Move the Tape SAVE/LOAD Address into the Pointer at 172 ($AC)

64407 $FB97

Reset Counters for Reading or Writing a New Byte of Cassette Data

64422 $FBA6

Toggle the Tape Data Output Line

This routine sets CIA #1 Timer B, and toggles the Tape Data Output line on the 6510 on-chip I/O port (Bit 3 of location 1).

64456 $FBC8

Write Data to Cassette—Part 2 (IRQ)

This IRQ handler routine is one part of the write data to cassette routine.

64618 $FC6A

Write Data to Cassette—Part 1 (IRQ)

This IRQ handler routine is the other part of the write data to cassette routine.

64659 $FC93

Restores the Default IRQ Routine

At the end of tape I/O operations, this subroutine is used to turn the screen back on and stop the cassette motor. It then resets CIA #1 Timer A to generate an interrupt every sixtieth of a second, and restores the IRQ vector to point to the normal interrupt routine that updates the software clock and scans the keyboard.

64696 $FCB8

Terminate Cassette I/O

This routine calls the subroutine above and returns from the interrupt.

64714 $FCCA

Turn off the Tape Motor

64721 $FCD1

Check the Tape Read/Write Pointer

This routine compares the current tape read/write address with the ending read/write address.

64731 $FCDB

Advance the Tape Read/Write Pointer

This routine is used to move the pointer to the current read/write address up a byte.

64738 $FCE2

Power-On Reset Routine

This is the RESET routine which is pointed to by the 6510 hardware RESET vector at 65532 ($FFFC).

This routine is automatically executed when the computer is first turned on. First, it sets the Interrupt disable flag, sets the stack pointer, and clears the Decimal mode flag. Next it tests for an autostart cartridge. If one is found, the routine immediately jumps through the cartridge cold start vector at 32768 ($8000). If no cartridge is found, the Kernal initialization routines IOINIT, RAMTAS, RESTOR, and CINT are called, the Interrupt disable flag is cleared, and the BASIC program is entered through the cold start vector at 40960 ($A000).

64770 $FD02

Check for Autostart Cartridge

This routine tests for an autostart cartridge by comparing the characters at locations 32772-6 ($8004-8) to the text below. The Zero flag will be set if they match, and cleared if they don't.

64784 $FD10

Text for Autostart Cartridge Check

The characters stored here must be the fifth through ninth characters in the cartridge in order for it to be started on power-up. These characters are the PETASCII values for CBM, each with the high bit set (+128), and the characters "80".

64789 $FD15 RESTOR

Restore RAM Vectors for Default I/O Routines

This documented Kernal routine can be entered through the jump table at 65418 ($FF8A).

It sets the values for the 16 RAM vectors to the interrupt and important Kernal I/O routines in the table that starts at 788 ($314) to the standard values held in the ROM table at 64816 ($FD30) below.

64794 $FD1A VECTOR

Set the RAM Vector Table from the Table Pointed to by .X and .Y

This documented Kernal routine can be entered through the jump table at 65421 ($FF8D).

It is used to read or change the values for the 16 RAM vectors to the interrupt and important Kernal I/O routines in the table that starts at 788 ($314). If the Carry flag is set when the routine is called, the current value of the 16 vectors will be stored at a table whose address is pointed to by the values in the .X and .Y registers. If the Carry flag is cleared, the RAM vectors will be loaded from the table whose address is pointed to by the .X and .Y registers. Since this routine can change the vectors for the IRQ and NMI interrupts, you might expect that the Interrupt disable flag would be set at its beginning. Such is not the case, however, and therefore it would be wise to execute an SEI before calling it and a CLI afterwards (as the power-on RESET routine does) just to be safe.

64816 $FD30

Table of RAM Vectors to the Default I/O Routines

This table contains the 16 RAM I/O vectors that are moved to 788-819 ($314- $333).

64848 $FD50 RAMTAS

Perform RAM Test and Set Pointers to the Top and Bottom of RAM

This documented Kernal routine, which can be entered through location 65415 ($FF87) of the jump table, performs a number of initialization tasks.

First, it clears Pages 0, 2, and 3 of memory to zeros. Next, it sets the tape buffer pointer to address 828 ($33C), and performs a nondestructive test of RAM from 1024 ($400) up. When it reaches a non-RAM address (presumably the BASIC ROM at 40960, $A000), that address is placed in the top of memory pointer at 643-4 ($283-4). The bottom of memory pointer at 641-2 ($281-2) is set to point to address 2048 ($800), which is the beginning of BASIC program text.

Finally, the pointer to screen memory at 648 ($288) is set to 4, which lets the Operating System know that screen memory starts at 1024 ($400).

64923 $FD9B

Table of IRQ Vectors

This table holds the vectors to the four IRQ routines which the system uses. The first points to Part 1 of the cassette write routine at 64618 ($FC6A), the second to Part 2 of the cassette write routine at 64461 ($FBCD), the third to the standard scan-keyboard IRQ at 59953 ($EA31), and the last to the cassette read routine at 63788 ($F92C).

64931 $FDA3 IOINIT

Initialize CIA I/O Devices

This documented Kernal routine, which can be entered through the jump table at 65412 ($FF84), initializes the Complex Interface Adapter (CIA) devices, and turns the volume of the SID chip off. As part of this initialization, it sets CIA #1 Timer A to cause an IRQ interrupt every sixtieth of a second.

65017 $FDF9 SETNAM

Set Filename Parameters

This is a documented Kernal routine, which can be entered through the jump table at location 65469 ($FFBD).

It puts the value in the Accumulator into the location which stores the number of characters in the filename, and sets the pointer to the address of the ASCII text of the filename from the .X and .Y registers. This sets up the filename for the OPEN, LOAD, or SAVE routine.

65024 $FE00 SETLFS

Set Logical File Number, Device Number, and Secondary Address

This is a documented Kernal routine, which can be entered through the jump table at location 65466 ($FFBA).

It stores the value in the Accumulator in the location which holds the current logical file number, the value in the .X register is put in the location that holds the current device number, and the value in the .Y register is stored in the location that holds the current secondary address. If no secondary address is used, the .Y register should be set to 255 ($FF). It is necessary to set the values of the current file number, device number, and secondary address before you OPEN a file, or LOAD or SAVE.

65031 $FE07 READST

Read the I/O Status Word

This is a documented Kernal routine, which can be entered through the jump table at location 65463 ($FFB7).

Whenever an I/O error occurs, a bit of the Status Word is set to indicate what the problem was. This routine allows you to read the status word (it is returned in the Accumulator). If the device was the RS-232, its status register is read and cleared to zero. For the meanings of the various status codes, see the entry for location 144 ($90) or 663 ($297) for the RS-232 device.

65048 $FE18 SETMSG

Set the Message Control Flag

This documented Kernal routine can be entered through its jump table vector at 65424 ($FF90).

The routine controls the printing of error messages and control messages by the Kernal. If Bit 6 is set to 1 (bit value of 64), Kernal control messages can be printed. These messages include SEARCHING FOR, LOADING, and the like. If Bit 6 is cleared to 0, these messages will not be printed (BASIC will clear this bit when a program is running so that the messages do not appear when I/O is performed from a program). Setting Bit 6 will not suppress the PRESS PLAY ON TAPE or PRESS PLAY & RECORD messages, however.

If Bit 7 is set to 1 (bit value of 128), Kernal error messages can be printed. If Bit 7 is set to 0, those error messages (for example, I/O ERROR #nn) will be suppressed. Note that BASIC has its own set of error messages (such as FILE NOT FOUND ERROR) which it uses in preference to the Kernal's message.

65057 $FE21 SETTMO

Set Time-Out Flag for IEEE Bus

This documented Kernal routine can be entered from the jump table at 65442 ($FFA2).

The routine sets the time-out flag for the IEEE bus. When time-outs are enabled, the Commodore 64 will wait for a device for 64 milliseconds, and if it does not receive a response to its signal it will issue a time-out error. Loading the Accumulator with a value less than 128 and calling this routine will enable time-outs, while using a value over 128 will disable time-outs.

This routine is for use only with the Commodore IEEE add-on card, which at the time of this writing was not yet available.

65061 $FE25 MEMTOP

Read/Set Top of RAM Pointer

This is a documented Kernal routine, which can be entered through the jump table at location 65433 ($FF99).

It can be used to either read or set the top of RAM pointer. If called with the Carry flag set, the address in the pointer will be loaded into the .X and .Y registers. If called with the Carry flag cleared, the pointer will be changed to the address found in the .X and .Y registers.

65076 $FE34 MEMBOT

Read/Set Bottom of RAM Pointer

This is a documented Kernal routine, which can be entered through the jump table at location 65436 ($FF9C).

It can be used to either read or set the bottom of RAM pointer. If called with the Carry flag set, the address in the pointer will be loaded into the .X and .Y registers. If called with the Carry flag cleared, the pointer will be changed to the address found in the .X and .Y registers.

65091 $FE43

NMI Interrupt Entry Point

This routine is the NMI interrupt handler entry, which is pointed to by the hardware NMI vector at 65530 ($FFFA).

Any time an NMI interrupt occurs, the Interrupt disable flag will be set, and the routine will jump through the RAM vector at 792 ($318), which ordinarily points to the continuation of this routine. The standard handler first checks to see if the NMI was caused by the RS-232 device. If not, the RESTORE key is assumed. The routine checks for a cartridge, and if one is found it exits through the cartridge warm start vector at 32770 ($8002). If not, the STOP key is checked, and if it is being pressed, the BRK routine is executed. If the RS-232 device was the cause of the NMI, the cartridge and STOP key checks are bypassed, and the routine skips to the end, where it checks whether it is time to send or receive a data bit via the RS-232 device.

65126 $FE66

BRK, Warm Start Routine

This routine is executed when the STOP/RESTORE combination of keypresses occurs. In addition, it is the default target address of the BRK instruction vector. This routine calls the Kernal initialization routines RESTOR, IOINIT, and part of CINT. It then exits through the BASIC warm start vector at 40962 ($A002).

65138 $FE72

NMI RS-232 Handler

This is the part of the NMI handler that checks if it is time to receive or send a bit on the RS-232 channel, and takes the appropriate action if it is indeed time.

65218 $FEC2

RS-232 Baud Rate Tables for U.S. Television Standard (NTSC)

This table contains the ten prescaler values for the ten standard baud rates implemented by the RS-232 Control Register at 659 ($293). The table starts with the two values needed for the lowest baud rate (50 baud) and finishes with the entries for the highest baud rate, 2400 baud. The RS-232 routines are handled by NMI interrupts which are caused by the timers on CIA #2. Since the RS-232 device could both receive and send a bit in a single cycle, the time between interrupts should be a little less than half of the clock frequency divided by the baud rate. The exact formula used is:

((CLOCK/BAUD)/2)–100

where CLOCK is the processor clock speed and BAUD is the baud rate. The clock frequency for machines using the U.S. television standard (NTSC) is 1,022,730 cycles per second, while the frequency for the European (PAL) standard is 985,250 cycles per second. For this reason, separate baud rate tables were added for European machines at 58604 ($E4EC).

65238 $FED6

RS-232 Receive the Next Bit (NMI)

The NMI handler calls this subroutine to input the next bit on the RS-232 bus. It then calls the next subroutine to reload the timer that causes the interrupts.

65306 $FF1A

Load the Timer with Prescaler Values from the Baud Rate Lookup Table

65352 $FF48

Main IRQ/BRK Interrupt Entry Point

The 6510 hardware IRQ/BRK vector at 65534 ($FFFE) points to this address.

Anytime the BRK instruction is encountered or an IRQ interrupt occurs, this routine will be executed. The routine first saves the .A, .X, and .Y registers on the stack, and then tests the BRK bit of the status register (.P) to see if a BRK was executed. If it was, the routine

exits through the RAM BRK vector at 790 ($316), where it will usually be directed to the BRK routine at 65126 ($FE66). If not, the routine exits through the RAM IRQ vector at 788 ($314), where it will usually be directed to the handler that scans the keyboard at 59953 ($EA31).

If you plan to change either of these vectors to your own routine, remember to pull the stored register values off the stack before finishing.

Location Range: 65371-65407 ($FF5B-$FF7F)

Patches Added to Later Kernal Versions

This area contains additional code not found in the original version of the Kernal. It is used to test whether a European (PAL) or U.S. (NTSC) standard monitor is used, and to compensate so that the sixtieth of a second interrupt will be accurately timed on either system.

65371 $FF5B CINT

Initialize Screen Editor and VIC-Chip

This is a documented Kernal routine whose entry in the jump table is located at 65409 ($FF81).

The start of the routine appears to be a patch that was added to later versions of the Kernal. It first calls the old routine at 58648 ($E518). This initializes the VIC-II chip to the default values, sets the keyboard as the input device and the screen as the output device, initializes the cursor flash variables, builds the screen line link table, clears the screen, and homes the cursor. The new code then checks the VIC interrupt register to see if the conditions for a Raster Compare IRQ have been fulfilled. Since the Raster Register was initialized to 311, that can only occur when using a PAL system (NTSC screens do not have that many scan lines). The PAL/NTSC register at 678 ($2A6) is set on the basis of the outcome of this test. The CIA #1 Timer A is then set to cause an IRQ interrupt every sixtieth of a second, using the prescaler figures for a PAL or NTSC system, as appropriate.

65390 $FF6E

End of Routine to Set Timer for Sixtieth of a Second IRQ

This appears to be a patch added to compensate for the extra length of the current version of this routine, which chooses either the PAL or NTSC prescaler values for the timer.

65408 $FF80

Kernal Version Identifier Byte

This last byte before the jump table can be used to identify the version of the Kernal. The first version has a 170 ($AA) stored here,

the next version 0, and the most current version at the time of this writing has a 3 in this location.

The portable SX-64 has a value of 67 ($43) at this location. The PET 64, a one-piece version with an integrated monochrome display, has an ID byte of 100 ($64). Commodore 64 Logo uses this byte to recognize the PET 64, and adjust its display accordingly.

Location Range: 65409-65525 ($FF81-$FFF5)

Kernal Jump Table

The following jump table is provided by Commodore in an effort to maintain stable entry points for key I/O routines. Each three-byte table entry consists of a 6510 JMP instruction and the actual address of the routine in the ROM. Although the actual address of the routine may vary from machine to machine, or change in later versions of the Kernal, these addresses will stay where they are. By jumping to the entry point provided by this table, rather than directly into the ROM, you insure your programs against changes in the Operating System. In addition, this jump table may help you write programs that will function on more than one Commodore machine. The 15 table entries from 65472-65514 ($FFC0-$FFEA) are the same for all Commodore machines, from the earliest PET on.

As an additional aid, some of these routines are also vectored through the table which starts at 788 ($314). Since this table is in RAM, you can change those vectors to point to your own routines which support additional I/O devices. Programs that use the jump table entry points to the I/O routines will be able to use these I/O devices without a problem.

The following table will give the entry point, routine name, RAM vector if any, its current address, and a brief summary of its function.

65409 ($FF81) CINI (65371, $FF5B) initialize screen editor and video chip
65412 ($FF84) IOINIT (64931, $FDA3) initialize I/O devices
65415 ($FF87) RAMTAS (64848, $FD50) initialize RAM, tape buffer, screen
65418 ($FF8A) RESTOR (64789, $FD15) restore default I/O vectors
65421 ($FF8D) VECTOR (64794, $FD1A) read/set I/O vector table
65424 ($FF90) SETMSG (65048, $FE18) set Kernal message control flag
65427 ($FF93) SECOND (60857, $EDB9) send secondary address after LISTEN

65430 ($FF96) TKSA (60871, $EDC7) send secondary address after TALK
65433 ($FF99) MEMTOP (65061, $FE25) read/set top of memory pointer
65436 ($FF9C) MEMBOT (65076, $FE34) read/set bottom of memory pointer
65439 ($FF9F) SCNKEY (60039, $EA87) scan the keyboard
65442 ($FFA2) SETTMO (65057, $FE21) set time-out flag for IEEE bus
65445 ($FFA5) ACPTR (60947, $EE13) input byte from serial bus
65448 ($FFA8) CIOUT (60893, $EDDD) output byte to serial bus
65451 ($FFAB) UNTLK (60911, $EDEF) command serial bus device to UNTALK
65454 ($FFAE) UNLSN (60926, $EDFE) command serial bus device to UNLISTEN
65457 ($FFB1) LISTEN (60684, $ED0C) command serial bus device to LISTEN
65460 ($FFB4) TALK (60681, $ED09) command serial bus device to TALK
65463 ($FFB7) READST (65031, $FE07) read I/O status word
65466 ($FFBA) SETLFS (65024, $FE00) set logical file parameters
65469 ($FFBD) SETNAM (65017, $FDF9) set filename parameters
65472 ($FFC0) OPEN (VIA 794 ($31A) TO 62282, $F34A) open a logical file
65475 ($FFC3) CLOSE (VIA 796 ($31C) TO 62097, $F291) close a logical file
65478 ($FFC6) CHKIN (VIA 798 ($31E) TO 61966, $F20E) define an input channel
65481 ($FFC9) CHKOUT (VIA 800 ($320) TO 62032, $F250) define an output channel
65484 ($FFCC) CLRCHN (VIA 802 ($322) TO 62259, $F333) restore default devices
65487 ($FFCF) CHRIN (VIA 804 ($324) TO 61783, $F157) input a character
65490 ($FFD2) CHROUT (VIA 806 ($326) TO 61898, $F1CA) output a character

65493 ($FFD5) LOAD (62622, $F49E THROUGH 816, $330) load from a device
65496 ($FFD8) SAVE (62941, $F5DD THROUGH 818, $332) save to a device
65499 ($FFDB) SETTIM (63204, $F6E4) set the software clock
65502 ($FFDE) RDTIM (63197, $F6DD) read the software clock
65505 ($FFE1) STOP (VIA 808 ($328) TO 63213, $F6ED) check the STOP key
65508 ($FFE4) GETIN (VIA 810 ($32A) TO 61758, $F13E) get a character
65511 ($FFE7) CLALL (VIA 812 ($32C) TO 62255, $F32F) close all files
65514 ($FFEA) UDTIM (63131, $F69B) update the software clock
65517 ($FFED) SCREEN (58629, $E505) read number of screen rows and columns
65520 ($FFF0) PLOT (58634, $E50A) read/set position of cursor on screen
65523 ($FFF3) IOBASE (58624, $E500) read base address of I/O devices

Location Range: 65530-65535 ($FFFA-$FFFF)

6510 Hardware Vectors

The last six locations in memory are reserved by the 6510 processor chip for three fixed vectors. These vectors let the chip know at what address to start executing machine language program code when an NMI interrupt occurs, when the computer is turned on, or when an IRQ interrupt or BRK occurs.

65530 $FFFA

Non-Maskable Interrupt Hardware Vector

This vector points to the main NMI routine at 65091 ($FE43).

65532 $FFFC

System Reset (RES) Hardware Vector

This vector points to the power-on routine at 64738 ($FCE2).

65534 $FFFE

Maskable Interrupt Request and Break Hardware Vectors

This vector points to the main IRQ handler routine at 65352 ($FF48).

Chapter 8

GEOS

GEOS

In addition to the standard operating system Kernal in ROM, a new, entirely separate operating system is available for the Commodore 64. This operating system is called *GEOS,* which stands for Graphics Environment Operating System. A disk-based program, *GEOS* can be used on any 64 with a disk drive. The newest model of the Commodore 64, the 64C, comes with a *GEOS* disk. From the user's standpoint, *GEOS* provides a friendly new operating environment, featuring pull-down menus, icons, multiple text fonts, dialog boxes, a screen pointer that can be driven by mouse or joystick, support for a variety of printers, a desktop-type disk management system, and a disk speedup system. For programmers, *GEOS* offers a rich set of standardized routines that allow them to incorporate all of the features listed above into their own programs, as well as providing a complete library of bitmapped graphics, math, string-handling, and fast disk I/O routines.

Not only does *GEOS* offer routines that a program may call the way it does the standard ROM Kernal routines; it offers a framework in which to use these routines as well. Programming a *GEOS* application is very different from what the 64 programmer has come to expect. If a programmer isn't using *GEOS,* he or she is responsible for organizing the entire program structure: figuring out how to get the input from the user, and what to do when that input is received; figuring out the timing of each action, and where each part of the program fits in with every other part. But with *GEOS,* a substantial part of the application's structure is already provided for the programmer.

When *GEOS* loads and runs a program, it calls the program's initialization routine (which is always at memory location 1024 ($400)). After the program has initialized, execution returns once more to *GEOS.* At that point, the only programs that are running are the *GEOS* Main Loop and the *GEOS* Interrupt Handler.

The *GEOS* interrupt code replaces the normal 64 Kernal IRQ handler which updates the software clock, flashes the cursor, and checks the keyboard for input (see the entries for location 59953 ($EA31) on pages 219–20 and location 788–789 ($314–$315) on page 73). Like the 64 Interrupt Handler, this code interrupts the normal program every 1/60 second in order to take care of housekeeping chores (although *GEOS* uses raster interrupts to initiate the

interrupt routines, instead of the timer interrupts used by the old 64 operating system). The *GEOS* version of the handler checks the keyboard and buffers its input, checks the mouse input device for movement and/or button presses, decrements the process timers (more about them later), and moves the mouse pointer and sprites. It also sets system flags that tell the Main Loop whether or not there has been any input.

By checking the input information that it receives from the Interrupt Handler, the Main Loop can do its job, which is to determine if any significant *events* have occurred, and to dispatch the code to handle those events if necessary. For example, if it finds that the mouse button has been clicked, it checks to see if the mouse was positioned over an icon, a menu bar, or a menu item. If it is positioned in such a spot when the button is pressed, an event has occurred—either an icon has been selected, a menu has been opened, or a menu item has been selected. For any of these events, the Main Loop calls a user-supplied subroutine that performs whatever action should be taken when the icon or menu item is selected. If the mouse button is not over an icon or menu, the Main Loop calls a user-supplied routine that handles an otherwise insignificant mouse button click.

In the same way, the Main Loop calls user-supplied routines when other significant events happen, such as a key being pressed on the keyboard, or a process timer counting down to zero. This allows the programmer to create what is called an *event-driven* application. Most programs are structured so that they wait for the user to give them some input, and then react to that input. *GEOS* takes over the jobs of looking for the input and interpreting what it means. As a result, all the programmer has to do is to write the initialization code that sets up menus, or icons, or processes, and the service routines that respond when *GEOS* has determined that the icon or menu has been selected.

Although for many applications, this event-driven scheme will be sufficient, there are some cases where the programmer will want to insert code into either the Main Loop or Interrupt Handler. *GEOS* provides *hooks* for those cases. If you want a routine to be executed before the regular interrupt code, all you have to do is place the address of that routine at the location named interruptTopVector, and end the routine by jumping through the old vector that was stored there. If you want to insert a routine that executes after the regular interrupt handler, you can place the address of that routine at the location named interruptBottomVector. And if you have a subroutine that you want to run as part of the Main Loop, all you have to do is to store its address in the location labeled applicationMain.

For tasks that must be performed periodically, according to a fixed schedule, *GEOS* provides time-based processes. These are sub-

routines which the user can specify to be run after a certain number of interrupts have elapsed. They are set up just by placing the addresses of each subroutine in a table, along with a number indicating how many interrupts to wait before calling that subroutine. Each time the process timer counts down to zero, the subroutine will execute, and the timer will be reset to start the process all over again. *GEOS* also provides a way to suspend process execution.

The Price of *GEOS*

There is, of course, a price for all of the additional features that *GEOS* offers. For one thing, *GEOS* is a RAM-based operating system. Once it is loaded from the disk, it resides in an area of memory that under the old 64 operating system would be available for program code and/or data. *GEOS* reduces the sting of this somewhat by placing most of its code in the less accessible areas of the 64—the RAM that is addressed at the same locations as the BASIC and Kernal ROMs, and I/O space. And, the provision of disk turbo routines makes it easier to use program *overlays,* a scheme whereby the program brings in additional routines from disk files as they are needed. But still, less than 24K of free RAM remains reserved exclusively for the use of application programs under *GEOS.*

Another potential problem area is compatibility with older 64 software. It's highly unlikely that pre-*GEOS* software will confine itself to the area that *GEOS* allocates for application programs. This means that such programs are likely to use some or all of the RAM occupied by *GEOS. GEOS* does its best to accommodate such programs. It provides an *Old C64* icon on the deskTop with the name of the file printed underneath for each file on a non-*GEOS* disk. When the user double-clicks on that icon, *GEOS* checks the load address and file size of the program. If the program does not extend past location 32768 ($8000) in memory, *GEOS* uses its disk turbo fast load routines to load the program. *GEOS* also does its best to automatically run the program once it is loaded. But since non-*GEOS* programs make no provisions for returning to the deskTop, and are likely to overwrite *GEOS* RAM, making such a return impossible, the only way to get back to *GEOS* after running an old 64 application is to reboot the system disk.

GEOS is structured so that to a large extent, the programmer need not be concerned with the details of how every routine works internally. The programmer should not rely on the absolute addresses of system routines except as entered in the jump table, since *GEOS* is not fixed in ROM, and is therefore subject to change. In fact, in the first few months since its introduction, revisions have already been made. So the following memory map is presented with a caution: *Do not rely absolutely on the addresses shown.* Although they seem to be correct for the current version (1.2) of *GEOS,* Berkeley

Softworks has stated that they are likely to change in future versions. For example, it is quite probable that additions will be made to the system jump table at location 49408 ($C100). Keeping this warning in mind, however, you should still find the *GEOS* memory map included in this volume worthwhile. While some of the locations discussed below may change, the *GEOS* programming principles presented in the discussion of those locations should prove helpful no matter what version of *GEOS* you are using.

System Hardware Configuration

Since the 64 hardware can be configured to use RAM and ROM memory in several different ways, it's useful to describe the particular configuration used by *GEOS.* Most of the time, *GEOS* sets location 1 to a value of 48 ($30). This sets up a system with 64K of RAM, instead of the default configuration of 8K BASIC ROM, 8K Kernal ROM, 4K I/O registers, and 44K RAM. Location 56576 ($DD00) determines which of the four 16K banks of memory the VIC II display chip will address. *GEOS* sets this location to select bank 2, at 32768–49151 ($8000–$BFFF). Location 53272 ($D018) determines where the character ROM and the screen memory used by *GEOS'* bitmapped display will be located. The default value used by *GEOS* is 132, which sets the video display RAM to the 8000-byte area starting with location 40960 ($A000), and the Character Dot Data area to 36864 ($9000), where it will not interfere with the bitmapped display.

GEOS Memory Map

Location Range: 2–127 ($2–$7F)

GEOS zero page

Under *GEOS,* this portion of memory is used for *GEOS* zero-page variables, rather than for BASIC zero-page storage as in the normal 64 configuration.

Location Range: 2–33 ($2–$21)

Pseudoregisters r0–r15

Since the 6502 has only three eight-bit registers for the programmer to use, the *GEOS* system uses part of zero-page memory to create a number of pseudoregisters. Though these registers don't have the versatility and speed of real registers, they perform many of the same functions. *GEOS* uses them to pass arguments between system routines, and for general temporary storage. Since these locations are used both by interrupt-level code and the Main Loop, their contents are saved at the beginning of the Interrupt Handler routine, and restored when the routine returns from the interrupt.

2–3	$2–$3	r0
4–5	$4–$5	r1
6–7	$6–$7	r2
8–9	$8–$9	r3
10–11	$A–$B	r4
12–13	$C–$D	r5
14–15	$E–$F	r6
16–17	$10–$11	r7
18–19	$12–$13	r8
20–21	$14–$15	r9
22–23	$16–$17	r10
24–25	$18–$19	r11
26–27	$1A–$1B	r12
28–29	$1C–$1D	r13
30–31	$1E–$1F	r14
32–33	$20–$21	r15

34–35 $22–$23 currentPattern
Pointer to the current pattern used by graphics fill routines

36–37 $24–$25 string
Used for string input

38 $26 baselineOffset
The offset from the top line of each character in the current character set to the baseline

39–40 $27–$28 currentSetWidth
The width of each character card in the current character set in pixels

41 $29 currentHeight
The height of each character card in the current character set in pixels

42–43 $2A–$2B currentIndexTable
Size of each character card in bytes

44–45 $2C–$2D cardDataPointer
A pointer to the location where the actual graphics data for the character is stored

46 $2E currentMode
This location is used to store a flag indicating the current character-drawing mode. Bit 7 is used to indicate whether or not the character is to be underlined. If bit 7 is set to 1, the character is underlined,

and if it is reset to 0, the character is not underlined. Likewise, bit 6 is set if the character is to be in italics, while a 0 in this bit position means that italics will not be used. Bit 5 is set to 1 if the character is to be drawn in reverse video, or to 0 if the character is to be drawn normally.

47 $2F displayBufferOn

The *GEOS* graphics systems allow graphics operations to draw images on the the visible foreground screen, whose data is stored starting at location 40960 ($A000), or on an invisible background screen, whose data area starts at 24576 ($6000). The variable storage location named displayBufferOn is used by the programmer to indicate whether he or she wishes *GEOS* graphics operations to draw to the foreground screen, the background screen, or both. By setting bit 7 to 1, the programmer indicates that subsequent calls to the graphics routines are to write to the foreground screen. By setting bit 6 to 1, the programmer indicates that the graphics should be drawn on the background screen. Both bits may be set at once, in which case graphics operations will affect both foreground and background screens.

48 $30 mouseOn

This location is used for flags that indicate whether the mouse pointer, menus, and/or icons are currently in use. Bit 7 is set to 1 if the mouse pointer is on, and reset to 0 if it is not. Bit 6 is set to 1 if menus are on, and to 0 if they are not. Bit 5 is set to 1 if icons are on, and to 0 if they are not.

49–50 $31–$32 mousePicture

This location holds a pointer to the memory area where the mouse pointer graphics data is stored.

51 $33 windowTop

This variable is used for text clipping. It indicates the top screen line on which characters may be drawn. Any part of a character that appears above this line is clipped, and will not be drawn.

52 $34 windowBottom

The bottom line of the window for purposes of text clipping

53–54 $35–$36 leftMargin

The leftmost pixel of the window for purposes of text clipping

The carriage-return character sends the cursor back to this point.

55–56 $37–$38 rightMargin

The rightmost pixel to which text characters may be written
If a text string crosses this point, the routine pointed to by StringFaultVector is called.

57 $39 pressFlag

This location is used for a flag indicating whether the last keypress was a new key, or a repeat of the previous keypress. Bit 7 is set to 1 if the keypress was a new one, and reset to 0 if it was a repeat. Bits 6 and 5 are used in similar fashion to designate whether disk data (bit 6) or mouse button data (bit 5) is new. There is also a bit that indicates whether the device-specific information in inputData is new.

58–59 $3A–$3B mouseXPosition

The current horizontal screen position (0–319) of the mouse pointer

60 $3C mouseYPosition

The current vertical position (0–199) of the mouse pointer

61–62 $3D–$3E returnAddress

Several of the *GEOS* routines have *inline* versions. This means that the data that must be passed to these routines is imbedded in the code immediately following the JSR instruction. Since these routines must return to a point several bytes in advance of the normal return address, this location is used by those inline routines to designate the correct return address.

Location Range: 251–255 ($FB–$FF)

GEOS **user zero-page RAM**
GEOS leaves these four zero-page locations free for use by the application, as is the case with the normal 64 configuration. *GEOS* assigns locations 251–252 ($FB–$FC) the pseudoregister name a0, and locations 253–254 ($FD–$FE) the name a1.

Location Range: 1024–24575 ($400–$5FFF)

APPLICATION_RAM
GEOS **application program and data storage**
This block of memory represents most of the free RAM that *GEOS* leaves for application programs. When *GEOS* loads an application, it expects that the program's initialization code (or a JMP instruction to that code) will be located at address 1024 ($400). This initialization code should set up the screen graphics, the menus, and the icons, as well as processes and user vectors (see location range 33946–33981, $849A–$84BD) if necessary.

Although the program area left for *GEOS* applications is somewhat limited, you may find that the disk turbo routines compensate for this to some degree by allowing you to quickly swap portions of data or program code that are stored on disk in and out of the application memory area.

Location Range: 24576–32575 ($6000–$7F3F)

BACK_SCREEN_BASE
Background Screen
Besides the normal screen display data area at 40960 ($A000), *GEOS* also sets aside this 8000-byte area as a second, background screen data area. This screen buffer is used by the system as a storage area for saving application images before they are overwritten by the system's menus or dialog boxes. There are a number of *GEOS* graphics routines that make it easy to move blocks of image data between the foreground and background screen areas, and these routines are used to recover portions of the screen that are overwritten. Plus, the user can take advantage of the background screen area, using it for an *undo* buffer. There are also routines that support moving objects from one screen to the other, and these can be used to aid animation.

GEOS uses the contents of location displayBufferOn as a flag which determines whether drawing operations affect the foreground screen only, the background screen only, or both. If you do not display menus or dialog boxes in your program, you may turn off the graphics buffer and use this area for additional storage or code space. For the details on the use of displayBufferOn, see location 47 ($2F).

Location Range: 30976–32575 ($7900–$7F3F)

PRINTBASE
Load address for printer drivers
The bottom part of the background screen buffer area can also be used to store the printer driver software. This is the program that allows an application to print out a copy of the bitmapped screen using a dot-matrix printer. Since *GEOS* stores all text as bitmapped graphics, character-only models such as daisywheel printers are not suitable for use with *GEOS* programs unless the application itself undertakes to support such printers.

GEOS comes with a wide assortment of printer drivers that support Commodore's serial printers, such as the Commodore 801 and 1000 printers, as well as many popular parallel printers, such as Epson and Epson compatibles, Star SG-10, Okidata 92, and the C. Itoh Prowriter.

In addition, it is possible for the programmer to write printer driver code for other printers. The basic requirements are that drivers

must load in at location 30976 ($7900) and must be no more than 1792 bytes long. The application must allocate a 640-byte print buffer, to which it sends data to be printed, as well as a 640-byte work area which the printer driver uses to assemble the data into the format in which the printer wants it. Printer drivers must support several functions, each of which may be entered by a jump table which is located at the beginning of the driver. This jump table is organized like the 64 Kernal jump table (see locations 65409–65525, pages 243–45), with each entry consisting of the single-byte 6510 JMP opcode followed by the two-byte address of the routine. The functions themselves are explained below.

30976 $7900 InitForPrint

This routine gives the printer driver the opportunity to pass to the printer any special setup codes that might be required before starting to print. If the printer does not require any such codes, this routine does nothing except return control to the calling routine.

30979 $7903 StartPrint

StartPrint opens the printer device using the Commodore Kernal routines, and sends the codes necessary to put the printer into graphics mode. It first calls the *GEOS* SetDevice to let it know that the printer (device number 4) is being used. Next, the *GEOS* InitForIO routine is called to disable interrupts and bank in the Kernal ROM and I/O space in place of the RAM where the *GEOS* code resides. From then on, the normal 64 Kernal routines are used to open the printer device, and send it the codes to put the printer into graphics mode. Finally, the *GEOS* DoneWithIO routine is called to enable interrupts and bank out the I/O space and Kernal ROM.

30982 $7906 PrintBuffer

PrintBuffer actually does the printing. Before calling the routine, the application must put the data for one row of 80 cards into the print buffer pointed to by r0, and must pass the address of a 640-byte workspace in r1, whose location remains constant between calls to PrintBuffer. Note that the print buffer has space for a print row 80 cards wide, which means that it can hold up to 640 pixels across. Since this is twice as wide as the actual 64 screen itself, it is possible to print pictures that are larger than the physical screen (*geoPaint* can create such pictures, for example). Since each row must be 80 cards wide, it is up to the application to pad the beginning or end of the line with blanks if there is not enough screen data to fill up the buffer.

The Print Buffer routine takes the data in the buffer, converts it to a format that the printer can use, and sends as much of it to the

printer as the printer can fit on the page (see GetDimensions, below). The routines used to send this data to the printer are much the same as the ones used by StartPrint, above: SetDevice; InitForIO; then the Kernal routines to Listen the printer, send it data, and Unlisten it; followed by DoneWithIO.

30985 $7909 StopPrint

StopPrint is called at the end of the page to send a form-feed character to the printer, and to flush the work buffer. Before calling it, the application must put the address of the print buffer in r0, and the address of the work buffer in r1.

30988 $790C GetDimensions

This routine returns the number of cards of screen data that the printer can fit onto an 8 × 10.5–inch area of the paper, where a card is defined as a block eight pixels high and eight pixels wide. Upon return, the X register holds the maximum print width in cards, and the Y register holds the maximum print height in card rows.

30991 $790F PrintASCII

This routine is used to send an ASCII character to the printer in text character mode, not graphics mode.

Location Range: 32576–32767 ($7F40–$7FFF)

APPLICATION_VAR

Application-free RAM

The applications programmer may use the 192 bytes at the end of the background screen area as additional variable storage space.

Location Range: 32768–35327 ($8000–$89FF)

OS_VARS

***GEOS* Operating System RAM variable storage and work area**

This area includes space for disk buffers and storage of general information about disk operations, such as disk names, filenames, disk buffers, the number and types of drives available, and so on. It also includes the important vector table that is used to link the application code with the *GEOS* Main Loop and Interrupt Handler programs. Finally, there are a host of system variables used to maintain the mouse pointer, icon and menu selection imagery, the alarm clock, and other system information.

32768–33023 $8000–$80FF diskBlkBuf

General disk block buffer

33024–33279 $8100–$81FF fileHeader
Used to hold the header block for a *GEOS* file
The header block contains the icon shape data information, the Commodore file type, the *GEOS* file type, the starting and ending address data contained in the file, the initialization address, a permanent identifier, the author's name, and if the file is a data file, the name of the application file to which the data belongs.

33280–33535 $8200–$82FF curDirHead
Block containing directory header information for the disk in the currently selected drive

33536–33791 $8300–$83FF fileTrScTab
Buffer used to hold the track and sector chain for a file whose maximum size is 32,258 bytes. In order to save a file larger than this, use the partial-file read and write routines, and a special track and sector table of sufficient size.

33792–33821 $8400–$841D dirEntryBuf
Buffer used to build a file's directory entry

33822–33839 $841E–$842F DrACurDkNm
Disk name of disk in drive A
The first 16 characters are used for the name, the last 2 for the disk ID.

33840–33857 $8430–$8441 DrBCurDkNm
Disk name of disk in drive B
The first 16 characters are used for the name, the last 2 for the disk ID.

33858–33874 $8442–$8452 dataFileName
Name of the data file that is used to start an application
This name is passed to the application.

33875–33892 $8453–$8464 dataDiskName
The name of the disk on which the data file used to start the application is located

33893–33909 $8465–$8475 PrntFilename
Name of the current printer driver
The first 16 characters are used for the filename, the last character for a string terminator.

33910–33928 $8476–$8488 PrntDiskName
The name of the disk on which the current printer driver is located
The name of the disk is followed by a string terminator byte.

33929 $8489 curDrive
The drive number (8-11) of the disk drive that is currently active

33930 $848A diskOpenFlg
Flag used to indicate whether any files are currently open on a disk

33931 $848B isGEOS
Flag used to indicate whether or not the current disk is a *GEOS* disk

33932 $848C interleave
The BlkAlloc routine uses the value here as the desired interleave when selecting free blocks for a disk chain. *GEOS* sets this value to 8.

33933 $848D numDrives
The number of drives recognized by *GEOS*

33934–33937 $848E–$8491 driveType
This block is reserved for flags showing the disk drive types, but these flags are not yet implemented. This means the system does not yet distinguish between, say, a 1541 drive and a 1571 drive.

33938–33941 $8492–$8495 turboFlags
This block is set aside for turbo flags for each of drives 8, 9, 10, and 11, though currently only the first two are actually used. These flags indicate whether the drives are currently in turbo mode or normal (slow) mode.

Location Range: 33942–33946 ($8496–$849A

VLIR file data
The *GEOS* system maintains a new type of disk file called VLIR (Variable Length Indexed Record). This type of file is similar to Commodore's relative files.

33942 $8496 curRecord
The current record number for the open VLIR file

33943 $8497 usedRecords
The number of records in the the open VLIR file

33944 $8498 fileWritten
This flag indicates whether or not the file has been written to since the last update of the index table and the Block Allocation Map (BAM).

33945–33946 $8499–$849A fileSize

The current size (in blocks) of the VLIR file

This value is stored in the directory entry of the file.

Location Range: 33947–33982 ($849B–$84BE)

***GEOS* dispatch vector table**

This table contains the addresses of several important *GEOS* routines. During the course of its execution, *GEOS* will jump indirectly through these vectors in order to call the routines. If the value for an entry in the table is 0, however, *GEOS* will skip that call. In most cases, *GEOS* pushes the return address on the stack before making the jump, which means that the user should end these routines with an RTS instruction. The exception to this is interruptTopVector, which points to the address interruptMain. If you wish to divert this vector to your own use, you should preserve the address of interruptMain somewhere first, and end your routine by jumping through that saved vector.

33947–33948 $849B–$849C applicationMain

The address of the application's own main loop subroutine

If the application inserts an address here during its initialization, the *GEOS* Main Loop will execute the subroutine at the end of each pass through the loop. The application's routine should end with an RTS instruction.

33949–33950 $849D–$849E interruptTopVector

Vector which holds the address of the main *GEOS* interrupt routine, interruptMain

If you wish to perform some interrupt routine before *GEOS* reads the input devices and updates sprite positions, you must save the address found here and substitute the address of your own subroutine. You may, for example, wish to use other IRQ interrupts than the raster interrupt used by *GEOS* to check the keyboard and mouse. In such a case, the Interrupt Handler would have to check the source of each IRQ as soon it happened. This would allow the handler to determine whether the interrupt was caused by the *GEOS* raster IRQ and should result in the execution of the *GEOS* interrupt routine, or whether it was caused by the user's own interrupt, which should result in the execution of that interrupt routine. In order to execute the normal *GEOS* interrupt code at the end of the routine, you should jump to the address of interruptMain, which you have saved.

33951–33952 $849F–$84A0 interruptBottomVector

If you wish to perform some interrupt-level code after *GEOS* has checked the hardware and set its flags, put the address of that subroutine here. *GEOS* will execute the subroutine whose address is here at the end of its own interrupt routine, unless this vector is set to 0, in which case it skips right to the end of the interrupt. You should end your routine with an RTS instruction, not an RTI.

33953–33954 $84A1–$84A2 mouseVector

The address of the routine that is called when the mouse button is pressed

33955–33956 $84A3–$84A4 keyVector

Vector which holds the address of the user-supplied routine to call when a key is pressed

This routine can look in keyData (34063/$850F) to find the value of the key. This routine will not be run if the key is pressed when some other *GEOS* routine that handles keyboard input is already running. If you don't want to supply a routine to handle keypresses, you may leave this vector set to zero. If *GEOS* finds a zero here, it will not take any immediate action when a key is pressed, though it will still buffer the character in keyData.

33957–33958 $84A5–$84A6 inputVector

The address of a user-supplied routine that the *GEOS* Main Loop calls when the input device bit of pressFlag is set, indicating a change in inputData

33959–33960 $84A7–$84A8 mouseFaultVector

Vector which holds the address of the routine to call when the user tries to move the mouse pointer outside the boundaries set by mouseLeft, mouseRight, mouseTop, and mouseBottom

33961–33962 $84A9–$84AA otherPressVector

The address of the user-supplied routine that is called when the mouse button is clicked and the mouse pointer is not positioned over a menu or an icon. If the user leaves this value set to zero, *GEOS* will ignore mouse button presses that do not activate menus or icons.

33963–33964 $84AB–$84AC StringFaultVector

The address of the user-supplied routine that is called when the next character written by a routine like PutString or PutCharacter would be placed on the screen to the right of the value in rightMargin. This routine may advance the text to the next line after checking bottomMargin to make sure that the bottom of the page hasn't been

reached. Or, it can be used to implement fancier features like word-wrapping.

33965–33966 $84AD–$84AE alarmTmtVector
The address of the user-supplied routine that is called when the alarm clock timer counts down to zero. If the user leaves this value set to zero, *GEOS* will do nothing when the alarm goes off.

33967–33968 $84AF–$84B0 BRKVector
This vector points to the routine that is called when the BRK instruction is executed. The routine that normally is called is Panic, which puts up a dialog box with the address of the BRK instruction, and allows the program to die with dignity.

33969–33970 $84B1–$84B2 RecoverVector
The address of the routine which is called in order to recover the part of the screen that has been erased by menus or dialog boxes.

33971 $84B3 selectionFlash
This variable is used to determine the speed at which menu items and icons are flashed when selected.

33972 $84B4 alphaFlag
This variable is used during string input to flag alphanumeric characters.

33973 $84B5 iconSelFlag
Bits 6 and 7 of this flag are used to indicate how *GEOS* should highlight an icon that is selected. If no bits are set, the icon is not highlighted, and the service routine is called immediately. If bit 6 (the ST_FLASH flag) is set, the icon is flashed at the rate specified by selectionFlash, above. If bit 7 (the ST_INVERT flag) is set, the icon's image is inverted when the service routine is called. If both bits are set, the ST_FLASH flag takes precedence, and the icon is flashed, but not inverted.

33974 $84B6 faultData
Flags used to determine the current mouse faults

33975 $84B7 menuNumber
The number of the active menu

33976 $84B8 mouseTop
The top position to which the mouse pointer may travel

33977 $84B9 mouseBottom
The bottom position to which the mouse pointer may travel

33978–33979 $84BA–$84BB mouseLeft
The leftmost position to which the mouse pointer may travel

33980–33981 $84BC–$84BD mouseRight
The rightmost position to which the mouse pointer may travel

33982–33983 $84BE–$84BF stringX
The horizontal position for the current string input

33984 $84C0 stringY
The vertical position for the current string input

33985–34048 $84C1–$8500 mousePicData
The place where the actual sprite data for the mouse pointer picture is stored

34049 $8501 maximumMouseSpeed
The maximum speed at which the mouse pointer may travel
The joystick speed may be a value from 0 to 127.

34050 $8502 minimumMouseSpeed
The minimum speed at which the mouse pointer may travel
The joystick speed may be a value from 0 to 127.

34051 $8503 mouseAcceleration
The rate at which the mouse speed can accelerate to its maximum speed.

34052 $8504 keyData
Buffer that stores the last key pressed
The keyVector and other keyboard service routines should get the keypress data from here.

34053 $8505 mouseData
The place where data for the mouse input device service routines is stored
This number is negative when the mouse button is pressed, and nonnegative when it is released.

34054–34057 $8506–$8509 inputData
A four-byte area where the UpdateMouse routine can store device-specific information that can be used by application routines. When the joystick is the input device, the first byte of this value indicates

one of eight stick directions. Starting with 0 at the right-hand position, the values go counterclockwise, so that 1 represents right and up, 2 represents up, 3 up and left, and so on. If the joystick is centered, the value here reads −1. The second byte of this area is used to store the current mouseSpeed for the joystick input device.

34058–34059 $850A–$850B random
Holds a random number that is incremented during each interrupt

34060–34068 $850C–$8514 saveFontTab
GEOS saves the user's active font table here when going into menu display mode.

34069 $8515 dblClickCount
Variable used to determine when the user has double-clicked on an icon
It is loaded with a value when the mouse button is pressed while the pointer is over an icon, and this value is decremented during each interrupt. If it has not counted down to zero when the icon is selected again, then the double-click flag in the high byte of pseudoregister r0 is set to TRUE when the service routine is called. Otherwise, r0 is set to FALSE.

34070 $8516 year
The year value for the time of day clock

34071 $8517 month
The month value for the time of day clock

34072 $8518 day
The day value for the time of day clock

34073 $8519 hour
The hour value for the time of day clock

34074 $851A minutes
The minutes value for the time of day clock

34075 $851B seconds
The seconds value for the time of day clock

34076 $851C alarmSetFlag
Variable used to implement the alarm
If an alarm is set which *GEOS* should monitor, this value is set to 255 ($FF). If no alarm is set, the value stored here should be zero.

34077 $851D sysDBData
Variable used internally to indicate which command terminated a dialog, causing a return to the application

34078 $851E screencolors
The default screen colors

34079–34495 $851F–$86BF dlgBoxRamBuf
Area used to save all of the system RAM variables that define the state of an application
These include zero-page global variables, non–zero-page global variables, and information about menus, processes, and sprites.

Location Range: 35328–35839 ($8A00–$8BFF)

SPRITE_PICS
Sprite image data

35328–35391 $8A00–$8A3F spr0pic
35392–35455 $8A40–$8A7F spr1pic
35456–35519 $8A80–$8ABF spr2pic
35520–35583 $8AC0–$8AFF spr3pic
35584–35647 $8B00–$8B3F spr4pic
35648–35711 $8B40–$8B7F spr5pic
35712–35775 $8B80–$8BBF spr6pic
35776–35839 $8BC0–$8BFF spr7pic

Location Range: 35840–36823 ($8C00–$8FD7)

COLOR_MATRIX
Video color matrix

36824–36855 $8FD8–$8FF7
GEOS **free RAM**

Location Range: 36856–36863 ($8FF8–$8FFF)

Sprite shape data pointers
(For more information, see location 2040, page 82.)

Location Range: 40969–48959 ($A000–$BF3F)

SCREEN_BASE
Foreground screen RAM
GEOS uses the 64's high-resolution bitmap graphics mode for its display. (See the section about bit 5 of location 53265, pages 131–34, for more information on this mode.) The 16K VIC-II display space is set to bank 2 (32768–49151/$8000–$BFFF), and the top half of that bank is set aside for display data. The contents of the memory in this area determine whether any given dot on the screen is lit up or set to the background color, depending on whether the bit that represents that pixel is set to 1 or 0.

Besides the normal display area, *GEOS* maintains a second display buffer starting at location 24576 ($6000). This buffer holds a duplicate copy of the display data, from which the normal screen display may be recovered after it has been erased by a dialog box or menu.

Location Range: 49860–49151 ($BF40–$BFFF)

***GEOS* tables**

Location Range: 49152–53247 ($C000–$CFFF)

OS_ROM
4K *GEOS* code, always resident

49152 $C000 BootGEOS

The jump vector to the short program used to reboot *GEOS*
This vector is used to restart *GEOS* by having the user insert the *GEOS* system disk, as long as the program that calls it has kept the memory area from $C000–$C02F intact.

49155 $C003 ResetHandle

The jump vector to the code that initializes *GEOS* from a cold start

Location Range: 49408–49863 ($C100–$C2C7)

OS_JUMPTAB
***GEOS* Operating System jump table**
Just as Commodore maintains a jump table of Kernal routines in order to provide a fixed entry point for certain useful I/O functions, *GEOS* also maintains a jump table of important *GEOS* routines. This

table is organized in exactly the same way as the ROM Kernal jump table that starts at location 65409 ($FF81). Each entry in the table is three bytes long, and consists of the 6510 JMP opcode ($4C), followed by the actual address of the routine, in the normal low-byte, high-byte order.

While Commodore includes only a little more than three dozen such routines, *GEOS* provides over 150. Because of the number of routines, it is not possible to explain all of them here in detail. A brief description of most of the routines is provided to aid the curious in their experimentation.

49408 $C100 InterruptMain

The main portion of the normal IRQ interrupt handler

Users may insert their own routines into the interrupt-handler chain either before or after this routine executes. To insert code in front of this routine, place the address of your interrupt code in the interruptTopVector (33949/$849D), and end the routine by jumping here, with a JMP $C100 instruction. To insert a user interrupt routine that executes after this one, place the address of the routine in interruptBottomVector (33951/$849F), and end it with an RTS instruction.

Processes

49411 $C103 InitProcesses

Initializes all processes in the process table

The processes do not start running, but rather are marked frozen and blocked. The accumulator should contain the total number of processes, and r0 should point to the process table.

The process table is a data structure that describes a number of timed processes. These are subroutines that are run periodically, at fixed time intervals. They can be used for any short task that must be performed on a regular basis. When a process is initialized, *GEOS* assigns it a counter and a flag. The *GEOS* Interrupt Handler subtracts 1 from the counter value during each interrupt (every 1/60 second). When the counter reaches 0, a flag is set to show that the process should be run, and the counter is reset to its original value. The *GEOS* Main Loop checks the flags for each process every time that the Loop runs. If a process flag shows that it should run, the Main Loop calls the subroutine associated with that process.

The process table contains a four-byte entry for each process. The first two bytes give the address of the process subroutine. The next two bytes indicate the interval at which the process is to run, in low-byte, high-byte format. This interval is expressed in the number of interrupts which pass before the process runs. These interrupts occur every 1/60 second. Therefore, a value of 60 indicates that the process should run once per second, and a value of 600 means that the process runs once every ten seconds.

49414 $C106 RestartProcess

Used to start a process running

This routine marks the process whose number is stored in the X register as unfrozen and unblocked, and resets the process timer to start the process running.

49417 $C109 EnableProcess

EnableProcess is used to force the process whose number is stored in the X register to be run during the next Main Loop. If the process is blocked or frozen it remains so, and if it is running it is executed once, and its timer is reset.

49420 $C10C BlockProcess

This routine sets a flag that blocks the execution of the process whose number is stored in the X register. The process timer continues to count down and reset as normal, but the process is not marked for execution each time that the counter reaches zero.

49423 $C10F UnblockProcess

Unblocks the execution of the process whose number is stored in the X register

49426 $C112 FreezeProcess

Used to stop the timer of the process whose number is stored in the X register

Effectively halts execution of the process, since its timer never counts down to zero.

49429 $C115 UnfreezeProcess

Restarts the timer for the process indicated by the contents of the X register, allowing it to count down to zero and execute

Graphics

49432 $C118 HorizontalLine

Draws a horizontal line from r3,r11L to r4,r11L, using the eight-bit line pattern in the accumulator

49435 $C11B InvertLine

Inverts all of the bits in the horizontal line between points r3,r11L and r4,r11L

49438 $C11E RecoverLine

Copies the horizontal line between points r3,r11L and r4,r11L on the background screen to the same position on the foreground screen

49441 $C121 VerticalLine
Draws a vertical line from r4,r3L to r4,r3H using the eight-bit line pattern found in the accumulator

49444 $C124 Rectangle
Draws a rectangle from r3,r2L to r4,r3H, and fills it with the current fill pattern (set by SetPattern)

49447 $C127 FrameRectangle
Draws the outline of a rectangle from r3,r2L to r4,r3H
The outline is one pixel wide, and conforms to the pattern set by the current fill pattern (set by SetPattern).

49450 $C12A InvertRectangle
Inverts all of the pixels within the rectangle whose top left corner is defined by the point r3,r2L and whose bottom right corner is the point r4,r3H

49453 $C12D RecoverRectangle
Moves the rectangle bordered by r3,r2L to r4,r3H from the background screen to the foreground screen

49456 $C130 DrawLine
Draws a line from r3,r11L to r4,r11H
If the sign flag of the status register (.P) is set, this routine copies the designated line from the background screen to the foreground screen; otherwise it draws a new line. If the carry flag of the status register is set, the routine draws the line using the foreground color (using 1 bits); otherwise it draws it in the background color (using 0 bits).

49459 $C133 DrawPoint
Draws the point r3,r11L
If the sign flag of the status register (.P) is set, this routine copies the designated point from the background screen to the foreground screen; otherwise it draws a new point. If the carry flag of the status register is set, the routine draws the point using the foreground color (a 1 bit); otherwise it draws it in the background color (using a 0 bit).

49462 $C136 GraphicsString
Executes the string of graphics commands pointed to by r0
The commands consist of an opcode followed by data:

Opcode name	Number	Data	Function
MOVEPENTO	1	*x,y* (.word,.byte)	Move pen to *x,y*
LINETO	2	*x,y* (.word,.byte)	Draw line to *x,y*

Opcode name	Number	Data	Function
RECTANGLETO	3	*x,y* (.word,.byte)	Draw rectangle to *x,y*
PENFILL	4	NONE	Fill with current pattern
NEWPATTERN	5	PatternNumber (.byte)	Change current pattern
ESC_PUTSTRING	6	NONE	Interpret rest of string as a putstring command
FRAME_RECTO	7	*x,y* (.word,.byte)	Frame a rectangle to *x,y*
ESC_BITMAP	8	NONE	Interpret rest of string as bitmap object

Use 0 in the command position to end the string.

49465 $C139 SetPattern

Sets the area-fill pattern to one of 34 preset eight-bit × eight-bit patterns (0-33).

This pattern is used by all drawing routines that fill an area when they draw, such as Rectangle.

49468 $C13C GetScanLine

49471 $C13F TestPoint

Returns the color of point r3,r11L in the carry flag

If the bit is set to foreground color, the carry flag is set. If the bit is reset to background color, the carry flag is reset to zero.

Background Generation

49474 $C142 BitmapUp

Displays the compacted *GEOS* bitmap objects pointed to by r0 at the horizontal byte boundary (0-39) specified in r1L and the scan line designated by r1H

The width of the object in bytes (0–39) is passed in r2L, and its height in scan lines is passed in r2H.

The data for *GEOS* bitmap objects is arranged more like that of sprites than that of normal bitmap graphics. Each data byte appears on screen to the right of the previous one until an entire row is completed, and then a new row is started directly underneath the old one. *GEOS* also uses a compaction scheme called *run-length encoding* to reduce the size of the data required for the bitmap object. Under this scheme, the graphics data is treated like one long string of bytes, without regard to line boundaries. This string of bytes is compacted

by breaking it down into Count/Bitmap groups. Each Count/Bitmap group consists of a Count byte followed by one or more data bytes. If the Count byte has a value of 127 or less, the following Bitmap data byte is repeated Count number of times. For example, a Count byte of 3 followed by a Bitmap data byte of 255 means that 3 bytes with the value 255 (a solid line) are drawn in a row. When this notation is used, only two bytes of data are required to depict an image that is three bytes long. Of course, a large part of most images is made up of nonrepeating data. If the Count byte has a value of 128–220, the next Count − 128 bytes of nonrepeating data are drawn one byte at a time.

Sometimes you can find repeating patterns that are more than one byte long. For example, there may be a dotted-line pattern such as 255, 0, 0, 255, 0 that is repeated ten times in a row. For such cases, the *GEOS* bitmap format uses a Count byte of 221 to 255, followed by another special Count byte called Bigcount, followed by the data for the repeating pattern. The number of bytes in the repeating pattern equals Count − 220. Bigcount indicates the number of times that the pattern is repeated. The actual data follows the Bigcount byte. For the example mentioned above, you would code the data with a Count byte of 225 to show that there are five bytes in the pattern, followed by a Bigcount byte of 10 to show that the pattern is repeated ten times, followed by the five bytes of the pattern. This seven-byte sequence can be used to replace a repeating pattern that is 50 bytes long.

The following table summarizes the three different types of Count bytes, and the data that follows these bytes:

Count	Followed by
0–127	A single data byte to be repeated Count times
128–220	Count − 128 bytes of nonrepeating data
221–255	A Bigcount byte (equal to number of times to repeat pattern)

A repeating pattern of Count − 220 data bytes

A single Bitmap Object can be made up of several Count/Bitmap groups. The three compression schemes can be mixed in any order.

Character Manipulation

49477 $C145 PutChar

Prints the ASCII character in the accumulator at position r11, r1H

This command also supports a number of nonprinting command characters which are described under PutString, below. Note that commands that require data after the command character, such as the GOTO and NEWCARDSET commands, are not supported.

49480 $C148 PutString

Prints a string of characters that ends with a zero byte, the address of which is pointed to by r0, at position r11, r1H.
Besides the normal printing ASCII characters, PutString supports a number of nonprinting command characters as well:

Command	ASCII	FUNCTION
BACKSPACE	8	Erase last character (only once in a row)
FORWARDSPACE	9	Advance one space
LINE FEED	10	Move down one character
HOME	11	Move to upper left corner
UPLINE	12	Move up one character
CR (RETURN)	13	Move to left margin and down a character
UNDERLINEON	14	Turn underlining on
UNDERLINEOFF	15	Turn underlining off
REVERSEON	18	Turn reverse video on
REVERSEOFF	19	Turn reverse video off
GOTOX	20	Move to the horizontal position specified by the two data bytes that appear after this character
GOTOY	21	Move to the vertical position specified by the data byte that appears after this character
GOTOXY	22	Move to the horizontal position specified by the two data bytes that appear after this character, and to the vertical position specified by the data byte after that
NEWCARDSET	23	Ignore next three bytes (can't change font)
BOLDON	24	Turn bold print on
ITALICON	25	Turn italic print on
PLAINTEXT	25	Restore plain text

49483 $C14B UseSystemFont

Routine which instructs *GEOS* to use the resident system font, BSW-9

Mouse and Menus

49486 $C14E StartMouseMode

Starts the mouse functioning by activating mouseVector and calling InitMouse
If the carry flag is set when this routine is called, slowMouse is called, and the *x* and *y* positions of the mouse pointer are set to the values in r11 and the Y register, respectively.

49489 $C151 DoMenu

Used to display and activate the menu structure whose menu definition table is pointed to by r0

The cursor is placed on the menu item specified by the accumulator. If an application program uses menus (and *GEOS* expects that it will), the program initialization code at 1024 ($400) should contain a call to DoMenu.

The key to setting up a menu is constructing the menu definition table. This is a data structure that fully describes the way that the menu should be drawn and the action to be taken when a menu item is selected with the mouse. Each menu starts with a seven-byte header that is laid out as follows:

```
.byte TOP          ;y coordinate of top of menu box
.byte BOTTOM       ;y coordinate of bottom of menu box
.word LEFT         ;x coordinate of left side of menu box
.word RIGHT        ;x coordinate of right side of menu box
.byte MENU_TYPE ORed with NUMBER_OF_ITEMS
```

The first six bytes specify the size of the box that is drawn around the menu. If the menu is a horizontal one (like the menu bar at the top of the screen), *GEOS* draws one long horizontal menu box and prints all of the menu items on the same line, separating each of them with a vertical bar character. The top of the menu box should be placed near the top of the screen, and the bottom should be placed 14 pixels lower to allow for one row of text using the standard character set. The left and right margins depend on how many menu titles are included, but you should be sure to leave enough room for all of the text. If the menu is a vertical one (like the ones that drop down from the top menu bars), you should place the top vertical coordinate on the same line as the bottom row of the menu title bar, and place the bottom vertical coordinate so that there are 14 pixels of height for each item on the menu, plus 1 additional pixel. The left and right sides should be chosen to allow room for the menu items whose text is longest.

The last byte in the header is a flag made up of two parts. The first part specifies the orientation of the menu, and whether or not the mouse pointer is to be constrained to the area of the menu. If this byte has bit 8 set (indicated by a value of 128/$80), the menu orientation is vertical. If bit 8 is not set, the menu is a horizontal one, like the menu title bar that appears at the top of the screen. If bit 7 is set (indicated by a value of 64/$40), the mouse pointer will be constrained to the area of the menu, so that it can't "fall off," causing the menu box to disappear. This is helpful for long menus, or sub-submenus where it is easy for the mouse pointer to move off of the menu and cause the menu to vanish. If this bit is not set, the pointer will not be constrained.

The second part of the flag indicates how many items there are in the menu. The number of menu items is ORed with the orientation and constraint bits to produce a single number.

In addition to the menu header, the definition table must have entries for each of the items that appear on the menu. The entry for each menu item consists of five bytes arranged this way:

```
.word TEXTPOINTER      ;address of the text string for this item
.byte ITEM_TYPE        ;flag for submenu or subroutine
.word ACTION_POINTER   ;address of submenu or subroutine
```

The first two bytes give the address of the text string that will be printed out for this particular menu item. The address is given in low-byte, high-byte order as usual. The text string consists of a group of ASCII characters ending with a zero byte. The next byte is a flag that indicates whether this menu selection causes some subroutine to be executed, unrolls yet another menu, or both. If bit 8 is set (with a value of 128/$80), the item is of the type SUB_MENU, which indicates that choosing this item opens up another menu. If bit 7 is set (with a value of 64/$40), the item is of type DYNAMIC_SUB_MENU. This means that choosing this item first executes a subroutine, and then opens the menu whose address the subroutine leaves in r0. Finally, if this flag is set to 0, the item is of type MENU_ACTION, which means that a subroutine will be executed when the menu item is chosen.

The final two bytes of the item entry also comprise an address in low-byte, high-byte order. What this address signifies depends on the type of menu item selected. If the item type is SUB_MENU, this address should point to another menu definition table that has been set up for that submenu. The menu definition table for submenus follows exactly the same format as that of the top menu, and you can link any number of horizontal and vertical menus, so that the main menu opens a submenu that opens a sub-submenu, and so on. If the item type is MENU_ACTION, the address should point to the beginning of a subroutine that will be executed when this menu item is chosen. This subroutine performs whatever action the menu selection indicates. It should include a call to a *GEOS* function to handle the disposition of the menu that called it (such as GotoFirstMenu), either erasing the menu or reactivating it for further selections. If the menu type is DYNAMIC_SUB_MENU, this address will also point to the beginning of a subroutine. The difference here is that after the subroutine is finished, a submenu will also be displayed. The user's subroutine must leave a pointer to the submenu in r0 when it is finished.

49492 $C154 RecoverMenu

49495 $C157 RecoverAllMenus

49498 $C15A DoIcons

Used to display and activate the icons whose definition table is pointed to by r0. From then on, when the user clicks on one of those icons, the subroutine specified in the table entry for that icon will be executed.

The icon definition table is the data structure used to supply *GEOS* with all of the information it needs to know about the icon. The table starts with a four-byte header arranged as follows:

```
.byte NUMBER_OF_ICONS   ;The number of icon entries in the
                        table
.word MOUSE_X           ;The horizontal position of the mouse
.byte MOUSE_Y           ;The vertical position of the mouse
```

The first byte is used to indicate the number of icons that will be defined in the body of the table. The word and the byte following it specify the horizontal and vertical coordinates at which the mouse pointer will be positioned after the icons are first displayed. After the header come the entries for each individual icon. Each entry consists of eight bytes, arranged in the following manner:

```
.word ICON_GRAPHICS_DATA  ;Address of BitmapUp format
                          image data
.byte ICON_XPOSITION      ;Horizontal position of top left
                          corner
.byte ICON_YPOSITION      ;Vertical position of top left
                          corner
.byte ICON_WIDTH          ;Image width in bytes
.byte ICON_HEIGHT         ;Image height in pixels
.word SUBROUTINE          ;Address of subroutine to be
                          executed
```

The first two bytes form the address of the data for the graphics image that represents the icon. This data is laid out in a special format, the same one used by the *GEOS* routine BitmapUp. See the jump table entry for BitmapUp, 49474 ($C142), for the details of this format. Though this format allows you to compress the data to save storage space, if the graphics data is less than 92 bytes long, you can bypass the compaction scheme by starting your data area with a Count byte equal to 128 plus the number of data bytes. This Count byte is followed by the actual data bytes that make up the image.

The next two bytes specify the coordinates for the top left corner of the image. Note that the horizontal position is specified in bytes rather than pixels, since each Bitmap Object starts on an even byte boundary. The two bytes after the position coordinates specify the width of the object (in bytes), and its height (in pixels). This infor-

mation is used to translate the single long string of graphics data into a rectangular image. The first data byte is drawn at position ICON_XPOSITION, ICON_YPOSITION, and each subsequent data byte is drawn to the immediate right of the previous one, until ICON_WIDTH number of bytes have been drawn. After each row is completed, a new row is started directly below it, until ICON_HEIGHT number of rows have been drawn, at which point the image is completed. The final two bytes in the icon entry give the address for the subroutine to execute when the icon is selected, in the normal low-byte, high-byte format.

All of the icons defined in the table will be active until another call to DoIcons substitutes a different icon table. The only way to get rid of icons completely is to erase the screen and use DoIcons to activate a table with only one icon, whose graphic image consists of one zero byte (making the picture invisible), and whose dispatch subroutine points to an RTS instruction (which means that the icon will do nothing if selected). *GEOS* expects that your application will be using at least one icon, so if you do not want to use any, make up this kind of null icon definition table.

Utilities

49501 $C15D DShiftLeft

Shifts the contents of the pseudoregister whose address is passed in the X register arithmetically to the left by the number of places indicated by the contents of the Y register.

49504 $C160 BBMult

Multiplies the two bytes found in the pseudoregister whose address is passed in the X register, and stores the resulting word in the pseudoregister whose address is passed in the Y register.

49507 $C163 BMult

Multiplies the byte found in the pseudoregister whose address is passed in the Y register by the word found in the pseudoregister whose address is passed in the X register, and stores the result in the latter.

49510 $C166 DMult

Multiplies the word found in the pseudoregister whose address is passed in the Y register by the word found in the pseudoregister whose address is passed in the X register, and stores the result in the latter.

49513 $C169 Ddiv

Divides the word found in the pseudoregister whose address is

passed in the X register by the word found in the pseudoregister whose address is passed in the Y register, and stores the result in the former.

49516 $C16C DSdiv
Divides the signed word found in the pseudoregister pointed to by the X register by the signed word found in the pseudoregister whose address is passed in the Y register, and stores the result in the former. The remainder is stored in r8.

49519 $C16F Dabs
Changes the contents of the word found in the pseudoregister whose address is passed in the X register to its twos complement absolute value.

49522 $C172 Dnegate
Changes the contents of the word found in the pseudoregister pointed to by the X register to its twos complement negative value.

49525 $C175 Ddec
Decrements the unsigned word found in the pseudoregister pointed to by the X register.

49528 $C178 ClearRam
Sets to zero the number of bytes specified in r0, starting with the address pointed to by r1.

49531 $C17B FillRam
Fills with the value in r2L the number of bytes specified in r0, starting with the address pointed to by r1.

49534 $C17E MoveData
Moves the number of bytes specified in r2 from the address pointed to by r0 to the address pointed to by r1. The two areas may overlap.

49537 $C181 InitRam
Initializes a block of RAM according to the Initialization Table pointed to by r0. This table may contain a number of entries, each of which consists of a word specifying a location in RAM, followed by a byte indicating the number of bytes to be placed at that location, followed by the actual data bytes to be placed there. An entry with a zero as its first word marks the end of the table.

49540 $C184 PutDecimal

49543 $C187 GetRandom
Generates the next pseudorandom number using the seed value found in the variable random (34058/$850A), where the result will also be stored. The formula used is random = (seed + 1) * 2 MOD 65521.

Other

49546 $C18A MouseUp
Redraws the mouse pointer and enables the mouse after a call to MouseOff

49549 $C18D MouseOff
Temporarily turns the mouse pointer off and disables the mouse

49552 $C190 DoPreviousMenu
Used by the applications menu dispatch routine when a menu item is selected
When the user makes one selection, the current menu can be rolled up to prevent any further selections being made. DoPreviousMenu erases the current submenu and reenables the previous one (usually the horizontal menu title bar).

49555 $C193 ReDoMenu
This is another routine that may be used by the applications menu dispatch routine when a menu item is selected. ReDoMenu reactivates the current menu after an item has been selected, thus allowing the user to make another selection from the same menu.

49558 $C196 GetSerialNumber

49561 $C199 Sleep
Used to discontinue execution of a subroutine for the number of jiffies (each equal to 1/60 second) indicated in r0
When Sleep is called, the subroutine is ended with an RTS instruction, but its address is remembered. When the designated time has passed, an interrupt resumes execution of the subroutine at the instruction following the JSR Sleep instruction.

49564 $C19C ClearMouseMode
Turns off the mouse pointer, and disables the mouse, menus, and icons

49567 $C19F i_Rectangle

Draws a rectangle and fills it with the current fill pattern (set by SetPattern)

The boundaries of the rectangle are specified by the data that follows immediately after the JSR instruction. The first two bytes give the coordinates for the top and bottom of the rectangle, and the two words which follow give the coordinates of the left and right sides.

49570 $C1A2 i_FrameRectangle

Draws the outline of a rectangle which is one pixel wide, and conforms to the pattern set by the current fill pattern (set by SetPattern)

The boundaries of the rectangle are specified by the data that follows immediately after the JSR instruction. The first two bytes give the coordinates for the top and bottom of the rectangle, and the two words which follow give the coordinates of the left and right sides.

49573 $C1A5 i_RecoverRectangle

Moves a rectangle from the background screen to the foreground screen

The boundaries of the rectangle are specified by the data that follows immediately after the JSR instruction. The first two bytes give the coordinates for the top and bottom of the rectangle, and the two words which follow give the coordinates of the left and right sides.

49576 $C1A8 i_GraphicsString

This routine executes the string of graphics commands that is stored immediately after the JSR instruction. For an explanation of these commands, see GraphicsString, location 49462 ($C136).

Background Generation

49579 $C1AB i_BitmapUp

Displays the compacted *GEOS* bitmap objects pointed to by the data word that follows the JSR instruction

The next two bytes specify the horizontal byte boundary (0–39) and the scan line at which the display should begin. The final two data bytes indicate the width of the object in bytes and its height in scan lines.

GEOS bitmap objects are laid out more like sprites than like the normal bitmap graphics. (For example, each data byte appears on-screen to the right of the previous one, rather than directly below it.) In addition, a compaction scheme called *run-length encoding* is used to reduce the size of the data. For more details concerning the format of *GEOS* bitmap objects, see the jump table entry for BitmapUp, location 49474 ($C142), above.

Character Manipulation

49582 $C1AE i_PutString

Prints a string of characters ending with a zero byte

The horizontal position of the text appears as a data word that immediately follows the JSR calls, while the vertical position appears as the next byte. The string itself follows.

This routine treats some nonprinting characters as commands. See PutString, location 49480 ($C148), for more information.

49585 $C1B1 GetRealSize

Returns the width, height, and baseline offset of the ASCII character passed in the accumulator, using the current font and the font style indicated by the X register. (The flags passed in this register are the same as those used in location currentMode; see location (46/$2E).) The width of the character is returned in the Y register, the height in the X register, and the distance from the top of the character to the baseline in the accumulator.

Utilities

49588 $C1B4 i_FillRam

Fills a number of bytes of RAM with a data character

The number of bytes to fill appears in the first data word following the JSR call, followed by the word address of the RAM area to fill, and the byte value with which to fill the RAM.

49591 $C1B7 i_MoveData

Moves a number of bytes from one memory area to another

The address of the source area is contained in the word following the JSR instruction, the address of the destination is in the word following that, and the last data word holds the number of bytes to move. The source and destination areas may overlap.

Routines Added Later

49594 $C1BA GetString

GetString accepts a string of up to r2L characters from the keyboard, terminated by a carriage return, and places that string in the buffer pointed to by r0. The input is echoed to the screen at the location specified in r11 (horizontal) and r1H (vertical). Optionally, the user can set bit 7 of r1L to indicate that he or she has supplied a routine pointed to by r4 that handles the case in which the user types more than the maximum number of characters. If this routine is not supplied, the system routine pointed to by stringFaultVector (33963/$84AB) is called when too many characters are entered.

After the string has been gathered, the routine whose address is in keyVector (33955/$84A3) is called.

49597 $C1BD GotoFirstMenu

Used to remove the submenu display and go back to the main menu

Called by the application when a menu item is selected and no further selections are available.

49600 $C1C0 InitTextPrompt

Creates an edit cursor which appears as a vertical bar whenever a string input routine like GetString is called

The accumulator holds the height of the cursor bar (in pixels). Sprite number 1 is used for the cursor.

49603 $C1C3 MainLoop

The *GEOS* Main Loop routine, which runs continuously and controls the execution of the application

49606 $C1C6 DrawSprite

Moves the 64-byte sprite shape data that starts with the address pointed to by r4 to the data area for the sprite whose number is in r3L

Remember that sprite 0 is used for the mouse pointer, and sprite 1 is used for the text cursor.

49609 $C1C9 GetCharWidth

This routine takes the ASCII character passed in the accumulator, and returns the width of that character in the currentFont and currentMode variables, and also in the accumulator.

49612 $C1CC LoadCharSet

Instructs *GEOS* to begin using the font whose FontInfoTab is pointed to by r0

49615 $C1CF PosSprite

Positions the sprite specified in r3L at horizontal position r4 and vertical position r5L

Note that *GEOS* positions the sprites using screen coordinates, and not the sprite hardware coordinates which may correspond to positions that are offscreen. (See locations 53248–53264 ($D000–$D010) on pages 126–27 for more information about the hardware coordinates.)

49618 $C1D2 EnablSprite

Turns on the sprite display bit for the sprite whose number is passed in r3L

49621 $C1D5 DisablSprite
Turns off the sprite display bit for the sprite whose number is passed in r3L

49624 $C1D8 CallRoutine
This routine is used at the Main Loop level to perform an indirect JSR to the routine. The high byte of the target routine is supplied in the X register, and the low byte in the accumulator.

Disk and Miscellaneous Routines

Since the *GEOS* disk filing system varies somewhat from the Commodore scheme, a short explanation of these differences is appropriate before we discuss the disk routines.

The first difference between *GEOS* files and old 64 files is that *GEOS* alters the standard directory entry scheme somewhat. The directory entry is a data structure stored on the disk that holds important information that the filing system must know about a particular file, such as the filename, file type, the beginning track and sector number, and so forth.

GEOS takes the part of the directory entry ordinarily used by the Commodore DOS to store information about relative files (which aren't supported by *GEOS*) and stores special *GEOS* file information there. Bytes 19–20, which ordinarily are used as a pointer to the first side sector of a relative file, are used to point to the track and sector of the file header block, a special *GEOS* file structure that is used to store icon image data and other important information. (More will be said about this header block later.) Byte 21 is used to store the *GEOS* structure type. *GEOS* supports two different file structure types: sequential and VLIR (Variable Length Indexed Record) files. Sequential files are laid out like the familiar Commodore sequential files. The structure of VLIR files will be discussed later. A value of 0 in byte 21 indicates a *GEOS* sequential file, and a value of 1 indicates a VLIR file. Byte 22 is used to store the *GEOS* file type. This is a number that specifies the function of the file. Possible values for file type include:

Number	File Type	Description
0	NOT_GEOS	Old 64 file without header
1	BASIC	Old 64 BASIC program
2	ASSEMBLY	Old 64 machine language program
3	DATA	Old 64 data file
4	SYSTEM	*GEOS* system file
5	DESK_ACC	*GEOS* desk accessory
6	APPLICATION	*GEOS* application
7	APPL_DATA	Data file for a *GEOS* application
8	FONT	*GEOS* font file

Number	File Type	Description
9	PRINTER	*GEOS* printer driver
10	INPUT_DEVICE	*GEOS* input device driver
11	DISK_DEVICE	*GEOS* disk device driver

Finally, bytes 23–27 are used to store the date and time of file creation. One byte each is allocated to year, month, day, hour, and minute. The year value is an offset from the base year of 1900, so a value of 86 here refers to 1986, and a value of 101 refers to the year 2001. The time is stored in 24-hour military format, where 0 is used for midnight, 12 is used for noon, 13 is used for 1 p.m., and so on.

In addition to these alterations in the directory entry for each file, the *GEOS* file system reserves an additional block of disk space for each file, called the file header block. This block contains even more information about the file, including the shape data for the icon used to represent the file on the deskTop. If you wish to add a custom icon to an old 64 program, you must manually alter the directory entry using a sector editor program, and then reserve a block in the BAM (Block Allocation Map) for the header block. The contents of this header block are as follows:

```
.word 0, 255                    ;track and sector pointer
.byte ICON_WIDTH                ;in bytes
.byte ICON_HEIGHT               ;in pixels
.byte ICON_SIZE + 128           ;number of bytes of image data
.byte data, data ...            ;63 bytes of image data
.byte COMMODORE_FILE_TYPE       ;type OR $80
.byte GEOS_FILE_TYPE            ;Application, system, data, and
                                 so on
.byte GEOS_STRUCTURE_TYPE       ;Sequential or VLIR
.word StartAddress              ;starting memory address for file
.word EndAddress                ;ending memory address for file
.word InitAddress               ;address of initialization routine
.byte 'permanent filename',0    ;20-byte permanent filename
.byte 'name of author',0        ;20-byte author's name
.byte 'name of parent file',0   ;20-byte name of parent file
```

Some of these deserve further explanation. For now, all icons are 3 bytes wide and 21 pixels high (just like a sprite). Therefore, ICON_WIDTH is always 3, ICON_HEIGHT is always 21, and ICON_SIZE + 128 always comes to 191. Bytes 5–67 contain the actual image data for the icon, laid out just as sprite data is in memory.

The COMMODORE_FILE_TYPE is set to one of the normal 64 DOS file types, and is used to distinguish between program and data files. These file types include sequential, or SEQ, files (129); PRG, or program, files (130); and USR, or user, files (131).

The GEOS_FILE_TYPE and GEOS_STRUCTURE_TYPE use the same values as those in the directory entry, as shown above.

The last three entries are each text strings up to 20 characters long, ending with a zero byte. The first is the permanent application name. You can use this space to store an application name which cannot be changed easily by the user, as the filename can. Checking the permanent name instead of the filename is therefore a more reliable test of what the file actually contains. When the *GEOS* file selector box is displayed in a dialog box, it is possible to specify that only certain files be displayed, based on the contents of the permanent filename. The next string contains a space for the author's name. The last string can be used to indicate the *parent* application for a data file—the program to which the data file belongs.

In addition to sequential access files, *GEOS* also provides an indexed file type called VLIR (Variable Length Indexed Record) files. A VLIR file consists of 0–127 records. Each record can be any length up to 32K, with the stipulation that the application must have enough free RAM available to be able to load the entire record into memory at once. VLIR files may be used to store both programs and data. An example of their use for programs would be an application that is too large to store in RAM all at once. A main part of the program might reside in RAM at all times as a sort of traffic director, and load in other parts of the program as overlays when necessary. Each discrete program module might be stored as one record in a VLIR file. An example of data files that might require that parts be paged in and out is font files for the various text fonts.

The directory entry for a sequential *GEOS* file contains a pointer to the track and sector numbers of the first block of the file. VLIR files use this pointer to specify the location of an Index Table. The Index Table is just an entire block of track and sector pointers, each of which points to the first block of a VLIR record. This block can hold 0–127 pairs of pointers. Each block of a VLIR record contains a pointer to the next block, so that the record is composed of a linked list of sectors, just like a normal sequential file.

To create a VLIR file, use the SaveFile routine and pass an end address that has a value 1 less than the start address of the file. This creates an empty VLIR file. To add data, you use OpenRecordFile to open the file, and AppendRecord to add a record. Only one VLIR file may be open at a time. After you have added records, you may use AppendRecord, InsertRecord, or DeleteRecord to change the composition of the file. When you have finished adding records, use CloseRecordFile to close the file.

49627 $C1DB CalcBlksFree

Calculates the number of blocks that are free on the disk whose directory header is pointed to by r5.

49630 $C1DE ChkDkGEOS
Checks to see if the disk in the current drive is a *GEOS* disk, and, if it is, sets the flag isGEOS to TRUE.

49633 $C1E1 NewDisk
Loads the Block Availability Map (BAM) from the disk specified in curDrive (33929/$8489) into the internal RAM of the disk drive. This must be performed before files on the new disk may be read.

49636 $C1E4 GetBlock
Loads a chain of r2L blocks, starting with track r1L, sector r1H, into a buffer pointed to by r4.

49639 $C1E7 PutBlock
Writes a block of RAM data (which includes track and sector information) pointed to by r4 to disk at track r1L, sector r1H.

49642 $C1EA SetGEOSDisk
This routine writes the *GEOS* ID string to an unused portion of the RAM image of the directory header stored at curDirHead (33280–33535/$8200–$82FF).

49645 $C1ED SaveFile
Saves the region of memory indicated by the file header block pointed to by r9 to a *GEOS* sequential disk file. The number of the directory page of the deskTop note pad in which to store the the file is passed in r10L. (A zero can be used to indicate that the search for a free directory entry should begin with the first entry.)

When this routine is being used to create a new, empty VLIR data file, the start address specified in the header block should be 0, and the ending address −1. When it is being used to create an empty VLIR program file, the start address should be the program load address, and the ending address should be 1 less than the start address.

49648 $C1F0 SetGDirEntry
Creates a *GEOS* directory entry for a file according to the file header whose address is pointed to by r9.

49651 $C1F3 BldGDirEntry
Builds a *GEOS* directory entry in memory at location dirEntryBuf (33792–33821/$8400–$841D), using the information in the file header pointed to by r9.

49654 $C1F6 GetFreeDirBlk
Locates a free directory entry on the note pad page indicated in r10L, or on the closest following page if that page is full.

49657 $C1F9 WriteFile
Writes the data block whose address is pointed to by r7 from memory to disk using the table of free disk blocks pointed to by r6.

49660 $C1FC BlkAlloc
Allocates blocks on disk for writing r2 number of bytes using the track and sector allocation table pointed to by r6.

49663 $C1FF ReadFile

49666 $C202 SmallPutChar
A fast, simplified version of PutChar (see 49477/$C145) that does not perform boundary checking, update the cursor position, or execute commands.

49669 $C205 FollowChain
Builds a track/sector table in the buffer pointed to by r3, for the file starting at track r1L, sector r1H.

49672 $C208 GetFile
Used to load the file whose name is contained in the zero-terminated text string pointed to by r6. If the file is an application or desk accessory, the program is run.

49675 $C20B FindFile
This routine loads the directory entry for the file whose name is contained in the zero-terminated text string pointed to by r6. If it is found, its Directory Block will be diskBlkBuf (32768/$8000), and its Directory entry in dirEntryBuf (34048/$8500); otherwise an error code will be returned in the X register.

49678 $C20E CRC

49681 $C211 LdFile
Loads a system file whose directory is pointed to by r9, and returns control to the caller without executing the program.

49684 $C214 EnterTurbo
Makes the 1541 drive start executing the disk turbo program. If the turbo code is not yet stored in disk RAM, this routine moves it there.

49687 $C217 LdDeskAcc
Loads the desk accessory file whose Directory Entry is pointed to by r9, and runs it. The appropriate portions of the current application are saved to disk so that it may be restored when the accessory exits.

49690 $C21A ReadBlock

49693 $C21D LdApplic
Loads the application file whose Directory Entry is pointed to by r9, and runs it.

49696 $C220 WriteBlock

49699 $C223 VerWriteBlock

49702 $C226 FreeFile

49705 $C229 GetFHdrInfo
Reads a data file of size r2, starting at track r1L, sector r1H, into memory at the address pointed to by r7.

49708 $C22C EnterDeskTop
Reinitializes *GEOS* and loads and runs the deskTop program. Application programs should do a JMP here when they are finished.

49711 $C22F StartAppl

49714 $C232 ExitTurbo
This routine switches the 1541 disk drive back to using its own ROM operating system, rather than the turbo code stored in disk RAM. It leaves the turbo code in disk RAM, however, so that turbo mode may be easily entered again.

49717 $C235 PurgeTurbo
Stops execution of the turbo code and marks the turboFlag for the current drive to indicate that the turbo code is no longer loaded in disk RAM.

49720 $C238 DeleteFile
Deletes the file whose name is contained in the zero-terminated string pointed to by r0.

49723 $C23B FindFTypes
This routine is used to build a list of files that match the file type passed in r7L. File types include NOT_GEOS, BASIC, ASSEMBLY, DATA, SYSTEM, DESK_ACC, APPL_DATA, FONT, PRINTER,

INPUT_DEVICE, and DISK_DEVICE. It is also possible to limit data files to those created by a particular application, by passing a pointer to a permanent name string in r10. The filenames, up to a maximum of r7L of them, will be placed in the user-supplied buffer pointed to by r6.

49726 $C23E RstrAppl
After a desk accessory program has exited, this routine replaces the contents of memory so as to restore the application that was running when the desk accessory started.

49729 $C241 ToBasic
Transfers control from *GEOS* to Commodore 64 BASIC.

49732 $C244 FastDelFile
Used to quickly delete the file whose name is contained in the zero-terminated string pointed to by r0, when a list containing the track and sector number of every disk block used by the file is pointed to by r3.

49735 $C247 GetDirHead
Reads the directory header from the current disk into the buffer at CurDirHead.

49738 $C24A PutDirHead
Writes the directory header from curDirHead out to disk.

49741 $C24D NxtBlkAlloc
Continues allocating blocks on disk for writing r2 number of bytes using the track and sector allocation table pointed to by r6 after BlkAlloc has allocated its maximum of 32,258 bytes. The last track and sector allocated by BlkAlloc is indicated by r3.

49744 $C250 ImprintRectangle
Moves the rectangle bordered by r3,r2L to r4,r3H from the foreground screen to the background screen.

49747 $C253 i_ImprintRectangle
Moves a rectangle from the foreground screen to the background screen. The boundaries of the rectangle are specified by the data that follows immediately after the JSR instruction. The first two bytes give the coordinates for the top and bottom of the rectangle, and the two words which follow give the coordinates of the left and right sides.

49750 $C256 DoDlgBox

Opens the dialog box whose definition table is pointed to by r0, saving the portion of the screen that the box overwrites and all system variables. This dialog box remains active until the dialog is exited with a call to RstrFrmDialog.

A dialog box is a window that opens on top of the application screen, and temporarily takes control away from the application, forcing the user to take some action. When the user responds, the dialog is exited, and the application is restored. A dialog box can be used to display a message to the user, or to get text input, numeric input, or other types of instructions from the user. It can even provide a list of files to select from. The key to using a dialog box is in setting up the data table that defines its appearance and function. This table is made up of several parts, each of which can vary in length depending on how many entries you add.

The first part of the table specifies the position of the dialog box, and the fill pattern used to draw the shadow box that outlines the dialog box below and to the right. The first byte is a flag that shows whether the position used is the standard *GEOS* position or a user-defined position. Possible values are 128/$80, indicating that the box will be placed according to *GEOS* defaults, and 0, indicating that the box will be placed at a user-defined location. To these values are added the number of the fill pattern (0–33) used to shadow the box. If a user-defined position is specified, the next six bytes are used to define the top right and bottom left corners of the box:

```
.byte TopY      ;y1 (0-199)
.byte BottomY   ;y2 (0-199)
.word LeftX     ;x1 (0-319)
.word RightX    ;x2 (0-319)
```

After the position specifier come icon blocks or command blocks. Icon blocks are used to put up icons. *GEOS* provides a number of system icons that are available to the programmer. These icons display various messages, and when the user clicks on one, the dialog box exits, returning the number of the icon selected in r0L. To include one of these icons in the dialog box, the programmer need only include an icon block in the definition table, in the following format:

```
.byte ICON_NUMBER
.byte XPosition          ;in bytes—drawing starts on a byte
.byte YPosition          ;in pixels
```

The first value is the icon number. The system provides the following icons:

Number	Icon
1	OK
2	Cancel
3	Yes
4	No
5	OPEN
6	DISK
7–10	Reserved for future use

The next two bytes give position offsets for the icon. These offsets show how far the icon should be placed from the upper left corner of the box. The horizontal offset is measured in bytes, while the vertical one is measured in pixels.

In addition to icon blocks, your dialog box definition table may include one or more command blocks. These consist of a command number followed by some additional information needed to implement the command. These commands allow you to draw text, receive text from the user, or even set up a scrolling file-selection window. The eight defined commands are discussed below. After you have included all of the icon blocks and command blocks that you want in the dialog, use an entry of a single zero byte to signal the end of the dialog box definition table.

DBTXSTR (11) displays a text string ending in a zero byte. The format for this command block is:

```
.byte 11              ;command number
.byte Xoffset         ;horizontal offset from left of box
                      ; (in bytes)
.byte Yoffset         ;vertical offset from top of box
                      ; (in pixels)
.word Text_Pointer    ;address of the text string
```

DBVARSTR (12) displays a variable text string ending in a zero byte. You can change the contents of the string by changing the contents of the pseudoregister used to point to it between calls to DoDlgBox. The format for this command block is:

```
.byte 12              ;command number
.byte Xoffset         ;horizontal offset from left of box
                      ; (in bytes)
.byte Yoffset         ;vertical offset from top of box
                      ; (in pixels)
.byte Register_Number ;number of pseudoregister that contains
                      ; the address of the string
```

DBGETSTRING (13) gets a string of ASCII characters from the user and places them into a text buffer. The string is displayed in the dialog box as the user types it in. The format for this command block is:

```
.byte 13                ;command number
.byte Xoffset           ;horizontal offset from left of box
                        ; (in bytes)
.byte Yoffset           ;vertical offset from top of box
                        ; (in pixels)
.byte Register_Number   ;number of pseudoregister that contains
                        ; the address of the buffer in which the
                        ; the string will be stored
.byte Maximum_Chars     ;the maximum number of characters to be
                        ; stored in the text buffer
```

DBSYSOPV (14) causes any mouse button press that occurs when the pointer is not over an icon to force a return to the application. This command block requires no additional information, and consists only of the command number:

```
.byte 14                ;command number
```

DBGRPHSTR (15) causes a graphics string to be executed. This is a string of graphics drawing instructions like that used by the system routine GraphicsString (see location 49462/$C136 for more information). The format for this command block is:

```
.byte 15                ;command number
.word String_Pointer    ;address of the graphics command string
```

DBGETFILES (16) opens a file-selection window. This is a window on the left side of the dialog box that contains a list of all of the files that match the type specifications which are passed to the routine. If a permanent filename is specified in the command, only files whose own permanent filenames match the one specified will be displayed. Otherwise, all files will be displayed. If there are more names than can fit in the window, the user can see the rest of the names by pressing the scroll arrows that appear under the box. The format for this command block is:

```
.byte 16        ;command number
.byte Xoffset   ;horizontal offset from left of box
                ; (in bytes)
.byte Yoffset   ;vertical offset from top of box
                ; (in pixels)
```

In addition, several pieces of information are passed to this command in the zero-page pseudoregisters before DoDlgBox is called:

```
r7L File_Type          ;GEOS file type (SYSTEM, DESk_ACC, and
                       so on)
r5 Filename_Buffer     ;pointer to address of buffer for storing
                       ; filenames (up to maximum of 15)
r10 Permanent_Pointer  ;pointer to a string containing the
                       ; permanent filename to which the
                       ; display is restricted. For example,
                       ; if the string reads geoPaint, only
                       ; files with that permanent filename
                       ; are listed. If this is set to 0, all
                       ; files will be found.
```

DBOPVEC (17) sets a user-defined OtherPressVector. This causes the execution of a user-supplied subroutine when a mouse button is pressed other than when the pointer is positioned over an icon. This routine may return to the dialog box code with an RTS, or exit the dialog with a JMP to RstrFrmDialog. The format for this command block is:

```
.byte 17          ;command number
.word UserVector  ;address of OtherPress routine
```

DBUSRICON (18) sets up a user-defined icon. This icon is defined by an icon table like that used by DoIcons (see location 49498/$C15A for more information about the structure of the entries in the icon definition table). The position bytes in the icon table should be set to zero, however, since the offset values in the command block will be used to position the icon.

```
.byte 18          ;command number
.word Icon_Table  ;address of user icon definition table
.byte Xoffset     ;horizontal offset from left of box
                  ; (in bytes)
.byte Yoffset     ;vertical offset from top of box
                  ; (in pixels)
```

DB_USR_ROUT (19) installs a user-supplied subroutine that will be executed as soon as the dialog box is drawn. This subroutine can be used for almost any purpose. It returns to the dialog box code with an RTS instruction. The format for this command block is:

```
.byte 18          ;command number
.word UserVector  ;address of the user-supplied subroutine
```

49753 $C259 RenameFile

Renames the file whose old name is contained in the zero-terminated text string pointed to by r6 with the new name contained in the zero-terminated text string pointed to by r0.

49756 $C25C InitForIO

Part of the *GEOS* Operating System program is stored in the RAM that lies beneath the BASIC ROM at location 40960 ($A000), the I/O registers at location 53248 ($D000), and the Kernal ROM at location 57344 ($E000). Therefore, most of the time, the memory configuration register at location 1 is set to a value of 48 ($30), which gives the system 64K of RAM, and no ROM or I/O registers. When an application needs to use the I/O registers or the Kernal ROM routines, it needs to stop all interrupts, and to change the memory configuration register so that the system can access the I/O and ROM. This routine performs those functions.

49759 $C25F DoneWithIO

After the application has finished accessing I/O and system ROM, this routine restores interrupts, and resets the memory map back to 64K of RAM.

49762 $C262 DShiftRight

Shifts the contents of the pseudoregister whose address is passed in the X register arithmetically to the right by the number of places indicated by the contents of the Y register.

49765 $C265 CopyString

Copies a text string that ends in a zero byte to another location

The address of the source string is contained in the zero-page pseudoregister indicated by the X register, and the destination address is contained in the pseudoregister indicated by the Y register.

49768 $C268 CopyFString

Copies a text string of a fixed length to another location

The address of the source string is contained in the zero-page pseudoregister indicated by the X register, the destination address is contained in the pseudoregister indicated by the Y register, and the length of the string is passed in the accumulator.

49771 $C26B CmpString

Compares a text string that ends in a zero byte with another such string

The address of the first string is contained in the zero-page pseudoregister indicated by the X register, and the address of the second string is contained in the pseudoregister indicated by the Y register. Upon completion, the status register Z flag is set if the two strings are equal, and the N flag is set if the first nonmatching character of the first string was lower in value than that of the second string.

49774 $C26E CmpFString
Compares two fixed-length text strings
The address of the first string is contained in the zero-page pseudoregister indicated by the X register, the address of the second string is contained in the pseudoregister indicated by the Y register, and the length of the strings is passed in the accumulator. Upon completion, the status register Z flag is set if the two strings are equal, and the N flag is set if the first nonmatching character of the first string was lower in value than that of the second string.

49777 $C271 FirstInit

49780 $C274 OpenRecordFile
Opens the existing VLIR file whose name is contained in the zero-terminated text string pointed to by r0 for access.

49783 $C277 CloseRecordFile
Calls UpdateRecordFile and sets an internal variable to show that there are no VLIR files currently open

49786 $C27A NextRecord
Advances the VLIR current record pointer to the next record

49789 $C27D PreviousRecord
Moves the VLIR current record pointer back to the previous record

49792 $C280 PointRecord
Moves the VLIR current record pointer directly to the record indicated by the contents of the accumulator

49795 $C283 DeleteRecord
Deletes the current record, and moves the current record pointer to the record following the one deleted

49798 $C286 InsertRecord

49801 $C289 AppendRecord

49804 $C28C ReadRecord
Reads the current record (up to a maximum of r2 bytes) into the RAM buffer whose address is pointed to by r7

49807 $C28F WriteRecord
Writes r2 bytes from the RAM buffer whose address is pointed to by r7 to the current record, deleting the old contents of the record

49810 $C292 SetNextFree

Used to find the first free sector on disk after track r3L and sector r3H

49813 $C295 UpdateRecordFile

Updates the VLIR file IndexTable, the time and date stamp in the Directory Entry, and the Block Allocation Map.

49816 $C298 GetPtrCurDkNm

Returns the address of the name of the current disk in the zero-page pseudoregister pointed to by the X register.

49819 $C29B PromptOn

Displays the cursor that was set up with InitTextPrompt at the horizontal position indicated by the variable stringX (33982/$84BE) and the vertical position indicated by stringY (33984/$84C0).

49822 $C29E PromptOff

Turns off the text cursor

49825 $C2A1 OpenDisk

Used after SetDevice to initialize the disk drive for reading or writing to disk. This routine reads the disk name into DrACurDkNm or DrBCurDkNm, sets r5 to point to the disk name, places the disk's directory header in curDirHead, and sets the isGEOS flag to show that the disk is a *GEOS* disk.

49828 $C2A4 DoInlineReturn

Common return for inline routines.

49831 $C2A7 GetNextChar

Returns the next character from the keyboard input buffer in the accumulator, or zero if no character is waiting.

49834 $C2AA BitmapClip

Displays a portion of the bitmap object pointed to by r0

The top left corner of the object is positioned at the horizontal byte boundary r1L (0–39) and the scan line r1H. The width of the display in bytes (0–39) is passed in r2L, and the height in scan lines in r2H. The horizontal offset—expressed as the number of bytes of each line that should be skipped before printing the image in the window—is passed in r11L, and the number of bytes that remain unprinted at the end of each line is passed in r11H. The vertical offset is passed in r12 as the number of scan lines to skip before displaying the object.

49837 $C2AD FindBAMBit

49840 $C2B0 SetDevice

Turns over the serial bus to the device whose device number is passed in the accumulator. The global variables curDevice and curDrive are updated, and any errors are returned in the X register.

49843 $C2B3 IsMseInRegion

Tests whether the mouse pointer is within the region whose upper left corner is r3, r2L and whose lower right corner is r4, r2H. If the pointer is within the region, the accumulator contains a nonzero value on return.

49846 $C2B6 ReadByte

49849 $C2B9 FreeBlock

49852 $C2BC ChangeDiskDevice

49855 $C2BF RstrFrmDialog

Exits from the dialog box routine, restoring system variables and the screen

If one of the system icons was used to exit the dialog, the icon number is returned in r0L.

49858 $C2C2 Panic

This is the default routine that is called when a BRK instruction is encountered. It opens a dialog box that displays the address of the BRK, and then quits.

49861 $C2C5 BitOtherClip

Uses the input routine pointed to by r13 (usually ReadByte) to read in a Count/Bitmap group of compacted bitmap object data to the 134-byte buffer pointed to by r0. Once the buffer has the compacted data in it, the routine uncompacts and displays the object. The top left corner of the object is positioned at the horizontal byte boundary r1L (0–39) and the scan line r1H. The width of the display in bytes (0–39) is passed in r2L, and the height in scan lines in r2H. The horizontal offset—expressed as the number of bytes of each line that should be skipped before printing the image in the window—is passed in r11L, and the number of bytes that remain unprinted at the end of each line is passed in r11H. The vertical offset is passed in r12 as the number of scan lines to skip before displaying the object.

After the clipped bitmap is displayed, BitOtherClip calls the subroutine pointed to by r14. This subroutine should reload r0 with the address of the buffer.

49864	**$C2C8**	**StashRAM**
49867	**$C2CB**	**FetchRAM**
49870	**$C2CE**	**SwapRAM**
49873	**$C2D1**	**VerifyRAM**
49876	**$C2D4**	**DoRAMOp**

Location Range: 53248–57343 ($D000–$DFFF)

4K *GEOS* code or I/O space

Location Range: 57344–65151 ($E000–$FE7F)

8K *GEOS* code or 64 Kernal ROM

Location Range: 65152–65529 ($FE80–$FFF9)

MOUSE_BASE
GEOS input driver

The input device is the device used to move the mouse pointer around the screen. The default device used for this purpose is a joystick, or a mouse that emulates a joystick. The program that reads the input device and reports the results to *GEOS* is located here. If *GEOS* is to use an input device other than the joystick, a new program that can read that specific device and report the results to *GEOS* must be placed here instead of the default joystick handler. Any such device must support three basic functions. The first nine bytes of this location range are used for a jump table, which provides fixed entry points for each of these routines.

65152	**$FE80**	**InitMouse**

InitMouse performs whatever initialization the device requires, and sets the variables mouseXPosition (58/$3A), mouseYPosition (60/$3C), and mouseData (34053/$8505) to their initial values.

65155 $FE83 SlowMouse

This routine slows down the input device so that it requires more time and movement to advance the mouse pointer a given distance. *GEOS* calls this routine when it opens a menu, and applications may call it as well.

65158 $FE86 UpdateMouse

This is the routine that *GEOS* calls during each interrupt to update the global variables that are associated with the input device. The routine should update mouseXPosition (58/$3A), mouseYPosition (60/$3C), mouseData (34053/$8505), pressFlag (57/$39), and inputData (34054/$8506).

Appendices

Appendix A

A Beginner's Guide to Typing In Programs

What Is a Program?

A computer cannot perform any task by itself. Like a car without gas, a computer has *potential,* but without a program, it isn't going anywhere. Most of the programs published in this book are written in a computer language called BASIC. BASIC is easy to learn and is built into all Commodore 64s.

BASIC Programs

Computers can be picky. Unlike the English language, which is full of ambiguities, BASIC usually has only one right way of stating something. Every letter, character, or number is significant. A common mistake is substituting a letter such as O for the numeral 0, a lowercase l for the numeral 1, or an uppercase B for the numeral 8. Also, you must enter all punctuation such as colons and commas just as they appear in the book. Spacing can be important. To be safe, type in the listings *exactly* as they appear.

Braces and Special Characters

The exception to this typing rule is when you see the braces, such as {DOWN}. Anything within a set of braces is a special character or characters that cannot easily be listed on a printer. When you come across such a special statement, refer to Appendix B, "How to Type In Programs."

About DATA Statements

Some programs contain a section or sections of DATA statements. These lines provide information needed by the program. Some DATA statements contain actual programs (called machine language); others contain graphics codes. These lines are especially sensitive to errors.

If a single number in any one DATA statement is mistyped, your machine could lock up, or crash. The keyboard and STOP key may seem dead, and the screen may go blank. Don't panic — no damage is done. To regain control, you have to turn off your computer, then turn it back on. This will erase whatever program was in memory, *so always SAVE a copy of your program before you RUN it.* If

your computer crashes, you can LOAD the program and look for your mistake.

Sometimes a mistyped DATA statement will cause an error message when the program is RUN. The error message may refer to the program line that READs the data. *The error is still in the DATA statements, though.*

Get to Know Your Machine

You should familiarize yourself with your computer before attempting to type in a program. Learn the statements you use to store and retrieve programs from tape or disk. You'll want to save a copy of your program, so that you won't have to type it in every time you want to use it. Learn to use your machine's editing functions. How do you change a line if you made a mistake? You can always retype the line, but you at least need to know how to backspace. Do you know how to enter reverse video, lowercase, and control characters? It's all explained in your computer's manuals.

A Quick Review

1. Type in the program a line at a time, in order. Press RETURN at the end of each line. Use backspace or the back arrow to correct mistakes.
2. Check the line you've typed against the line in the book. You can check the entire program again if you get an error when you RUN the program.

Appendix B

How to Type In Programs

To make it easy to know exactly what to type when entering one of these programs into your computer, we have established the following listing conventions.

Generally, Commodore 64 program listings will contain words within braces which spell out any special characters: {DOWN} would mean to press the cursor down key. {5 SPACES} would mean to press the space bar five times.

To indicate that a key should be *shifted* (hold down the SHIFT key while pressing the other key), the key would be underlined in our listings. For example, S would mean to type the S key while holding the SHIFT key. This would appear on your screen as a heart symbol. If you find an underlined key enclosed in braces (e.g.,{10 N}), you should type the key as many times as indicated (in our example, you would enter ten shifted N's).

If a key is enclosed in special brackets, [<>], you should hold down the *Commodore key* while pressing the key inside the special brackets. (The Commodore key is the key in the lower-left corner of the keyboard.) Again, if the key is preceded by a number, you should press the key as many times as necessary.

Rarely, you'll see a solitary letter of the alphabet enclosed in braces. These characters can be entered by holding down the CTRL key while typing the letter in the braces. For example, {A} would indicate that you should press CTRL-A.

About the *quote mode:* You know that you can move the cursor around the screen with the CRSR keys. Sometimes a programmer will want to move the cursor under program control. That's why you see all the {LEFT}'s, {HOME}'s, and {BLU}'s in our programs. The only way the computer can tell the difference between direct and programmed cursor control is the quote mode.

Once you press the quote (the double quote, SHIFT-2), you are in the quote mode. If you type something and then try to change it by moving the cursor left, you'll only get a bunch of reverse-video lines. These are the symbols for cursor left. The only editing key that isn't programmable is the DEL key; you can still use DEL to back up and edit the line. Once you type another quote, you are out of quote mode.

You also go into quote mode when you INSerT spaces into a

Appendix B

line. In any case, the easiest way to get out of quote mode is to just press RETURN. You'll then be out of quote mode and you can cursor up to the mistyped line and fix it.

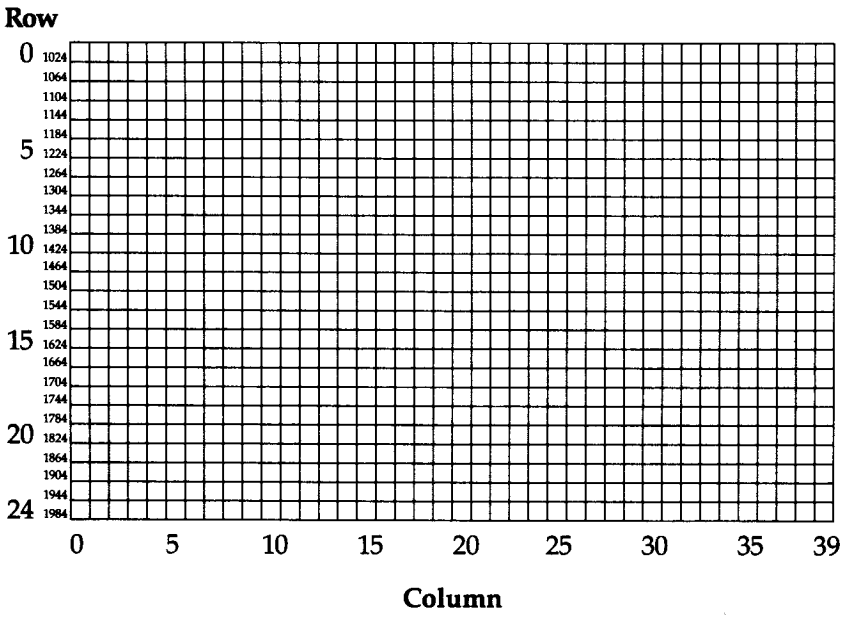
Use the following table when entering cursor and color control keys:

When You Read:	Press:	See:
{CLR}	SHIFT + CLR/HOME	
{HOME}	CLR/HOME	
{UP}	SHIFT + CRSR ↑	
{DOWN}	CRSR ↓	
{LEFT}	SHIFT + CRSR ←	
{RIGHT}	CRSR →	
{RVS}	CTRL + 9	
{OFF}	CTRL + 0	
{BLK}	CTRL + 1	
{WHT}	CTRL + 2	
{RED}	CTRL + 3	
{CYN}	CTRL + 4	
{PUR}	CTRL + 5	
{GRN}	CTRL + 6	
{BLU}	CTRL + 7	
{YEL}	CTRL + 8	

When You Read:	Press:	See:
{1}	COMMODORE + 1	
{2}	COMMODORE + 2	
{3}	COMMODORE + 3	
{4}	COMMODORE + 4	
{5}	COMMODORE + 5	
{6}	COMMODORE + 6	
{7}	COMMODORE + 7	
{8}	COMMODORE + 8	
{F1}	f1	
{F2}	SHIFT + f1	
{F3}	f3	
{F4}	SHIFT + f3	
{F5}	f5	
{F6}	SHIFT + f5	
{F7}	f7	
{F8}	SHIFT + f7	
£	£	

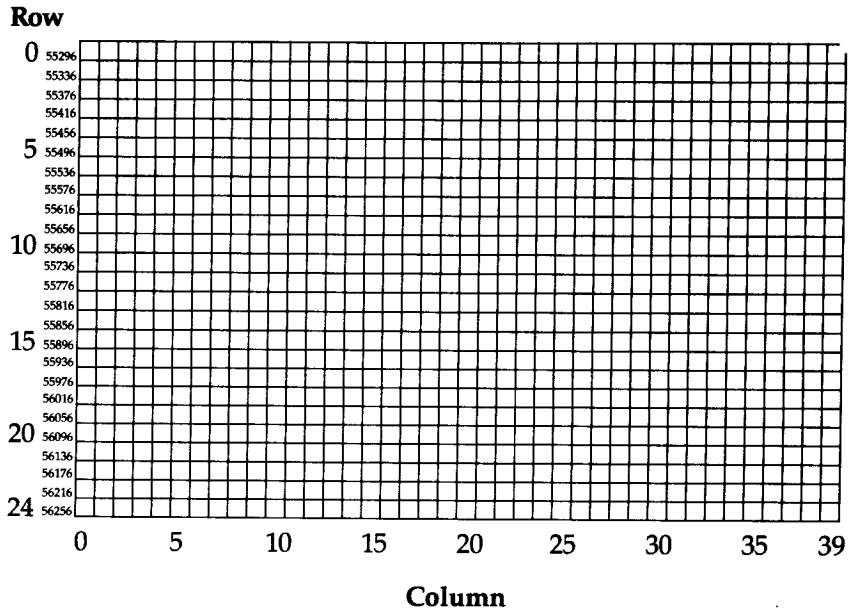
Appendix C

Screen Location Table



Appendix D

Screen Color Memory Table



Appendix E

Screen Color Codes

Value To POKE For Each Color

Color	Low nybble color value	High nybble color value	Select multicolor color value
Black	0	0	8
White	1	16	9
Red	2	32	10
Cyan	3	48	11
Purple	4	64	12
Green	5	80	13
Blue	6	96	14
Yellow	7	112	15
Orange	8	128	—
Brown	9	144	—
Light Red	10	160	—
Dark Gray	11	176	—
Medium Gray	12	192	—
Light Green	13	208	—
Light Blue	14	224	—
Light Gray	15	240	—

Where To POKE Color Values For Each Mode

Mode*	Bit or bit-pair	Location	Color value
Regular text	0	53281	Low nybble
	1	Color memory	Low nybble
Multicolor text	00	53281	Low nybble
	01	53282	Low nybble
	10	53283	Low nybble
	11	Color memory	Select multicolor
Extended color text †	00	53281	Low nybble
	01	53282	Low nybble
	10	53283	Low nybble
	11	53284	Low nybble
Bitmapped	0	Screen memory	Low nybble ‡
	1	Screen memory	High nybble ‡
Multicolor bitmapped	00	53281	Low nybble
	01	Screen memory	High nybble ‡
	10	Screen memory	Low nybble ‡
	11	Color memory	Low nybble

Appendix E

* For all modes, the screen border color is controlled by POKEing location 53280 with the low nybble color value.
† In extended color mode, Bits 6 and 7 of each byte of screen memory serve as the bit-pair controlling background color. Because only Bits 0-5 are available for character selection, only characters with screen codes 0-63 can be used in this mode.
‡ In the bitmapped modes, the high and low nybble color values are ORed together and POKEd into the *same location* in screen memory to control the colors of the corresponding *cell* in the bitmap. For example, to control the colors of cell 0 of the bitmap, OR the high and low nybble values and POKE the result into location 0 of screen memory.

Appendix F

ASCII Codes

ASCII	CHARACTER	ASCII	CHARACTER
5	WHITE	50	2
8	DISABLE SHIFT COMMODORE	51	3
		52	4
9	ENABLE SHIFT COMMODORE	53	5
		54	6
13	RETURN	55	7
14	LOWERCASE	56	8
17	CURSOR DOWN	57	9
18	REVERSE VIDEO ON	58	:
19	HOME	59	;
20	DELETE	60	<
28	RED	61	=
29	CURSOR RIGHT	62	>
30	GREEN	63	?
31	BLUE	64	@
32	SPACE	65	A
33	!	66	B
34	"	67	C
35	#	68	D
36	$	69	E
37	%	70	F
38	&	71	G
39	'	72	H
40	(	73	I
41	)	74	J
42	*	75	K
43	+	76	L
44	,	77	M
45	-	78	N
46	.	79	O
47	/	80	P
48	0	81	Q
49	1	82	R

Appendix F

ASCII	CHARACTER	ASCII	CHARACTER
83	S	120	
84	T	121	
85	U	122	
86	V	123	
87	W	124	
88	X	125	
89	Y	126	
90	Z	127	
91	[	129	ORANGE
92	£	133	f1
93	]	134	f3
94	↑	135	f5
95	←	136	f7
96		137	f2
97		138	f4
98		139	f6
99		140	f8
100		141	SHIFTED RETURN
101		142	UPPERCASE
102		144	BLACK
103		145	CURSOR UP
104		146	REVERSE VIDEO OFF
105		147	CLEAR SCREEN
106		148	INSERT
107		149	BROWN
108		150	LIGHT RED
109		151	GRAY 1
110		152	GRAY 2
111		153	LIGHT GREEN
112		154	LIGHT BLUE
113		155	GRAY 3
114		156	PURPLE
115		157	CURSOR LEFT
116		158	YELLOW
117		159	CYAN
118		160	SPACE
119		161	

Appendix F

ASCII	CHARACTER	ASCII	CHARACTER
162		200	
163		201	
164		202	
165		203	
166		204	
167		205	
168		206	
169		207	
170		208	
171		209	
172		210	
173		211	
174		212	
175		213	
176		214	
177		215	
178		216	
179		217	
180		218	
181		219	
182		220	
183		221	
184		222	
185		223	
186		224	SPACE
187		225	
188		226	
189		227	
190		228	
191		229	
192		230	
193		231	
194		232	
195		233	
196		234	
197		235	
198		236	
199		237	

Appendix F

ASCII	CHARACTER
238	
239	
240	
241	
242	
243	
244	
245	
246	
247	
248	
249	
250	
251	
252	
253	
254	
255	

0-4, 6, 7, 10-12, 15, 16, 21-27, 128, 130-132, and 143 are not used.

Appendix G

Screen Codes

POKE	Uppercase and Full Graphics Set	Lower- and Uppercase	POKE	Uppercase and Full Graphics Set	Lower- and Uppercase
0	@	@	31	←	←
1	A	a	32	-space-	
2	B	b	33	!	!
3	C	c	34	"	"
4	D	d	35	#	#
5	E	e	36	$	$
6	F	f	37	%	%
7	G	g	38	&	&
8	H	h	39	'	'
9	I	i	40	(	(
10	J	j	41	)	)
11	K	k	42	*	*
12	L	l	43	+	+
13	M	m	44	,	,
14	N	n	45	-	-
15	O	o	46	.	.
16	P	p	47	/	/
17	Q	q	48	0	0
18	R	r	49	1	1
19	S	s	50	2	2
20	T	t	51	3	3
21	U	u	52	4	4
22	V	v	53	5	5
23	W	w	54	6	6
24	X	x	55	7	7
25	Y	y	56	8	8
26	Z	z	57	9	9
27	[	[	58	:	:
28	£	£	59	;	;
29	]	]	60	<	<
30	↑	↑	61	=	=

Appendix G

POKE	Uppercase and Full Graphics Set	Lower- and Uppercase
62	>	>
63	?	?
64		
65		A
66		B
67		C
68		D
69		E
70		F
71		G
72		H
73		I
74		J
75		K
76		L
77		M
78		N
79		O
80		P
81		Q
82		R
83		S
84		T
85		U
86		V
87		W
88		X
89		Y
90		Z
91		
92		
93		
94		
95		
96	-space-	
97		
98		

POKE	Uppercase and Full Graphics Set	Lower- and Uppercase
99		
100		
101		
102		
103		
104		
105		
106		
107		
108		
109		
110		
111		
112		
113		
114		
115		
116		
117		
118		
119		
120		
121		
122		
123		
124		
125		
126		
127		

Appendix H

Commodore 64 Keycodes

Key	Keycode	Key	Keycode
A	10	6	19
B	28	7	24
C	20	8	27
D	18	9	32
E	14	0	35
F	21	+	40
G	26	-	43
H	29	£	48
I	33	CLR/HOME	51
J	34	INST/DEL	0
K	37	←	57
L	42	@	46
M	36	*	49
N	39	↑	54
O	38	:	45
P	41	;	50
Q	62	=	53
R	17	RETURN	1
S	13	,	47
T	22	.	44
U	30	/	55
V	31	CRSR↑↓	7
W	9	CRSR⇄	2
X	23	f1	4
Y	25	f3	5
Z	12	f5	6
1	56	f7	3
2	59	SPACE	60
3	8	RUN/STOP	63
4	11	NO KEY	
5	16	PRESSED	64

The keycode is the number found at location 197 for the current key being pressed. Try this one-line program:

10 PRINT PEEK (197): GOTO 10

Index (By Memory Location)

ABS 48216
AND 45033
ASC 46987
ATN 58126
BASIC
  adding new commands 115, 768
  current line number 57
  execution of statements 776, 42980
  expression evaluation 778, 44446, 44675
  function evaluation 44967
  pointer to bottom of string text 51
  pointer to current data item address 65
  pointer to current statement address 61
  pointer to end of array storage 49
  pointer to start of array storage 47
  pointer to start of program text 43
  pointer to start of variable storage 45
  pointer to top of BASIC RAM 55
  program text storage 2048
  RAM vector table 768
Buffer
  cassette I/O buffer 178, 828
  keyboard buffer 198, 631
  RS-232 input buffer 247
  RS-232 output buffer 249
  text input buffer 512
cartridge, autostart ROM 32768
cassette
  data output line 01
  I/O buffer 828
  Kernal ROM routines 63466-64737
  motor control 01, 192
  switch sense 01
character generator ROM 01, 4096, 36864, 53248
character graphics 53248
CHAREN 01
CHRGET 115, 58274
CHR$ 46828
CIA (Complex Interface Adapter)
  CIA #1 56320-56335
  CIA #2 56576-56591
  data direction registers 56322, 56323, 56578, 56579
  data ports 56320, 56321, 56576, 56577
  timers 56334, 56335, 56590, 56591
clock
  clock speed (6510 microprocessor) 56334
  software clock 160
  Time of Day clock 56328, 56584
CLOSE 57799
CLR 42590
CMD 43654
cold start, BASIC 40960, 58260
color
  background 53281
  border (frame) 53280
  color codes 646
  color RAM nybbles 55296
  current character color 646
  multicolor background registers 53282, 53283
  PETASCII color change characters 59601
  sprite color registers 53287-53294
  sprite multicolor registers 53285, 53286
CONT 43095
COS 57956
DATA 43256
data direction register 00, 56322, 56323, 56578, 56579
data port 01, 56320, 56321, 56576, 56577
DEF 46003
DIM 45185
dynamic keyboard 198, 631
error
  BASIC error handler 768, 42039, 58251
  error message control flag 157
  I/O error status 144
  RS-232 I/O error status 663
EXP 49133
expression evaluation 778, 44446, 44675
floating point
  addition 47207, 47210
  division 47887, 47890
  fixed-floating point conversion 05, 45969
  floating-fixed point conversion 03, 45482, 45503
  Floating Point Accumulators 97, 110
  floating point-ASCII conversion 48605

multiplication 47656, 47667
subtraction 47194, 47197
FN 46068
FOR 42818
FRE 45949
garbage collection, string variable 46374
GET, GET# 43899
GOSUB 43139
GOTO 43168
graphics
bitmapped graphics 53265
character graphics 53248
extended background color mode 53265
fine scrolling 53265, 53271
multicolor mode 53271
raster position 53265, 53266
screen blanking 53265
sprites *see* sprite graphics
HIRAM 01
IF 43304
INT 48332
interrupt
CIA hardware FLAG line 56589, 56333
CIA serial shift register 56589, 56333
CIA Time of Day clock alarm 56589, 56333
CIA Timers A and B 56589, 56333
IRQ handler 59953, 65352
IRQ vector 788, 65534
light pen IRQ 53273, 53274
NMI handler 65091
NMI vector 792, 65530
raster compare IRQ 53266, 53273, 53274
sprite-display data collision IRQ 53273, 53275
sprite-sprite collision IRQ 53273, 53275
INPUT 43967
INPUT# 43941
I/O
current device number 186
current filename address 187
current filename length 183
current input device 153
current I/O channel number 19
current logical file number 184
current output device 154
current secondary address 185
device number table 611
logical file table 601
number of I/O files open 152
RS-232 status 663
secondary address table 621
status word codes 144
joystick controllers 56320, 56321
Kernal
jump table 65409
RAM vector table 794
ACPTR 60947, 65445
CHKIN 798, 61966, 65478
CHKOUT 800, 62032, 65481
CHRIN 804, 61783, 65487
CHROUT 806, 61898, 65490
CINT 65371, 65409
CIOUT 60893, 65448
CLALL 812, 62255, 65511
CLOSE 796, 62097, 65475
CLRCHN 802, 62259, 65484
GETIN 810, 61758, 65508
IOBASE 58624, 65523
IOINIT 64931, 65412
LISTEN 60684, 65457
LOAD 816, 62622, 65493
MEMBOT 65076, 65436
MEMTOP 65061, 65433
OPEN 794, 62282, 65472
PLOT 58634, 65520
RAMTAS 64848, 65415
RDTIM 63197, 65502
READST 65031, 65463
RESTOR 64789, 65418
SAVE 818, 62941, 65496
SCNKEY 60039, 65439
SCREEN 58629, 65517
SECOND 60857, 65427
SETLFS 65024, 65466
SETMSG 65048, 65424
SETNAM 65017, 65469
SETTIM 63204, 65499
SETTMO 65057, 65442
STOP 808, 63213, 65505
TALK 60681, 65460
TKSA 60871, 65430
UDTIM 63131, 65514
UNLSN 60926,65454
UNTLK 60911, 65451
VECTOR 64794, 65421
keyboard
current key pressed 203
keyboard buffer 631
keycodes 203
keyboard matrix 245, 655, 56321
last key pressed 197
number of characters in buffer 198
pointer to matrix lookup table 245
reading the keyboard 56320
repeating keys 650
LEFT$ 46848
LEN 46972

LET 43429
light pen 53267, 53268
LIST 42652
LOAD 57704
LORAM 01
MID$ 46903
NEW 42562
NEXT 44318
NMI 65095
ON GOSUB, ON GOTO 43339
OPEN 57790
Operating System (OS)
  OS end of RAM pointer 643
  OS screen memory pointer 648
  OS start of RAM pointer 641
OR 45030
paddle controllers 54297, 54298
paddle fire button 56320, 56321
PAL/NTSC flag 678
PEEK 47117
POKE 47140
POS 45982
PRINT 43680
PRINT# 43648
program text area 43, 2048
program text input buffer 512
RAM
  BASIC pointer to end of RAM 55
  RAM/ROM selection 01
  OS pointer to end of RAM 643
  OS pointer to start of RAM 641
random number generator 54299
READ 44038
registers, reading/setting from BASIC 780
REM 43323
RESET, power-on 64738, 65532
reset, VIC-II chip 53270
RESTORE 43037
RESTORE key, disabling 792, 808
RETURN 43218
RIGHT$ 46892
RND 139, 57495
RS-232
  baud rate 659, 661, 665
  baud rate tables 58604, 65218
  buffers 247, 249
  command register 660
  connector pin assignments 56576, 56577
  control register 659
  duplex mode 660
  handshaking protocol 660
  Kernal ROM routines 57344–65535
  parity 660
  status register 663
  stop bits 659
  word length 659
RUN 43121
SAVE 57686
screen editor
  current character color 646
  cursor color RAM position 243
  cursor flash 204, 205, 207
  cursor maintenance 206, 647
  cursor screen position 209, 211, 214
  insert mode flag 216
  key repeat 650, 651, 652
  quote mode flag 212
  reverse character flag 199
  screen line link table 217
  screen RAM 1024, 648, 53272
  shift flag 653, 654
Serial Bus I/O 56576, 60681-61114
Serial Data Port (CIA) 56332, 56588
SGN 48185
SID chip register 54272–54300
  see also sound
SIN 57960
sound
  ADSR envelope control 54278-54279, 54285-54286, 54292-54293
  filtering 54293-54296
  frequency (pitch) control 54272-54273, 54279-54280, 54286-54287
  gate bit 54276
  Oscillator 3 envelope generator 54300
  Oscillator 3 output 54299
  pulse waveform pulse width 54274-54275, 54281-54282, 54288-54289
  ring modulation 54276, 54283, 54290
  synchronization (hard sync) 54276, 54283, 54290
  volume control 54296
  waveform control 54276, 54283, 54290
sprite graphics
  color registers 53287-53294
  display priority 53275
  enabling sprite display 53274
  horizontal expansion 53277
  multicolor color registers 53285-53286
  multicolor sprites 53276
  position registers 53248-53264
  shape data pointers 2040
  sprite-display data collision detection 53279
  sprite-sprite collision detection 53278
  vertical expansion 53271
SQR 49009
ST (I/O status word) 144
stack, 6510 microprocessor 256

STOP 43055
STOP key 145, 808
STR$ 46181
SYS 780, 57642
TAN 56083
Time of Day clock 56328-56331, 56584-56587
timers, hardware 56324-56327, 56334-56335, 56580-56583, 56590-56591
tokens, keyword 772, 774, 40972, 41042, 41088, 41118, 42364, 42772
User Port 56567-56577
USR 785
VAL 47021
variable
  array variable storage 47
  find or create variable routine 45195
  storage format 45
  string text area 51
VERIFY 57701
VIC-II CHIP
  memory bank switching 56576
  registers 53248-53294
  see also graphics, sprite graphics
warm start, BASIC 40962, 58235
WAIT 47149
wedges 115

GEOS Index (By Memory Location)

alarmSetFlag	34076
alarmTmtVector	33965–33966
alphaFlag	33972
AppendRecord	49801
applicationMain	33947–33948
baselineOffset	38
BBMult	49504
BitmapClip	49834
BitmapUp	49474
BitOtherClip	49861
BldGDirEntry	49651
BlkAlloc	49660
BlockProcess	49420
BMult	49507
BootGEOS	49152
BRKVector	33967–33968
CalcBlksFree	49627
CallRoutine	49624
cardDataPointer	44–45
ChangeDiskDevice	49852
ChkDkGEOS	49630
ClearMouseMode	49564
ClearRam	49528
CloseRecordFile	49783
CmpFString	49774
CmpString	49771
CopyFString	49768
CopyString	49765
CRC	49678
curDirHead	33280–33535
curDrive	33929
curRecord	33942
currentHeight	41
currentIndexTable	42–43
currentMode	46
currentPattern	34–35
currentSetWidth	39–40
Dabs	49519
dataDiskName	33875–33892
dataFileName	33858–33874
day	34072
dblClickCount	34069
Ddec	49525
Ddiv	49513
DeleteFile	49720
DeleteRecord	49795
dirEntryBuf	33792–33821
DisablSprite	49621
diskBlkBuf	32768–33023
diskOpenFlg	33930
displayBufferOn	47
dlgBoxRamBuf	34079–34495
DMult	49510
Dnegate	49522
DoDlgBox	49750
DoIcons	49498
DoInlineReturn	49828
DoMenu	49489
DoneWithIO	49759
DoPreviousMenu	49552
DoRAMOp	49876
DrACurDkNm	33822–33839
DrawLine	49456
DrawPoint	49459
DrawSprite	49606
DrBCurDkNm	33840–33857
driveType	33934–33937
DSdiv	49516
DShiftLeft	49501
DShiftRight	49762
EnableProcess	49417
EnablSprite	49618
EnterDeskTop	49708
EnterTurbo	49684
ExitTurbo	49714
FastDelFile	49732
faultData	33974
FetchRAM	49867
fileHeader	33024–33279
fileSize	33945–33946
fileTrScTab	33536–33791
fileWritten	33944
FillRam	49531
FindBAMBit	49837
FindFile	49675
FindFTypes	49723
FirstInit	49777
FollowChain	49669
FrameRectangle	49447
FreeBlock	49849
FreeFile	49702
FreezeProcess	49426
GetBlock	49636
GetCharWidth	49609
GetDimensions	30988
GetDirHead	49735
GetFHdrInfo	49705
GetFile	49672
GetFreeDirBlk	49654
GetNextChar	49831
GetPtrCurDkNm	49816
GetRandom	49543
GetRealSize	49585
GetScanLine	49468

GetSerialNumber 49558
GetString 49594
GotoFirstMenu 49597
GraphicsString 49462
HorizontalLine 49432
hour 34073
i_BitmapUp 49579
iconSelFlag 33973
i_FillRam 49588
i_FrameRectangle 49570
i_GraphicsString 49576
i_ImprintRectangle 49747
i_MoveData 49591
ImprintRectangle 49744
InitForIO 49756
InitForPrint 30976
InitMouse 65152
InitProcesses 49411
InitRam 49537
InitTextPrompt 49600
inputData 34054–34057
inputVector 33957–33958
InsertRecord 49798
interleave 33932
interruptBottomVector 33951–33952
InterruptMain 49408
interruptTopVector 33949–33950
InvertLine 49435
InvertRectangle 49450
i_PutString 49582
i_RecoverRectangle 49573
i_Rectangle 49567
isGEOS 33931
IsMseInRegion 49843
keyData 34052
keyVector 33955–33956
LdApplic 49693
LdDeskAcc 49687
LdFile 49681
leftMargin 53–54
LoadCharSet 49612
MainLoop 49603
maximumMouseSpeed 34049
menuNumber 33975
minimumMouseSpeed 34050
minutes 34074
month 34071
mouseAcceleration 34051
mouseBottom 33977
mouseData 34053
mouseFaultVector 33959–33960
mouseLeft 33978–33979
MouseOff 49549
mouseOn 48
mousePicData 33985–34048
mousePicture 49–50
mouseRight 33980–33981
mouseTop 33976
MouseUp 49546
mouseVector 33953–33954
mouseXPosition 58–59
mouseYPosition 60
MoveData 49534
NewDisk 49633
NextRecord 49786
numDrives 33933
NxtBlkAlloc 49741
OpenDisk 49825
OpenRecordFile 49780
otherPressVector 33961–33962
Panic 49858
PointRecord 49792
PosSprite 49615
pressFlag 57
PreviousRecord 49789
PrintASCII 30991
PrintBuffer 30982
PrntDiskName 33910–33928
PrntFilename 33893–33909
PromptOff 49822
PromptOn 49819
PurgeTurbo 49717
PutBlock 49639
PutChar 49477
PutDecimal 49540
PutDirHead 49738
PutString 49480
random 34058–34059
ReadBlock 49690
ReadByte 49846
ReadFile 49663
ReadRecord 49804
RecoverAllMenus 49495
RecoverLine 49438
RecoverMenu 49492
RecoverRectangle 49453
RecoverVector 33969–33970
Rectangle 49444
ReDoMenu 49555
RenameFile 49753
ResetHandle 49155
RestartProcess 49414
returnAddress 61–62
rightMargin 55–56
RstrAppl 49726
RstrFrmDialog 49855
SaveFile 49645
saveFontTab 34060–34068
screenColors 34078
seconds 34075
selectionFlash 33971
SetDevice 49840

SetGDirEntry	49648	stringX	33982–33983
SetGEOSDisk	49642	stringY	33984
SetNextFree	49810	SwapRAM	49870
SetPattern	49465	sysDBData	34077
Sleep	49561	TestPoint	49471
SlowMouse	65155	ToBasic	49729
SmallPutChar	49666	turboFlags	33938–33941
spr0pic	35328–35391	UnblockProcess	49423
spr1pic	35392–35455	UnfreezeProcess	49429
spr2pic	35456–35519	UpdateMouse	65158
spr3pic	35520–35583	UpdateRecordFile	49813
spr4pic	35584–35647	usedRecords	33943
spr5pic	35648–35711	UseSystemFont	49483
spr6pic	35712–35775	VerifyRAM	49873
spr7pic	35776–35839	VerticalLine	49441
StartAppl	49711	VerWriteBlock	49699
StartMouseMode	49486	windowBottom	52
StartPrint	30979	windowTop	51
StashRAM	49864	WriteBlock	49696
StopPrint	30985	WriteFile	49657
string	36–37	WriteRecord	49807
StringFaultVector	33963–33964	year	34070