

OsborneMcGraw-Hill

Commodore 128™

PROGRAMMING SECRETS



William M. Wiese, Jr.

Commodore 128™ Programming Secrets

William M. Wiese, Jr.

Osborne McGraw-Hill
Berkeley, California

Osborne McGraw-Hill
2600 Tenth Street
Berkeley, California 94710
U.S.A.

For information on translations and book distributors outside of the U.S.A., please write to **Osborne McGraw-Hill** at the above address.

C-64 and C-128 are trademarks of Commodore Business Machines, Inc. *Commodore 128 Programming Secrets* is not sponsored or approved by or connected with Commodore Business Machines, Inc. Commodore Business Machines, Inc., makes no warranty, expressed or implied, of any kind with regard to the information contained herein, its accuracy, or its completeness.

A complete list of trademarks appears on page viii.

Portions of this book adapted with permission from *Your Commodore 64* by John Heilborn and Ran Talbott (Berkeley: **Osborne McGraw-Hill**, 1986) and *Your Commodore 128* by John Heilborn (Berkeley: **Osborne McGraw-Hill**, 1986).

Commodore 128™ Programming Secrets

Copyright © by McGraw-Hill, Inc. All rights reserved. Printed in the United States of America. Except as permitted under the Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher, with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

1234567890 DODO 89876

ISBN 0-07-881250-X

Jon Erickson, Acquisitions Editor
Greg Berlin, Technical Reviewer
Lyn Cordell, Project Editor

Contents

| | | | |
|-----------|--------------------------------------|---|----|
| I | The Commodore 128's C-64 Mode | 1 | |
| | 1 | Inside C-64 Mode | 3 |
| | | C-64 Mode Selection | 3 |
| | | Initial RAM Bank Selection | 4 |
| | | Other Differences in the C-64 System Map | 5 |
| | | Disk Compatibility Problems: 1541 Versus 1571 | 11 |
| | | A C-64 Bibliography | 14 |
| II | C-128 "Native" Mode | 15 | |
| | 2 | C-128 Architecture and Memory Management | 19 |
| | | What's Where in the C-128 | 21 |
| | | The Banking Concept | 25 |
| | | The C-128 MMU | 28 |

| | | |
|----------|---|-----|
| | The Non-Maskable Interrupt | 48 |
| | The IRQ Interrupt | 52 |
| 3 | C-128 Memory Usage | 57 |
| | Zero-Page and Page 1 Memory Usage | 59 |
| | Map of C-128 Zero-Page and Stack Usage | 60 |
| | Page 1 Memory Usage by BASIC 7.0 and BASIC's DOS Interface | 77 |
| | Low Memory, Pages 2 and 3 | 78 |
| | BASIC 7.0, Kernal, and Screen Editor Page 3 Vector Storage | 80 |
| | Page 3 RAM-Resident Indirect-Load Subroutines | 84 |
| | Bank 0 RAM Usage | 88 |
| | Shadow Register Area for VIC-II and 8563 Video Chips | 91 |
| 4 | The C-128 BASIC 7.0 Interpreter | 119 |
| | C-128 BASIC 7.0 Variable Storage | 120 |
| | Floating-Point Numbers | 126 |
| | BASIC Program Storage | 130 |
| | BASIC 7.0 Dictionary | 138 |
| | BASIC 7.0 Functions | 205 |
| | BASIC Math Functions and the Jump Table | 226 |
| | BASIC String Handling From Machine Language | 233 |
| 5 | The C-128 Video System | 253 |
| | Screen Editor Escape and Control Sequences | 254 |
| | The C-128's 80-Column Video Display System | 262 |
| | The 8563 Independent RAM Block | 263 |
| | The 8563 Registers | 265 |
| | High-Resolution Bit-Mapped Graphics on the 8563 | 273 |
| | User-Defined Character Sets for the 80-Column Text Screen | 276 |
| | The VIC-II Video Chip | 278 |
| 6 | The C-128 Kernal: An Overview | 287 |
| | Vectors | 289 |

| | | |
|------------|--|-----|
| | Kernal Dictionary | 290 |
| | Modified Kernal Routines | 296 |
| | New C-128 Kernal Routines | 300 |
| 7 | Disk and I/O Operations on the C-128 | 313 |
| | 1541 and 1571 Disk Compatibility | 314 |
| | Organization of a 1571 Double-Sided Disk | 316 |
| | The 1571 Drive's CHGUTL Utility Commands | 319 |
| | 1541 and 1571 Drive Internals: The Job Queue | 325 |
| | Autobooting Programs With the C-128 | 338 |
| | Burst Mode Data Transfer With the 1571 Drive | 341 |
| III | The CP/M Operating System | 351 |
| 8 | CP/M on the Commodore 128 System | 353 |
| | Booting CP/M Plus | 354 |
| | Memory Usage by the CP/M Plus Operating System | 355 |
| | Disks and C-128 CP/M Plus | 358 |
| | Transferring Files Between CP/M and Other Modes | 360 |
| | Revisions to the CP/M Plus System | 362 |
| A | DIGIFONT: A New C-128 Character Set | 365 |
| B | C-128 I/O Pinouts | 371 |
| C | Conversion Tables: Trigonometric Functions | 377 |
| D | Character Sets and Graphic Characters | 387 |
| E | Displaying 80-column Text on a Monochrome Composite Monitor | 397 |
| | Index | 399 |

I'd like to dedicate this book to my mother and father. Without their patience and support (and tolerance of cables strewn all over the house), none of this would have been possible.

Acknowledgments

There are many people who made this book “happen.” I am particularly indebted to Greg Berlin, of Commodore Business Machines, for supplying me with armloads of C-128 programming documentation. His review of my text also helped locate several mistakes and “bugs” that could have led many C-128 hackers astray.

I would also like to thank my editor, Jon Erickson, for his time and support — even when I interrupted his lunch hour with a barrage of questions, he never lost his patience. And, to editorial assistant Lynn Heimbacher, a special thanks for the time and energy she put into my book.

Trademarks

The following names are trademarked products of the corresponding companies.

| | |
|--------------------|--|
| C=® | Commodore Business Machines, Inc. |
| Apple® | Apple Computer, Inc. |
| CP/M® | Digital Research, Inc. |
| CP/M Plus™ | Digital Research, Inc. |
| Datassette™ | Commodore Business Machines, Inc. |
| Epson™ | Epson America, Inc. |
| IBM® | International Business Machines Corp. |
| Kaypro™ | Kaypro Corp. |
| Morrow® | Morrow Designs |
| MS-DOS® | Microsoft Corp. |
| OSBORNE® | Osborne Computer Corp. |
| PET® | Commodore Business Machines, Inc. |
| VIC-20™ | Commodore Business Machines, Inc. |

C-128 Programming Secrets Disk Offer

All of the programs and subroutines in *Commodore 128 Programming Secrets* are available on a 5 1/4-inch disk priced at \$14.95. Along with all the subroutines, the disk contains three bonus programs:

- *The 3-Dimensional Plotter* lets you see a three-dimensional plot of any user-defined math expression. This program uses the 640×200 ultrahigh-res mode of the 8563 video chip for sharp resolution. The Plotter is compiled for speed, although source code is given.
- *The Terminal program* includes XMODEM support for uploading, downloading, and Hayes modems. It is compiled for speed.

- *The Disk Sector Editor* supports the 1571 disk drive and allows editing of sectors 36 through 70 of a double-sided disk.

Send your check or money order, along with the return coupon below, to

C-128 Disk Offer
W. M. Wiese, Jr.
118 West 36th Avenue, Apt. D
San Mateo, CA 94403

Please type or print legibly in ink. This will be your return label.

_____ cut along this line _____

Please send me the *C-128 Programming Secrets* companion disk. My \$14.95 payment is enclosed.

Name _____

Address _____

City _____ State _____ ZIP _____

C-128 Disk Offer: W. M. Wiese, Jr., 118 W. 36th Ave. Apt. D, San Mateo, CA 94403

Part I

The Commodore 128's C-64 Mode

The Commodore 128, besides its “native” 128K and CP/M modes, has a built-in C-64 mode. This mode is virtually the same as an ordinary C-64 computer. In fact, Commodore software engineers duplicated the original C-64 BASIC and Kernal ROMs exactly—bugs and all—for the C-128’s implementation of the C-64, simply for the sake of compatibility.

There are, however, some slight differences between the C-64/1541 disk drive combination and a C-128 (in C-64 mode, of course) with a matching 1571 disk drive. The purpose of Part I is to describe these differences, the compatibility problems they may cause, and any cures available for the problems. Since you’re really dealing with a C-128 instead of a true-blue C-64 system, it might be

helpful to browse through Chapters 2 and 3 in Part II of the book so you can understand memory management and the overall architecture of the C-128 computer.

Chapter 1

Inside C-64 Mode

C-64 Mode Selection

Normally, the Commodore 128's "native" 128K mode, along with BASIC 7.0, is activated when the C-128 is turned on. To enter into C-64 mode, you simply type **GO64** in direct mode. You will then see an "ARE YOU SURE?" message, which should be answered **Y** (for yes) or **N** (for no) appropriately. If the **GO64** statement is issued from a running program, no query is displayed and the switch to C-64 mode occurs immediately.

Holding down the **COMMODORE** key during a power-up also

causes a direct switch to C-64 mode. In this case, the Z-80 processor detects the pressed key; for the first few moments after the computer is turned on, the Z-80—not the 8502—is in control of the system. The 8502 conducts a redundant check that also senses whether or not the `COMMODORE` key is pressed.

From a machine language program, a call to the ROM routine at `$FF4D` causes a switch to C-64 mode; in fact, this C-128 Kernal routine is called `C64MODE`. Before a `JSR $FF4D` or a `JMP $FF4D` is issued, however, the C-128's Kernal ROM bank must be selected. (C-128 memory-banking techniques are covered fully in Part II.)

If, upon power-up, a C-64 autostart cartridge has already been inserted in the expansion port, C-64 mode will also be activated. The Z-80 checks for C-64 cartridges first; later, the C-128 system initialization routine (now under 8502 control) will call the `PHOENIX` routine at `$FF56` and check for the presence of any type of cartridge, C-64 or C-128. Again, the C-64 cartridge detection by the 8502 is redundant. Cartridges in the expansion port are recognized as C-64 type if either the `GAME` or `EXROM` lines are held low (that is, grounded).

It should be noted that once the computer is in C-64 mode there is no way to return to C-128 mode; the memory manager unit cannot be accessed in C-64 mode.

Initial RAM Bank Selection

Because the C-128 is a 128K RAM machine, there are two 64K banks of RAM. Normally, a transfer to C-64 mode involves calling the C-128 Kernal routine `C64MODE`. Before this routine is called, the Kernal ROM bank must be selected. Although there are various banks (combinations of ROM, RAM, and I/O devices), the C-128 normally uses bank #15 during its Kernal operations. Bank #15 contains the low 64K (bank #0) of RAM and also contains the Kernal and BASIC ROMs from addresses `$4000` to `$FFFF`. Since the low

64K of RAM is selected prior to the switch to C-64 mode, this RAM bank is the one normally used while in C-64 mode. There is no reason why the upper 64K RAM bank cannot be used; the programmer can set up the MMU (the 8522 memory management unit) configuration register to point to this bank and then call C64MODE.

Other Differences In the C-64 System Map

The C-128's 40-column video chip, called the 8564, is almost exactly the same as its counterpart on the C-64, the VIC-II chip. Because of the extended keyboard on the C-128, three extra output lines are necessary for scanning the numeric keypad and the "outboard" cursor keys, as well as the ESC, TAB, ALT, HELP, LINE FEED, and NO SCROLL keys. The outboard CAPS LOCK and the 40/80 DISPLAY keys are, like the RESTORE key, not part of either the regular or extended key matrix and therefore are not scanned. (Reading these keys will be discussed later.) Additionally, there is a provision to switch between 1 MHz and 2 MHz clock speeds. The C-64 mode of the C-128 can function, with some limitations, with the doubled clock rate. (These clock rates are actually based upon an oscillator frequency of 1.022730 MHz rather than 1 MHz.)

The 8564 (VIC-like) video chip has, besides its usual complement of 47 registers controlling various video output parameters, two extra registers, #47 and #48 (locations \$D02F and \$D030, respectively). These registers are present even while the system is in C-64 mode, and the programmer can take advantage of the extra functions controlled by these two registers.

Register #47 is called the *keyboard control register*. Only the low three bits of this register (bits 2, 1, and 0, denoted as K2, K1, and K0) are used. The remaining bits (7, 6, 5, 4, 3) always return 1's when read; writing data to these upper five bits has no effect. The lower three bits are output pins that function similarly to the output lines on the 6526

| <i>Read from</i> | <i>Keyboard Control Register, \$D02F, outputs</i> | | |
|------------------|---|-----------|-------------------|
| <i>SDC0I</i> | K0 | K1 | K2 |
| Bit #0 | HELP | ESC | ALT |
| Bit #1 | 8 | + | 0 |
| Bit #2 | 5 | — | . (decimal point) |
| Bit #3 | TAB | LINE FEED | UP |
| Bit #4 | 2 | ENTER | DOWN |
| Bit #5 | 4 | 6 | LEFT |
| Bit #6 | 7 | 9 | RIGHT |
| Bit #7 | 1 | 3 | NO SCROLL |

Figure 1-1. C-128 extended keyboard matrix arrangement

VIA I/O chips. With the keyboard control register, however, you don't need to preset a data direction register—all three bits are always regarded as outputs.

The arrangement of the C-128 extended keyboard matrix is shown in Figure 1-1. The alphanumeric portion of the C-128 keyboard (QWERTY...) is identical to the C-64's and need not be shown.

Because extra registers are present in C-64 mode, you can use the C-128 extended keyboard in C-64 mode. A routine using the three extra keyscan lines at location \$D02F is shown in Figure 1-2. This routine wedges itself into the IRQ interrupt routine by diverting the IRQ vector at \$0314/\$0315 to point to the code. Note that to scan a given column in the key matrix (say, that connected to the K0 line), the output line is held *low* (logical zero); matrix columns that are not being scanned should have their respective output lines held *high* (logical one). (This is merely a Commodore convention; another manufacturer's system might scan a keyboard similarly but would

```

CB00 78      SEI          ; Disable interrupts.
CB01 A0 CB    LDY # $CB   ; Direct IRQ vector to code @ $CB0D.
CB03 A2 0D   LDX # $0D
CB05 8C 15 03 STY $0315
CB08 8E 14 03 STX $0314
CB0B 58      CLI          ; Enable interrupts.
CB0C 60      RTS          ; Return to caller.

CB0D A2 F8   LDX # $F8   ; X1111 1000
CB0F 8E 2F D0 STX $D02F ; All 3 extended KB output lines active.
CB12 A9 FF   LDA # $FF
CB14 8D 00 DC STA $DC00 ; CIA #1's DDR set for all 8 lines as inputs.
CB17 CD 01 DC CMP $DC01 ; Not = $FF? Key held down,
CB1A D0 06   BNE $CB22   ; so process it.
CB1C 8D 2F D0 STA $D02F ; Else all KB output lines inactive.
CB1F 4C 31 EA JMP $EA31 ; Jump to normal IRQ KB handler code.

CB22 A9 FB   LDA # $FB   ; X1111 1011 ... K2's column scanned first.
CB24 8D 2F D0 STA $D02F
CB27 A2 00   LDX # $00   ; Zero SHIFT/CTRL/CBM flag.
CB29 8E 8D 02 STX $028D
CB2C A0 08   LDY # $08   ; Counter ... 8 rows to test.
CB2E AD 01 DC LDA $DC01 ; Read input port.
CB31 CD 01 DC CMP $DC01 ; Value changing?
CB34 D0 F8   BNE $CB2E   ; Yes, loop again for another read.
CB36 C9 FF   CMP # $FF   ; Any keys in this column depressed?
CB38 D0 0A   BNE $CB44   ; Yes, process keystroke.
CB3A 18      CLC          ; Clear .C to prepare for add.
CB3B 8A      TXA          ; Get key number (index) and
CB3C 69 08   ADC # $08   ; add 8 since we can skip this column.
CB3E AA      TAX          ; Put index back where it belongs.
CB3F 6E 2F D0 ROR $D02F ; Now K1 held low (X1111 1101).
CB42 D0 E8   BNE $CB2C   ; Zero? No way! We're forcing a jump
; to $CB2C to scan the next column.

CB44 4A      LSR          ; Process keystroke... shift bits into .C .
CB45 90 09   BCC $CB50   ; If .C = 0 key struck, so we quit scanning.
CB47 E8      INX          ; Bump key index counter.
CB48 88      DEY          ; Decrement row counter.
CB49 D0 F9   BNE $CB44   ; 8 rows done? No, then loop again.
CB4B 6E 2F D0 ROR $D02F ; Prepare to scan next column. (Kn=0)
CB4E BD DC   BCS $CB2C   ; If .C=0 we're done no matter what!
CB50 BD 69 CB LDA $CB69,X ; Look up C-64 keycode from table.
CB53 10 07   BPL $CB5C   ; If hi-bit (#7) set, it's <SHIFT>ed,
CB55 A0 01   LDY # $01   ; so we set SHIFT flag.
CB57 8C 8D 02 STY $028D
CB5A 29 7F   AND # $7F   ; And lop off bit#7 to get true C-64 keycode.
CB5C 85 CB   STA $CB     ; Save keycode.
CB5E A6 FF   LDX # $FF
CB60 8E 2F D0 STX $D02F ; Deactivate extended keyscan lines.
CB63 2D DD EA JSR $EADD ; Convert keycode to ASCII byte, place in buffer.
CB66 4C 7E EA JMP $EA7E ; Restore register contents & return from IRQ.

>CB69 40 23 2C 87 07 82 02 40 40 28 2B 40 01 13 20 08
;      0 . up dn Lef rt + - CR 6 9 3

>CB79 40 1B 10 40 3B 0B 18 38 40
;      8 5 2 4 7 1

```

Figure 1-2. Extended keyboard handler code for C-128 in C-64 mode. To activate from BASIC, type SYS 51968. To deactivate, press RUN-STOP/RESTORE.

reverse the process —scanned key matrix column output lines would be held high, while unscanned ones would be held low.)

The routine first holds K0, K1, and K2 low and then reads the \$DC01 input register. If \$DC01 returns \$FF, no key on the extended keyboard has been depressed and processing continues with the normal \$EA31 interrupt-handling code that, among other things, reads the normal C-64 keyboard. If, however, one of the “extended” keys has been struck, the routine sees a value other than \$FF at \$DC01. The routine then individually scans (holds low) K2, K1, and then K0 while incrementing a counter register and testing the contents of \$DC01 for a zero bit. Once this low bit is found, the counter register’s value is used as an index to locate a normal C-64 keycode within a look-up table. The routine does some housekeeping and calls the part of the keyboard handler routine that puts PetASCII characters in the keyboard buffer.

Finally, the code jumps into the tail end of the \$EA31 IRQ code to restore the original 8502 register contents and return from the interrupt. Even though the code runs while in C-64 mode, it can be entered using the C-128’s monitor and then saved. To install this code after loading, simply type **SYS 51968**. The RUN-STOP/RESTORE key sequence will restore normal keyboard operation, as will a call to the Kernal’s RESTOR routine at \$FF8A (65418 decimal).

Because the 40/80 DISPLAY key is connected to the memory manager chip, which is not available to the bus while in C-64 mode, there’s no way to read the status of this key. However, the outboard CAPS LOCK key is connected to the I/O port on the 8502 processor. Bit #6 of C-64 memory location \$0001 (D6510) indicates the status of the CAPS LOCK key. If bit #6 of D6510 is zero, the CAPS LOCK key is pressed. (Remember, CAPS LOCK is a push-on/push-off switch.) When the CAPS LOCK key is not pressed, location \$0001 (in C-64 mode) normally contains \$77; when it is depressed, location \$0001 contains \$37. You should note that these values are valid only when the C-64 has its normal memory configuration: the BASIC ROM, Kernal ROM, and I/O chips are all available to the bus. Other configurations will change D6510’s values appropriately.

The low two bits (1 and 0) of register #48 (at \$D030) control an 8564 chip test mode and the system clock speed, respectively. Bits 7 through 2 of this register are unused.

Bit #1, when set, activates a test of the 8564 chip that causes the screen image to disappear and be replaced by a “test pattern.” Thus, it is of little use except for servicing.

Bit #0 of register \$D030 activates a clock speed of 2 MHz (approximately). Thus, the user can have a “turbo” C-64 that performs at twice the speed. There is a penalty, however: the video screen cannot be displayed while in 2 MHz mode. This is similar to C-128 mode, in which the 40-column screen is only visible in SLOW (1 MHz) mode. No serial bus (that is, disk or printer) operations can take place, either; although the 6526 VIA chips have a 1 MHz clock-signal input, the C-64 Kernal serial bus routines don't function properly at 2 MHz. Even at the higher clock speed, RS-232 (user port) I/O and the 6581 sound chip work just fine—no reprogramming or new control parameters are necessary.

The C-128's second video chip that displays 80-column text, the 8563, occupies locations \$D600 and \$D601. Since these locations in the C-64's I/O bank are not used, there are no compatibility problems with commercial software. Because of the 8563's presence in the C-64's memory map, 80-column screen output from C-64 mode is possible.

Before you can use the 8563 chip in C-64 mode, its registers must be initialized. You can let C-128 mode do this for you by first powering up in C-128 mode with the 80-column display active. The C-128's editor routine, CINT, whose calling address is 49152 (\$C000), sets up the 8563 registers properly. CINT also downloads the ROM character definitions to the 8563's independent 16K of RAM, as well as initializes other editor variables. After C-128 mode displays the “READY” prompt, you can enter C-64 mode ready to use the 8563. If a C-128 power-up is not desired, you must write a program that initializes the 8563 registers properly while in C-64 mode. An 8563 chip register map can be found in Chapter 5 of Part II.

Even though the 8563 chip is ready to be used, the C-64's Kernal

has no way to communicate with it. Unlike the VIC chips, whose text storage area, bit maps, and 48 registers can all be accessed by the 6502 (8502), the system can only communicate with the 8563 via an *address register* and a *data register*. These two registers are located at \$D600 and \$D601, respectively. The address register selects the desired register from the 8563's 37 internal registers, while the data register contains the byte that will be either written to or read from the chip, depending on the operation. Bit #7 of location \$D600 contains a chip

```

C000 A9 00 LDA #$00 ; High byte of 8563 text RAM start addr. ($0000)
C002 A2 12 LDX #$12 ; Select 8563 Register 18 (Update Location, High).
C004 20 1A C0 JSR $C01A ; Send to 8563.
C007 A9 00 LDA #$00 ; Low byte of 8563 text RAM start addr.
C009 A2 13 LDX #$13 ; Select 8563 Register 19 (Update Location, Low).
C00B 20 1A C0 JSR $C01A ; Send to 8563.

C00E A2 1F LDX #$1F ; Select 8563's internal temp. data register.
C010 A0 00 LDY #$00 ; Clear counter.
C012 98 TYA ; .A contains "screen" code (not ASCII!)
C013 20 1A C0 JSR $C01A ; Write to 8563's screen RAM. R18 & R19 incremented.
C016 C8 INY ; Bump counter.
C017 D0 F9 BNE $C012 ; Zero? No, loop again.
C019 60 RTS ; Yes, then return to caller.

; General purpose 8563 write-to-register routine.
; Register # in .X, data to be transferred in .A .

C01A 8E 00 D6 STX $D600 ; Select 8563 register #.
C01D 2C 00 D6 BIT $D600 ; 8563 status bit high?
C020 10 FB BPL $C01D ; Loop until Bit #7 set,
C022 8D 01 D6 STA $D601 ; else store to 8563 data register.
C025 60 RTS ; Return to caller.

; General purpose 8563 read-a-register routine.
; Register # in .X, data obtained in .A .
; NOTE: This particular demo doesn't need this
; routine - it's here merely to show you how
; to read the 8563's registers if you have to.

C026 8E 00 D6 STX $D600 ; Select 8563 register #.
C029 2C 00 D6 BIT $D600 ; 8563 status bit high?
C02C 10 FB BPL $C029 ; Loop until Bit #7 set,
C02E AD 01 D6 LDA $D601 ; else read valid data to .A.
C031 60 RTS ; Return to caller.

```

3

Figure 1-3. Demo program for 80-column chip (8563) use in C-64 mode. To activate, type **SYS 49152**. Note that the characters printed to the screen are in order of Commodore screen code instead of ASCII.

status bit. When this bit is high, the data read from \$D601 is valid. When high, it also indicates that a byte can be written to \$D601—unpredictable results can take place if a write to the 8563 occurs while the status bit (\$D600, bit #7) is low. Thus any routine that reads from or writes to the 8563 should always check this status bit.

Figure 1-3 lists a machine-language demonstration program that contains 8563 read and write routines. It assumes that the 8563 registers have been initialized and that character definitions have been downloaded from ROM to the 8563's independent RAM. The program loads into the C-64's 4K block of unused RAM at \$C000 and is activated by typing `SYS 49152`.

With some “meat” added to this routine (ASCII/screen code conversions, cursor control, and so on), you can create a complete 80-column output routine—or even a full-screen editor—for C-64 mode. See Chapter 4 of Part II for complete 8563 programming details. (Another good source of 8563 programming information is contained in C-128 mode's editor ROM code, located at \$C000 to \$CFFF.) And since the 8563 can function at 2 MHz, many C-64 programs that output text via the \$FFD2 (CHROUT) Kernal vector can use the 80-column display while running at “turbo” speed (of course, this entails diverting the CHROUT indirect RAM vector so that it points to the user-written 80-column driver code).

Disk Compatibility Problems: 1541 Versus 1571

For the average user, the 1541 and 1571 drives are completely compatible; the 1571's DOS (disk operating system) commands are really a superset of those in 1541 DOS. Almost any C-64 program that accesses the 1571 drive will perform flawlessly. Without special programming, however, the C-64 or the C-128's C-64 mode cannot take advantage of most of the enhanced features of the 1571 drive. Programs will still load, save, or execute other disk operations at the same speed as the 1541.

Problems arise when the 1571 drive is not informed of the switch from C-128 to C-64 mode. This occurs when the 1571 and then the C-128 (in "native" mode) are powered up. After the user switches to C-64 mode, the 1571 still is in 1571 mode. The user isn't usually aware of any problems because ordinary LOADs, SAVEs, or OPENs work properly. However, in this state, the 1571 drive controller will often lock up when using copy-protected software, special high-speed disk-copy programs, or the various fast-load programs available for the C-64 that increase the speed of disk access. Some of these programs halt all operation of the C-64 until the drive returns data — if the drive is locked up, then the C-64 itself is also locked up. Only a RUN-STOP/RESTORE combination — or a cold reset — is likely to return control to the user.

To avoid these symptoms, the user should observe a proper power-on sequence for C-64 mode usage. Starting with the C-128 and all serial bus devices off, the user should

1. Turn on the 1571 disk drive(s) first. Any serial bus printer(s) or printer interface device(s) should be turned on after the disk drives are.
2. Turn on the C-128, either with a C-64 autostart cartridge in place or while holding down the COMMODORE key during power-up.

By directly entering C-64 mode, you have avoided the C-128's disk autoboot sequence, which would have told the 1571 that it was connected to a C-128. If, however, you find it necessary to work under C-128 mode first and later switch to C-64 mode without turning the computer off and then on again, you should type in this line after entering C-64 mode:

```
OPEN 15, 8, 15, "U0>M0": CLOSE 15
```

This command will ensure that the 1571 is actually in 1541 mode. The

drive will remain in 1541 mode until it is reset. Alternately, after typing **G064**, you can simply turn the 1571 off and then on again.

Software that directly accesses the disk controller of a 1541 may not function properly on a 1571 even after the above command sequence is entered. This is simply because the 1571's DOS ROM has had to be modified to support the extra features and different hardware in the controller. Commodore engineers have tried to maintain 1541-1571 compatibility as much as possible, but it's almost impossible to ensure complete compatibility. Again, these DOS ROM differences should only affect some copy-protected programs, disk-copy utilities, and fast loaders that have to call machine-code routines in the DOS ROM. Normal, well-behaved programs should not have any trouble whatsoever.

One minor problem occurs when a 1571-formatted double-sided disk is validated (using the DOS V command) while in 1541 mode. If the disk is more than half full (in other words, if there is data on both sides of the disk), a validation done while in 1541 mode will abort with an error. This happens because DOS will have run into an illegal track; track numbers greater than 35 are not recognized by the 1541 DOS. If the disk is not yet half full, the validation while in 1541 mode will take place properly.

One minor hitch occurs when a 1571-formatted double-sided disk is validated on a true 1541 disk drive rather than a 1571 in 1541 mode. During validation, if 1541 DOS encounters any references to sectors on the disk's second side, DOS will abort with an error, just like the 1571 in 1541 mode. However, if the double-sided disk is less than half full, an actual 1541 drive will complete a normal validation. At the end of this validation, when the BAM (Block Allocation Map) is rewritten, the fourth byte (byte #3 of track 18, sector 0) of the BAM sector will be set to zero. A 1571-formatted double-sided disk has its fourth byte of track 18, sector 0 set to \$80 (decimal 128) in order to indicate a double-sided disk. When an actual 1541 changes this byte to zero, the disk becomes single-sided; the user can set this byte back to \$80 to restore the disk's double-sided status. The 1571 in 1541 mode will not change this byte; when a double-sided disk is validated in 1541

mode, but is less than half full, validation will take place completely and the disk will still be double-sided.

A C-64 Bibliography

The following is a list of books useful to those doing C-64/1541 programming:

The Advanced Machine Language Book for the Commodore 64, by Lothar Englisch (Grand Rapids, Mich.: Abacus Software, 1984, 210 pages).

Anatomy of the Commodore 64, by Michael Angerhauser, Achim Becker, Lothar Englisch, and Klaus Gertis (Grand Rapids, Mich.: Abacus Software, 1983, 292 pages).

Commodore 64 Programmer's Reference Guide, Howard Sams, Inc., (Indianapolis, Ind.: 1982, 486 pages).

Inside Commodore DOS, by Richard Immers and Gerald C. Neufeld (Chatsworth, Calif.: Datamost, Inc., 1984, 508 pages).

Inside the 1541 Disk Drive, by Lothar Englisch and Norbert Szczepanowski (Grand Rapids, Mich.: Abacus Software, 1984, 323 pages).

Mapping the Commodore 64, by Sheldon Leemon (Greensboro, N.C.: COMPUTE! Publications, 1984, 268 pages).

Programming the Commodore 64, by Raeto Collin West (Greensboro, N.C.: COMPUTE! Publications, 1984, 609 pages).

Part II

C-128 “Native” Mode

The Commodore 128’s “native” mode is the topic of Part II, which consists of Chapters 2 through 6. Included in these chapters is a description of the computer’s complex but elegant system of memory management.

Chapter 2 also discusses the overall C-128 system architecture, power-up activities, and important operating-system vectors and memory locations. Chapter 3 is bread-and-butter material for any advanced C-128 programmer. Here you’ll find comprehensive memory maps of zero-page and low memory. Also included are maps of the Monitor and Kernal ROMs. Chapter 3 also shows programmers useful places in RAM in which to contain programs, as well as indicates possible RAM-usage conflicts. You’ll also find out how to

use Kernal routines that are not accessible via the standard jump table.

Chapter 4 is devoted to the BASIC 7.0 interpreter. Although BASIC's low memory usage has been "mapped" in Chapter 3, this chapter details how BASIC 7.0 programs are stored. Techniques to speed the operation of BASIC 7.0 programs will be covered, as well as methods of linking machine-code programs to BASIC 7.0 programs. The dark art of using the floating-point routines in the BASIC interpreter to create user-defined math functions—or faster implementations of the ones already included in the ROM—is also covered. In addition Chapter 4 details the complete operation of BASIC 7.0, including text tokenization, line storage, and storage of numeric, array, and string variables.

Chapter 5 is devoted to C-128 text and graphics displays. Because C-128 sound-chip programming is so similar to that on the C-64, little attention is given to C-128 sound generation. The chapter discusses manipulation of the C-128's dual video outputs as well as the programming of the 8563 80-column video chip. Chapter 5 also discusses important screen editor routines, as well as windowing techniques and the use of editor "escape sequences."

Although Chapter 6 is called "Disk and I/O Operations," it really emphasizes 1571 disk drive programming. Here, you will understand the creation of autobooting 1541 and 1571 disk drives, and you'll discover how to make the 1571 drive act like two separate drives. Double-sided disk organization is detailed too, along with 1571 disk commands. You'll also find out how to dump text displays and high-resolution graphics screens to a standard Commodore printer.

Chapter 7 is devoted to the C-128's Kernal operating system. Although a map of the whole Kernal ROM was given in Chapter 3,

Chapter 7 details the standard PET/CBM and C-128-specific Kernal calls and their necessary parameters. Each Kernal routine's function is also described, along with any error returns or register values that change after calling the routine.

Although Part II offers a detailed look at C-128 programming, it really only touches the surface. The C-128 is a remarkably complex and functional machine. A complete memory map of the C-128 could easily occupy a whole book to itself. There's so much material that "explorers" will be busy for the next few years charting the C-128's waters.

Chapter 2

C-128 Architecture And Memory Management

When the Commodore 128 is turned on, it normally functions in C-128 mode. In this mode, sometimes called “native” mode, the programmer can access 128K of bank-switched RAM memory. Commodore also produces 128K and 512K RAM expansion cartridges that communicate with the system via a high-speed DMA (direct memory access) port. The computer is supplied with a 16K ROM that contains a BASIC 7.0 interpreter. Another 16K of ROM

contains the character ROM, screen editor code, and the Kernal, which is the C-128's operating system. This 32K of ROM does not include either the 16K BASIC and Kernal ROM for C-64 mode or the BIOS ROM for CP/M mode.

Both C-128 "native" mode and C-64 mode use the 8502 micro-processor. The 8502 is fundamentally the same as the 6510 processor in the C-64. And, although the 8502's pin arrangement is slightly different than that of the 6510 or 6502, it uses exactly the same machine code and assembly language. The 8502 can run at a 2-MHz clock speed in the C-128's FAST mode. Even the C-64 clock speed can be doubled (see Chapter 1).

The standard C-64 hardware includes the VIC-II sprite color graphics chip and the 6526 I/O timer chips. C-128 mode also supports the 8563 80-column video chip, with its own independent 16K of video RAM that stores the text screen, character attributes, and character definitions. The 8563 chip can also display 640×200 ultrahigh-resolution bit-mapped graphics. Its output can drive either an RGB (red-green-blue) color monitor or an NTSC composite monochrome monitor. The C-128 Kernal operating system includes enhanced serial port driver software for accomplishing high-speed data transfer between the system and the C-128's companion disk drive, the 1571.

The unifying element in the C-128's architecture is the 8722 memory management unit (MMU). This chip is a complex gate-array and register set that is used to select combinations of RAM banks, ROMs, and I/O chips. Other functions of the MMU chip include processor selection (8502 or Z-80), high-speed disk communication with the 1571, and detection of the presence of expansion ROM cartridges.

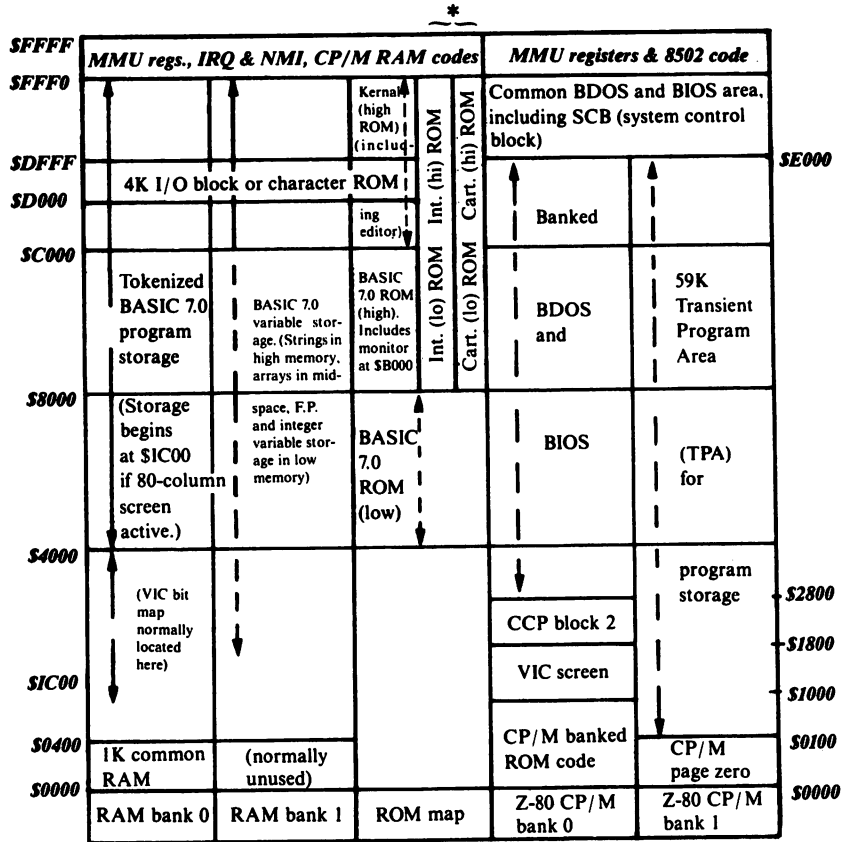
Although this chapter deals with the overall architecture of the C-128 system, it is necessary to give much attention to the programming and use of the MMU chip. Details of memory management techniques also assume a prominent place here.

What's Where In the C-128

Figure 2-1 shows a simple memory map of the C-128 system. Notice that much more space is allotted to system variable storage—locations \$0000 through \$12FF are used by the Kernal and BASIC 7.0. Much of the “fat,” when compared to C-64 low-memory usage, is due to BASIC 7.0’s graphics and DOS commands. Additionally, the Kernal stores several special memory management routines in this part of memory. Although you will soon learn of all the available “banks” of RAM, ROM, and I/O, there are really only two true banks of RAM, numbered 0 and 1. All other “banks” to be discussed are really just combinations of these RAM banks with various layouts of ROM and I/O.

The first 1K of RAM is considered *common RAM*. No matter what memory bank the system is currently using, the C-128 will always have access to locations \$0000 through \$03FF so that it can use important system variables. If common RAM did not exist, the system would get lost (crash) every time an interrupt occurred while in another RAM bank. There also would be no efficient way to communicate between banks if common RAM did not exist. As you will see later, the amount of common RAM can be changed by the programmer for special purposes.

BASIC program storage normally begins at \$1C00 in RAM bank #0 unless BASIC 7.0 causes a bit-mapped graphics image to be displayed. When this occurs, BASIC 7.0 program text is moved so it begins at \$4000 in bank #0. This move allots locations \$1C00 through \$1FFF in bank #0 for color storage and locations \$2000 through \$3FFF for storage of the bit-mapped image. About 60K (when not in one of BASIC 7.0’s graphics modes) is available for program storage because BASIC 7.0 stores variables, arrays, and strings in bank #1. Since over 60K in bank #1 is also allotted for BASIC’s variable storage (no BASIC program is stored in bank #1), there is usually



*These internal and external (cartridge) ROMs are not part of a "stock" C-128 system.

Figure 2-1. Overall memory map of the C-128 system (see Chapter 8 for a more detailed memory map, including 8502 memory usage while under CP/M)

little or no garbage collection delay when you run a BASIC program in C-128 mode. (The term “garbage collection” refers to the rearrangement of strings in memory in order to eliminate old strings that will not be used.)

The BASIC 7.0 ROM begins at \$4000, while the ROM space that contains the Kernal, the monitor, and the screen editor begins at \$C000. In the C-128, ROM often exists directly “over” your own BASIC or machine-code programs. The Kernal also supports the addressing of ROM banks other than those that are already present in the C-128. The system has a socket for an *internal ROM* whose addressing range covers \$8000 through \$FFFF. *External ROM*, sometimes called *cartridge ROM*, can also be added via the expansion port in the back of the machine; this ROM is mapped into the same \$8000-\$FFFF range.

The 4K block from \$D000 through \$DFFF contains the various I/O chip registers. Figure 2-2 shows the layout of this I/O block. The block can be either visible or invisible to the 8502 processor, since it can be “banked” in or out. The programmer can choose RAM space to fill this 4K block or can access the character generator ROM instead. When the high ROM (\$C000-\$FFFF) is selected, there will always be a “hole” at \$D000-\$DFFF. This hole allows direct access to the I/O block or the character generator ROM.

The VIC-II chip occupies locations \$D000-\$D030. This chip has two extra registers, compared to its counterpart on the C-64. The low three bits of the register at \$D02F control three extra output lines used to scan the extended keyboard (the numeric keypad, outboard cursor keys, and so on). The register at \$D030 selects the system clock speed, 1 or 2 MHz. Both these extra VIC-II chip registers are fully explained in Chapter 1. *Ghosts* of the VIC-II chip appear up to \$D3FF. Ghosting signifies that a chip’s registers can be accessed outside of the chip’s normal address range due to the lack of fully decoded address lines.

Locations \$D400 through \$D41C contain the SID (sound interface device) chip registers, while locations \$D500 through \$D50B

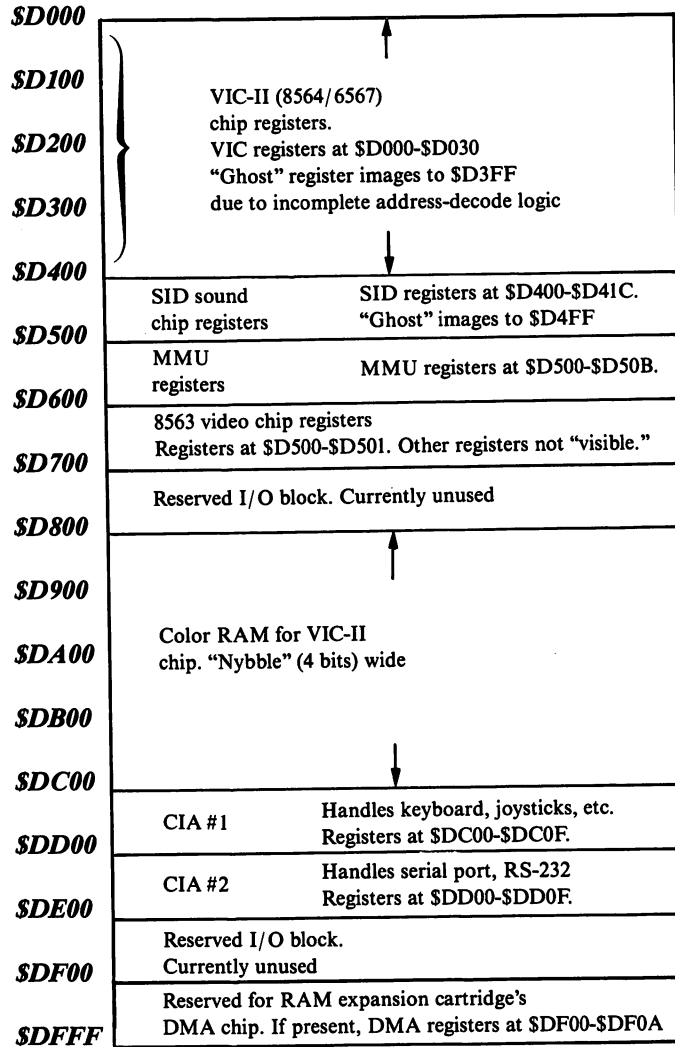


Figure 2-2. C128's 4K I/O block; the processor can replace this block (between \$D000-\$DFFF) with either RAM or character ROM by setting the appropriate bits in the MMU chip's configuration register

contain the 8722 MMU chip registers. And, as in the C-64, locations \$D800-\$DBFF contain the nybble-wide color RAM for the VIC-II chip. Remember, the 8563 80-column chip does not use this color RAM area. The two 6526 CIA (complex interface adapter) chips are located at \$DC00-\$DC0F and \$DD00-\$DD0F, just as in the C-64. The block \$DE00-\$DEFF, called IO1, is still reserved by Commodore for future expansion. The IO2 block, at \$DF00 through \$DFFF, will contain the registers of the 8726 DMA chip (at \$DF00-\$DF0A) if a RAM expansion cartridge is inserted.

One of the most significant features of the C-128's memory map is the appearance of five of the 8722 MMU's registers in the \$FF00-\$FF04 range. No matter what combination of RAM, ROM, and I/O blocks is selected, these five MMU registers will always be visible to the processor. Thus, a program can always access the MMU chip. Otherwise, a program could never change configurations once the I/O block was switched out, since the MMU chip wouldn't be accessible to the 8502.

Also present in any configuration are the interrupt (IRQ), reset, and non-maskable interrupt (NMI) vectors at \$FFFE, \$FFFC, and \$FFFA, respectively. These vectors must always be present in any configuration, because interrupt or reset events may occur at any time.

The Banking Concept

As mentioned before, there are really only two true RAM banks in the C-128. However, the term *bank* can be generalized for the C-128 environment to include any combination of RAM, ROM, and I/O that forms a contiguous 64K block of addresses. For example, the C-128 regards bank #0 as being a contiguous 64K RAM space; bank #1 merely is the other 64K block of RAM. Bank #14 consists partly of RAM in RAM bank #0 along with the BASIC and Kernal ROMs and the 4K character generator ROM starting at \$D000. It should be noted that the MMU chip is always available at \$FF00-\$FF04. Also,

under normal conditions, the lowest 1K of RAM in bank #0 is common to both RAM banks. Other C-128 banks are not composed solely of RAM. For example, the standard system bank (bank #15) consists of RAM for the first 16K. Beginning at \$4000, the rest of bank #15 consists of the BASIC and Kernal ROMs, with the I/O block at \$D000 also engaged. Incidentally, a true RAM bank, ROM, or I/O block is said to be *in context* if it is accessible within the current memory configuration (or bank).

The current memory configuration is determined by the MMU's configuration register at locations \$FF00 and \$D500. This register does not use bank numbers; bank numbers should really be considered "logical bank numbers" that make it easier for a programmer to signify a particular memory configuration. (An analogous situation is the logical file-numbering scheme used in Commodore file-handling operations. This number signifies to the Kernal a given device and secondary address whenever file I/O occurs.) The MMU's Configuration Register, often called the CR, is loaded instead with a bit pattern. A given bit will bank in or out some block of memory. For example, setting bit #0 of the CR to zero will select the I/O block at \$D000-\$DFFF.

Kernal routines that either transfer data between banks or call routines in other banks do use the logical bank-numbering scheme, however. For example, the Kernal INDSTA routine that stores the contents of the 8502 accumulator (.A) to any location in a selected bank requires that bank number to be preset in the 8502's .X register before INDSTA is called. Since a logical bank number is used instead of absolute data to be loaded directly into the MMU's CR, the programmer can avoid the direct selection of the proper MMU configuration register bits by calling a new (that is, not in the C-64 Kernal) Kernal routine, named GETCFG. GETCFG, which has an entry in the Kernal jump table at \$FF6B, returns the proper bit pattern in .A to be written to an MMU CR when given the desired logical bank number in .X prior to the call.

Because all eight bits written to a configuration register control the selection and partitioning of various memory devices, there are

many possible combinations of RAM, ROM, and I/O. Although you will see in an upcoming section how each MMU bit controls memory configuration, Commodore's Kernal in the C-128 recognizes 15 standard, "logical" banks. Table 2-1 lists these logical banks along with the corresponding MMU bit patterns.

Table 2-1 indicates that the C-128 can access four true RAM banks (numbered 0 through 3). The current C-128 only uses RAM

Table 2-1. C-128 Logical Banks and Corresponding MMU Data

| <i>Hex MMU Data Bank #</i> | <i>Selected Devices</i> |
|--------------------------------|---|
| \$3F 0 | RAM bank 0 only. |
| \$7F 1 | RAM bank 1 only. |
| \$BF 2 | RAM bank 2 only. |
| \$FF 3 | RAM bank 3 only. |
| \$16 4 | RAM bank 0, internal ROM, I/O. |
| \$56 5 | RAM bank 1, internal ROM, I/O. |
| \$96 6 | RAM bank 2, internal ROM, I/O. |
| \$D6 7 | RAM bank 3, internal ROM, I/O. |
| \$2A 8 | RAM bank 0, cartridge ROM, I/O. |
| \$6A 9 | RAM bank 1, cartridge ROM, I/O. |
| \$AA 10 | RAM bank 2, cartridge ROM, I/O. |
| \$EA 11 | RAM bank 3, cartridge ROM, I/O. |
| \$06 12 | * RAM bank 0, Kernal ROM, low internal ROM, I/O. |
| \$0A 13 | * RAM bank 0, Kernal ROM, low cartridge ROM, I/O. |
| \$01 14 | RAM bank 0, Kernal ROM, BASIC ROM, character ROM. |
| \$00 15 | RAM bank 0, Kernal ROM, BASIC ROM, I/O. |

*Only the lower half of the internal or cartridge ROM space appears. Thus, in banks 12 and 13 you can only access internal or cartridge ROMs between \$8000 and \$BFFF and cannot access the \$C000-\$FFFF portions of these ROMs. This is because the Kernal and editor ROMs (with either the I/O space or character ROM appearing at \$D000 through \$DFFF) occupy the \$C000-\$FFFF part of the memory map.

banks 0 and 1; RAM banks 2 and 3 are reserved for future implementations of the C-128 that contain more than 128K RAM memory.

The C-128 MMU

As mentioned before, the C-128 uses an 8722 memory management unit (MMU chip). Besides the relatively simple task of selecting appropriate blocks of memory, the MMU possesses many other useful—and necessary—features. During the following discussion of the MMU, please refer to the MMU register map in Table 2-2 whenever necessary.

Table 2-2. C-128 Memory Management Unit Register Map

| <i>Register*</i> | <i>Function</i> | <i>Mnemonic</i> |
|------------------|-------------------------------|----------------------|
| \$D500 | Configuration register | (CR, also at \$FF00) |
| \$D501 | Preconfiguration register A | (PCR A) |
| \$D502 | Preconfiguration register B | (PCR B) |
| \$D503 | Preconfiguration register C | (PCR C) |
| \$D504 | Preconfiguration register D | (PCR D) |
| \$D505 | Mode configuration register | (MCR) |
| \$D506 | RAM configuration register | (RCR) |
| \$D507 | Page 0 pointer, low byte | (P0L) |
| \$D508 | Page 0 pointer, high byte | (P0H) |
| \$D509 | Page 1 pointer, low byte | (P1L) |
| \$D50A | Page 1 pointer, high byte | (P1H) |
| \$D50B | MMU version register | (VR) |
| \$FF00 | Configuration register | (CR, also at \$D500) |
| \$FF01 | Load configuration register A | (LCR A) |
| \$FF02 | Load configuration register B | (LCR B) |
| \$FF03 | Load configuration register C | (LCR C) |
| \$FF04 | Load configuration register D | (LCR D) |

*The MMU registers starting at \$D500 will only be visible to the processor if the I/O block at \$D000-\$DFFF is selected. However, the 8502 can always access the MMU registers at \$FF00-\$FF04 no matter what bank is selected.

The Configuration Register

The MMU's configuration register (CR) directly controls the selection of the various RAM, ROM, and I/O blocks. This CR can be accessed either in the I/O block or in the \$FF00 "hole" that always appears in any possible memory map.

Bit #0 of the CR controls the inclusion of the I/O block at \$D000-\$DFFF in the current bank. If bit #0 is zero, then the I/O block—including the VIC and SID chips, the 8563, the MMU registers, color RAM, and the 6526 CIA chips—is accessible by the 8502 processor. If this bit is high, however, ROM or RAM is mapped into the \$D000-\$DFFF block instead. The selection of ROM or RAM in this space, otherwise occupied by I/O chips, is determined by bits 4 and 5 of the configuration register.

The CR's bit #1 determines whether the processor will access either low (\$4000-\$7FFF) ROM or RAM. If bit #1 is high, RAM is selected; if it is low, the 8502 accesses ROM—in this case, the lower portion of the BASIC 7.0 interpreter. If the block at \$4000-\$7FFF is designated as RAM, CR bits 6 and 7 will select the desired RAM bank.

Bits 2 and 3 of the CR determine which type of memory device will appear in the \$8000-\$BFFF range, which is also denoted as *mid-space*. When both these bits are zero, the upper portion of the BASIC 7.0 interpreter ROM will be accessible to the processor. When bits 2 and 3 are both high, the C-128's mid-space will contain RAM from the RAM bank selected by bits 6 and 7. Bit #2 high causes the internal ROM to appear in mid-space. This ROM is currently not included in the C-128. Bit #3 high selects external cartridge ROM to occupy the \$8000-\$BFFF mid-space. Note that CR bits 2 and 3 only control the selection of the \$8000-\$BFFF portion of internal ROM or external ROM. Bits 4 and 5 will also have to be set appropriately in order to activate the upper portion of either of these ROMs.

The configuration register's bits 4 and 5 function similarly to CR bits 2 and 3, but they also determine the type of memory appearing in the \$C000-\$FFFF high space. When bits 4 and 5 are both set, high space contains RAM. Again, the desired RAM bank is controlled by

CR bits 6 and 7. When these bits are both low, the C-128's Kernal ROM is selected. When only bit #4 is set, the upper portion of the internal ROM appears in the high space. Bit #5 set by itself causes the upper portion of the cartridge ROM to appear in the \$C000-\$FFFF high space.

When bit #0 of the configuration register is zero, the I/O block at \$D000-\$DFFF is banked in, no matter what type of memory is selected by CR bits 4 and 5. The character generator ROM appears at \$D000-\$DFFF, in place of the I/O block, whenever the CR's bit #0 (the I/O bit) is set and when CR bits 4 and 5 enable some type of ROM access.

The desired RAM bank is selected by CR bits 6 and 7. In a C-128, with 128K of system RAM, only CR bit #6 is used. Predictably, when CR bit #6 is zero, RAM bank 0 is selected; otherwise, RAM bank 1 is enabled. The high bit (#7) of the configuration register is not used in current versions of the C-128 and is reserved by Commodore for use in future computers with a larger amount of system RAM. Thus, CR bits 7 and 6 could select up to four 64K RAM banks. Currently, the status of CR bit #7 has no effect.

To see all this in action, let's say you need to access the Kernal and BASIC 7.0 ROMs, as well as the character generator. Also assume that the program being created resides in RAM bank 0. Because you don't want the I/O bank, you will set CR bit 0. CR bit #1 is zeroed, since you will be using the lower half of the BASIC ROM; CR bits 2 and 3 are zeroed, since you also need to access the upper part of the BASIC ROM. And, since you also need to access the Kernal ROM, you should zero CR bits 4 and 5. Finally, zero CR bits 6 and 7 to select RAM bank 0. Thus, a value of \$01 is stored in the MMU's configuration register to appropriately select the memory devices your program needs to access.

It so happens that this particular configuration results in one of the fifteen standard Kernal banks being selected: bank #14 (decimal) in this case. The programmer can create many other valid configurations that are not necessarily Kernal banks. For example, suppose your program must access the C-128's Kernal ROM. Also assume it

needs access to the I/O bank and must use RAM bank 1. The CR's bit #6 must be set high to activate RAM bank 1, while CR bits 4 and 5 must be zeroed to select Kernal ROM, too. Finally, you have to set CR bits 1, 2, and 3 high in order to have the RAM available in both the \$4000-\$7FFF low space and the \$8000-\$BFFF mid-space. Since you also need to access the 4K I/O block as well, the CR's bit #0 should be zeroed. Therefore, the value \$4E should be stored to the MMU's configuration register. This particular memory configuration has no "logical" Kernal bank number associated with it and consequently cannot be referenced with a bank number. Because of this, the GETCFG routine—the one that looks up MMU data when given a logical bank number—will not be of any use. Similarly, other Kernal calls that require a bank number (instead of a raw MMU data byte), such as SETBNK, cannot be used.

If you encounter these problems when you need a custom configuration, you'll have to find a standard Kernal bank number with a configuration as close as possible to yours. Although the program you're creating may use your custom configuration much of the time, you'll have to use the Kernal bank that is similar to yours to actually use many bank-dependent Kernal (and BASIC ROM) routines. Because slight differences between these two configurations might "hide" part of your code or data under a ROM, your program's data structures and the code that must be accessed by the operating system should be placed in RAM, not banked in or out when switching between the two configurations. This situation, however, is a relatively rare occurrence; most C-128 programming tasks do not require the use of "special" banks.

The Preconfiguration And Load Configuration Registers

The MMU chip, besides containing the standard configuration register (CR), also has four preconfiguration registers and four load configuration registers (PCRA-PCRD and LCRA-LCRD). (Refer again to Table 2-2 for a register map of the MMU.) Preconfiguration

registers A, B, C, and D are located from \$D501 to \$D504 in the C-128's I/O block. Their corresponding load configuration registers are, however, located in the \$FF01-\$FF04 "hole." Remember, the processor can always access the MMU at \$FF00-\$FF04, no matter what the system configuration currently is.

This feature of the MMU chip lets the user switch to a predetermined memory configuration already stored in a PCR. To accomplish this switch, your program should complete the following operations:

1. Get the current system configuration by reading the MMU's CR at \$FF00. Store this data in a PCR. If the PCR register is not accessible to the processor (i.e., CR bit #0 is high), clear the CR's bit #0 to enable the I/O block and then store the current configuration byte to a PCR. (For the sake of explanation, we'll use PCRA.)
2. Determine the proper data that should be stored to the MMU's configuration register for the new C-128 configuration. Use Table 2-2 and the material in the section called "The Configuration Register" in this chapter as an aid in selecting the proper memory devices. Do *not* store your MMU data to the CR now!
3. Find a spare preconfiguration register. Store the new configuration to another PCR. (PCRB is used for this example.)
4. Write to the LCR that corresponds to the PCR used in step 3; this will reconfigure the C-128 according to the data in this PCR. A "write" implies that we are not worried about the actual data being stored to an LCR. Thus, STY \$FF02—regardless of what the processor's .Y register contains—causes the contents of PCRB to be transferred to the CR, selecting your second memory configuration.

5. To return to the original configuration prior to this use of the LCR and PCR registers, write to the LCR that corresponds to the PCR containing the “old” MMU data. For this example, an STA \$FF01 instruction will return the system to its original configuration prior to step 1.

Because any one of four banks can be selected by merely writing to an LCR (after its matching LCR has been set up), a simple STA \$FF0x instruction can switch configurations. Since the LCR doesn't care what's written to it, this method saves a substantial number of clock cycles (on the order of 50%) when compared to the direct manipulation of the MMU's configuration register (that is, LDA #\$xx, STA \$FF00).

The C-128's BASIC 7.0 interpreter sets up the the MMU's pre-configuration registers upon either cold or warm starts. (To *cold* start BASIC, use a JMP \$4000 instruction; a *warm* start will take place if JMP \$4003 is executed. The BASIC ROMs must be available to the 8502, of course.) Table 2-3 shows the four main configurations normally used by the interpreter; BASIC 7.0 may also use other configurations occasionally by writing directly to the MMU's CR.

Table 2-3. MMU Preconfiguration Register as Used by BASIC 7.0

| <i>LCR and PCR</i> | <i>PCR Contents</i> | <i>Comments</i> |
|--------------------|---------------------|--|
| \$FF01, \$D501 | \$3F | RAM bank 0 only (no ROM or I/O). |
| \$FF02, \$D502 | \$7F | RAM bank 1 only (no ROM or I/O). |
| \$FF03, \$D503 | \$01 | RAM bank 0, BASIC ROM, Kernal ROM, no I/O. |
| \$FF04, \$D504 | \$41 | RAM bank 1, BASIC ROM, Kernal ROM, no I/O. |

The Mode Configuration Register

The mode configuration register (also called MCR) of the MMU contains bits that control several disparate system functions. As its name implies, it controls operating-system and mode selection. The MCR is located at \$D505 in the C-128's I/O block, and it is really just a specialized I/O port.

Bit #7 of the MCR reflects the status of the 40/80 DISPLAY keyswitch. This bit can only be read, and a write to MCR bit #7 will have no effect. When the 40/80 DISPLAY key is in the 80-column position (down), this MCR bit is zero. If bit #7 is high, it means the 40/80 DISPLAY keyswitch is open (up). It should be noted that the 40/80 DISPLAY key does not actually select different video hardware. Instead, it is merely a one-bit wide I/O port that is tested by the Kernal software upon power-up, a system reset, or after a warm start (that is, a RUN-STOP/RESTORE key sequence).

The MCR's bit #6 determines under which 8502-controlled system mode the C-128 will operate. In "native" mode (and also in CP/M mode), this bit is zeroed, allowing full access to all the MMU registers. C-128 mode allows the MMU chip, along with the whole \$D000 I/O block, to be replaced with either RAM or the character ROM. Although the MMU at times may not be a part of the current bank, it can still be made available to the processor by appropriate banking. However, the MMU cannot ever be accessed by the processor when MCR bit #6 is set. When set, this bit also selects the C-64 BASIC and Kernal ROMs, and configures the C-128 as a normal C-64 system. In this instance, the MMU registers will not appear either in their \$D500 or \$FF00 pages. Other slight differences between a stock C-64 and the C-128's C-64 mode are covered in Chapter 1.

MCR bits 4 and 5 reflect the status of the $\overline{\text{GAMEIN}}$ and $\overline{\text{EXROMIN}}$ (called $\overline{\text{EXROM}}$ on the C-64) lines in the cartridge port, respectively. C-128 cartridges are polled, upon a system reset, by a Kernal routine that examines special ID bytes in the ROM extensions; the operating system does not test the status of the $\overline{\text{GAMEIN}}$ and

$\overline{\text{EXROMIN}}$ lines to sense the presence of a cartridge. If software detects that these lines are zeroed (they are normally active low), it should switch to C-64 mode, since a C-64 cartridge is occupying the expansion port. Because these are true I/O bits, they can also be used as output ports. However, if a C-64 cartridge is present — with active low $\overline{\text{GAMEIN}}$ and $\overline{\text{EXROMIN}}$ lines — one or both of these I/O lines will always read low, and their output functions will, of course, not be significant. It should be noted that the $\overline{\text{GAMEIN}}$ line is used in its output context to control the selection of color RAM (nybble-wide) for split-screen graphics.

Bit #3 of the MCR is referred to as the FSDIR control bit. The FSDIR line functions as a bidirectional I/O line (that is, one not needing a corresponding bit in a data direction register), although only its output function is in the C-128. FSDIR is used to control the fast disk data direction buffer hardware.

Bits 1 and 2 of the MMU are currently unused. When read, they return high, and a write to these two bits has no effect.

The MCR's bit #0 determines which microprocessor controls the C-128 system. When zero, this bit signifies that the 8502 processor is in control, and is normally low upon a reset. The Z-80 microprocessor is selected when bit #0 is high. When the Z-80 is active, the system uses a "translated" memory map; all accesses to locations \$0000 through \$0FFF are translated to the C-128 system's I/O block locations \$D000-\$DFFF.

The mode configuration register is not represented in the MMU's alternate registers at \$FF00-\$FF04. Thus, to read and write to the MCR, the MMU, with the rest of the I/O block, must be visible to the processor. Remember, the I/O block can be selected by zeroing bit #0 of the MMU's configuration register at location \$FF00. Even advanced programmers probably won't have much cause to use the MCR, because almost all C-128 programs run under one mode (C-64, C-128 "native," or CP/M), the operating system detects the presence of C-64 ROM cartridges, and other Kernal software handles the FSDIR bit.

The RAM Configuration Register

Location \$D506 in the C-128's I/O block contains the RAM configuration register, also called the RCR. This MMU register controls the allocation and extent of common RAM memory and relocation of zero-page RAM memory. The RCR also selects one of the two possible 64K RAM banks that the VIC-II text/sprite graphics chip can access.

Normally, the lowest 1K (locations \$0000-\$0400) of RAM is "common" RAM. The lowest 1K of RAM is, in reality, a part of RAM bank 0 no matter what RAM, ROM, and I/O devices form the current bank. Thus, even if RAM bank 1 is being accessed, all memory accesses in the lowest 1K of RAM really occur in RAM bank 0. Another way of thinking of common RAM is to consider RAM bank 0 as always being selected whenever the memory address falls within certain limits, despite possibly contrary RAM bank-selection information contained within the MMU's configuration register. This RAM "share" is done by necessity; the system has to keep track of important system variables no matter what configuration the system is currently accessing. In the C-128, common RAM is always located within RAM bank 0; furthermore, common RAM usage can be completely disabled for specialized applications.

RCR bits 0 and 1 determine how much RAM is common to any C-128 memory configuration. Table 2-4 shows how these low two bits control common RAM allocation, while Table 2-5 shows how RCR bits 2 and 3 control the positioning of common RAM within the 64K address space.

Although current versions of the C-128 contain an MMU chip that only handles 128K of system RAM, future versions of the MMU may be able to address up to 1 megabyte of RAM, in 256K blocks. RCR bits 5 and 4, respectively, can together be regarded as a two-bit number that selects the first (00), second (01), third (10), or fourth (11) 256K block of RAM (see Table 2-5). However, bits 4 and 5 return zeroes on current versions of the 8722 MMU chip, and a write to

Table 2-4. Common RAM Allocation With RAM Configuration Register

| <i>RCR</i> | <i>Bit#</i> | <i>Function</i> |
|------------|-------------|-----------------|
| 1 | 0 | |
| 0 | 0 | 1K common RAM. |
| 0 | 1 | 4K common RAM. |
| 1 | 0 | 8K common RAM. |
| 1 | 1 | 16K common RAM. |

Table 2-5. Common RAM Positions

| <i>RCR</i> | <i>Bit#</i> | <i>Function</i> |
|------------|-------------|---|
| 3 | 2 | |
| 0 | 0 | No common RAM allocated. |
| 0 | 1 | Shared RAM at bottom of memory. |
| 1 | 0 | Shared RAM at top of memory. |
| 1 | 1 | Shared RAM at bottom and top of memory. |

these two RCR bits does not have any effect.

The VIC-II video chip accesses one of the C-128's 64K RAM banks as determined by RCR bits 7 and 6. Currently, bit #7 is ignored, leaving RCR bit #6 to select RAM bank 0 when bit #6 is zero, and RAM bank 1 when bit #6 is high. In future computers with more system memory, RCR bit #7 will function along with bit #6 to select the RAM bank for VIC-II access, selecting one of four (RAM0 . . . RAM3) RAM banks for VIC-II text, bit-map, and sprite data storage. Programmers should note that the RAM bank the VIC-II chip uses can be allocated independently of the bank the processor

currently accesses. Additionally, this bank selection does not affect VIC-II programming itself—this video chip can only access a 16K block of memory at once, so the programmer will still have to select the desired 16K segment. This selection is accomplished by properly setting the low two bits (0 and 1) of CIA#2's port A I/O register at \$DD00 (56576 decimal).

The Page-Pointer Registers

One of the most powerful features of the C-128's MMU chip is the set of four *page-pointer* registers, P0L, P0H, P1L, and P1H located at \$D507 through \$D50A. (Refer to Table 2-2 again for the register map of the MMU.) These registers allow relocation of pages zero and one. There are two reasons why relocation of these first two pages is helpful. First, instructions addressing zero-page memory require fewer clock cycles; their use results in shorter routines because only one byte is needed for a machine instruction's operand. Zero-page memory, however, is at a premium because there are only 256 bytes available to this particular addressing mode. Second, the stack in 6500-series processors is "fixed"—it can only exist in page one—and is only 256 bytes long also.

Relocating these two pages in memory allows the creation of independent, hardware-accessible (via PHA, PLP, and so on) stacks and zero-page regions of memory. In fact, each major block of a large program can have its own zero-page and stack areas. However, the programmer must first find some free RAM and set it aside exclusively for zero-page and/or stack usage. When either or both of these pages have been "relocated," all reads and writes to *true* pages 0 and 1 are exchanged with reads and writes to pages x and y. Once these pages have been moved, a read or write operation that directly accesses pages x and y, using their original, normal addresses instead of zero-page or stack addresses, will really be accessing *true* zero-page or stack RAM areas. Thus, the new and original pages are

exchanged — the page is not simply relocated. This can be dangerous, because programs can destroy important zero-page system variables and/or the normal stack if write operations inadvertently access “true” addresses that are now located within either the “new” page zero or page one.

It is very important for the programmer to know when relocated pages are in use, and to avoid normal (untranslated) memory accesses to these affected memory regions. For example, if the program is using a “new” stack that has been relocated to the \$EF00-\$EFFF region, it should avoid direct accesses to this region (that is, only use stack-manipulation instructions) to avoid destroying the normal system stack at \$0100-\$01FF.

The actual location of page zero is determined by the MMU’s P0L and P0H registers at \$D507 and \$D508. Bit #0 of the P0H register is the only active bit; when high, page zero will be contained in the high RAM bank (RAM bank 1). When clear, P0H bit #0 will locate page zero within RAM bank 0. The byte contained in P0L selects the desired page within the particular RAM bank. Thus, the high-order byte of the address where the relocated zero-page memory begins should be stored in the P0L register. For example, suppose it is necessary to create a “new” page zero that is actually located at \$CD00 in RAM bank 1. Figure 2-3 shows how this is done. Remember, the I/O block must be in context before MMU access occurs.

The page-one relocation registers, P1L and P1H (at \$D509 and \$D50A, respectively), behave exactly like the P0L and P0H registers,

```

LDX #$01      ; Selects RAM bank 1.
LDY #$CD      ; "Zero-page" begins at $CD00.
STX $D508     ; Store to P0H.
STY $D507     ; Store to P0L.

```

Figure 2-3. An example of page relocation program

except that the 8502 stack area is “moved” instead of zero-page RAM. At power-up, the C-128’s Kernal sets the P0L/P0H and P1L/P1H register pairs to point to true pages zero and one, respectively. In either case, page switching does not take effect until the low-order data has been written to the P0L register. Thus, the P0H register should be written to first, in order to select the desired RAM bank. It should also be mentioned at this point that non-RAM devices — ROMs or I/O chips — can appear in a relocated page zero or one. Except for extraordinary circumstances, the appearance of ROM in zero-page or stack regions will most likely result in a system crash; if ROM appears in a stack area, subroutine return addresses and important operating system variables cannot be saved. In a similar manner, I/O register contents might be destroyed when overwritten with zero-page or stack data.

Nevertheless, manipulating I/O devices by using page relocation techniques can be a powerful tool in the programmer’s bag. For example, extremely fast (using fewer clock cycles) color manipulation can take place if color RAM is accessed by treating parts of it as zero-page data. (Since color RAM is only four bits wide — nybble wide — it would be difficult to treat it as stack-accessible RAM, because subroutine calls would probably be necessary. The nybble-wide RAM obviously cannot properly store the proper pending return addresses.) This method can save several clock cycles per memory access, possibly shaving dozens or even hundreds of clock cycles off the time required by a complex graphics operation.

The System Version Register

The system version register, mnemonically referred to as the VR, is located at \$D50B in the I/O block. This register contains a special code that indicates the MMU version and also indicates how much memory is available to the system. The lower four bits (3 through 0) contain the MMU version number. The upper nybble, consisting of

VR bits 7 through 4, indicates the number of 64K RAM blocks that are available to the system. Since the C-128 currently has 128K of RAM and current-production C-128s are using Revision 2 of the MMU chip, the MMU's version register will return a value of \$22. This register can only be read—writing to it has no effect. The memory data contained in the VR does not take into account the 16K of independent video RAM for the 8563 video chip, or the 1024 nybbles of color RAM.

Programs that can take advantage of the extra RAM of future C-128 systems (that is, RAM directly available to the processor, not the DMA-accessible RAM in an expansion cartridge) should check the version register and adjust memory usage accordingly.

The SYSTEM Vector

Most Commodore computers, based on the 6502 or a derivative, have three important vectors located at the very top of the memory map. These are the \$FFFA/\$FFFB NMI non-maskable interrupt vector, the \$FFFC/\$FFFD system reset vector, and the \$FFFE/\$FFFF IRQ interrupt vector. C-128 systems also have a third vector, equally important, called the *SYSTEM* vector. (All these vectors are stored in standard 6502 low-byte/high-byte form.)

Unlike its three other companions, the *SYSTEM* vector is not set up by the hardware in the 8502 processor, but is merely an important Kernal vector. Additionally, *SYSTEM* is contained only in RAM bank 1, while the other three vectors are located in ROM and all available RAM banks. The *SYSTEM* vector is preceded by a three-byte key sequence containing the PetASCII characters “CBM” (\$43, \$42, \$4D hex) at locations \$FFF5 through \$FFF7 in RAM bank 1. The key string is used to indicate whether or not the system has just been powered for the first time or is merely being reset; if the “CBM” string is not present, the Kernal assumes that the system is being cold-started.

Redirecting the SYSTEM vector is often useful in recovering a crashed system. If, for example, a program disturbs the proper operation of either screen editor, the sequence listed in Figure 2-4 will restore a normal screen display when the system is reset and return control to the BASIC 7.0 interpreter, providing that BASIC and Kernal operations have not been disturbed. This is especially useful when using 640×200 ultrahigh-resolution graphics with the 8563 80-column video chip, because the RAM-resident character set is overwritten by the graphics bit map and the contents of several 8563 registers have changed. (If the current program crashes and control returns to the user, there is often either no screen display or just garbage characters floating around the screen.) Additionally, this sequence ensures that the system will read the keyboard properly by restoring the original keyboard look-up tables.

In fact, the technique of modifying the SYSTEM vector can add a degree of protection to software. Some commercial software packages for the C-128 redirect this vector to point to a memory-clear routine or to a loop that hangs the system. In this case, the only way to recover is to turn the C-128's power off and then on again.

Whenever power is applied to a 6502-based computer system, execution passes through the reset, or cold-start, vector at \$FFFC. The reset switch on the side of the C-128 is directly connected to the 8502's $\overline{\text{RES}}$ line and holds it low when depressed. The $\overline{\text{RES}}$ line is shared between the Z-80 processor and the 8502 in the C-128; in fact,

```

>1FFF8 00 0B      ; Point SYSTEM to our patch code at $0B00.
A 0B00 LDA #$01    ; Use INIT STATUS to indicate editors need to
      STA $0A04   ; be rescued but to leave BASIC untouched!
      JSR $FF84   ; Call the Kernal IOINIT routine.
      JSR $C000   ; Call CINT to reset both screen editors.
      JMP $4003   ; Jump into BASIC 7.0 - warm start!

```

Figure 2-4. Patching the SYSTEM vector allows the screen editor to be reset

the Z-80 first takes control of the system during a reset before the 8502 does.

After the 8502 gains control of the system, it places all ROMs—Kernal, BASIC and monitor—as well as the I/O block in context. Immediately afterward, IRQ interrupts are disabled and the 8502's stack pointer is reset to point to the top of the stack. The MMU chip is then initialized, meaning that preconfiguration registers A, B, C, and D (\$D501-\$D504, respectively) are initialized.

Next, code that resides in the lowest 1K of RAM—known as common or shared RAM—is downloaded from ROM. Table 2-6 lists these routines in order of their calling addresses. RAM-resident IRQ- and NMI-code is also downloaded.

Up to this point, the programmer has no control over the progress of the C-128's initialization. However, an internal Kernal routine (mnemonically referred to as SECURE) checks the SYSTEM vector and can branch to user-written code if the “CBM” keysting at \$FFF5 is valid. If this string isn't valid, SECURE assumes a cold start is occurring.

After the execution of the SECURE routine has been completed, the Kernal's POLL routine (not available as a Kernal jump-table entry) checks for the presence of ROM cartridges. POLL first checks for the presence of a C-64 cartridge by testing whether or not the GAMEIN or EXROMIN expansion port lines are held low. If this test indicates a C-64 cartridge is present, the system completes a switchover to C-64 mode and begins a normal C-64 initialization process. As mentioned in Chapter 1, this test for C-64 cartridges is actually redundant—whenever a system reset occurs, the Z-80 processor is initially in control of the C-128 and also checks the status of the GAMEIN and EXROMIN cartridge port lines.

POLL will search for any C-128-type ROM cartridges if it doesn't find any C-64-type ROMs. Unlike those for the C-64, C-128 autoboot and “function” ROMs do not announce their presence to the Kernal by using any hardware techniques (no I/O lines are held low). There are four possible external ROM setups, two internal and two external, and the POLL routine always searches each of these during the power-up sequence for a special header that appears at the

Table 2-6. Operating-System Routines* Resident in Common RAM

| <i>Address</i> | <i>Mnemonic</i> | <i>Function</i> |
|----------------|-----------------|---|
| \$02A2 | FETCH | Cross-bank routine imitating LDA (addr),Y. |
| \$02AA | FETVEC** | Vector for FETCH contains address of page-zero pointer. No user setup necessary. |
| \$02AF | STASH | Cross-bank routine imitating STA (addr),Y. |
| \$02B9 | STAVEC** | Vector for STASH contains address of page-zero pointer. User setup required before using STASH. |
| \$02BE | CMPARE | Cross-bank routine imitating CMP (addr),Y. |
| \$02C8 | CMPVEC | Vector for CMPARE contains address of page-zero pointer. User setup required before using CMPARE. |
| \$02CD | JSRFAR | Cross-bank routine imitates JSR subroutine call. |
| \$02E3 | JMPFAR | Cross-bank imitation of JMP instruction. |
| \$039F | INSRA0 | Fetch-data-from-RAM subroutine; used by BASIC 7.0. |
| \$03AB | INSRO1 | Fetch-data-from-ROM subroutine; used by BASIC 7.0. |
| \$03F0 | EXEDMA | Routine called by DMA_CALL that executes a DMA device command. |

*These routines are not used by the Kernal. Instead, they are used by BASIC to perform a cross-bank fetch.

**Not an actual RAM-resident routine, but a vector directly used by RAM-resident routines.

start of the ROM block. POLL sequentially checks the external low ROM, the external high ROM, the internal low ROM, and then the internal high ROM. The low ROMs, which start at \$8000, can be either 16K or 32K long. The high ROMs, which start at \$C000, obviously can only be 16K long. These ROMs contain a keystack “CBM,” preceded by a special identifier byte and two ROM entry addresses. This header sequence is listed in Table 2-7. All expansion ROM identifier bytes must be non-zero—each ID byte is stored in the physical address table (PAT) at \$0AC1-\$0AC4. Thus, routines in installed ROMs can recognize each other’s presence by checking for non-zero entries in the PAT. Additionally, an identifier byte of one (\$01) indicates that the device is an autostart ROM, and execution immediately passes to the cold-start entry for that ROM. CURBNK, located at \$0AC0, contains the current expansion ROM number, letting the ROM find out where it’s located. This is useful for resident code that may have to download itself into RAM and relocate periodically.

Table 2-7. C-128 ROM Header Structure

| <i>Start Address*</i> | <i>Length</i> | <i>Contents</i> |
|-----------------------|---------------|--|
| offset + 0 | 3 bytes | ROM cold-start entry, JMP \$xxxx format. |
| offset + 3 | 3 bytes | ROM warm-start entry, JMP \$xxxx format. |
| offset + 6 | 1 byte | ROM identifier byte, \$01-\$FF. |
| offset + 7 | 3 bytes | “CBM” keystack—\$43, \$42, \$4D hex. |

*The value of “offset” can be either \$8000 or \$C000, the respective start addresses of low and high expansion ROM.

It should be noted that the Kernal makes no use of a C-128 ROM's warm-start entry. This entry is then optional, and the user has five bytes in which to pre-load a register, disable interrupts (in case the cold-start code is called later), turn on decimal mode, and so on. Other ROMs that do not contain an autostart identifier byte are later called by the Kernal routine PHOENIX (\$FF56) after the BASIC 7.0 interpreter has been initialized.

The Kernal routine IOINIT (\$FF84) is called after POLL finishes its duties. IOINIT is really the workhorse of the reset sequence, configuring both VIC-II and 8563 video-chip registers, both 6526 interface-chip register sets, the SID sound-chip registers, and so on. The contents of the INIT__STATUS byte at \$0A04 distinguish a true cold reset (when power has just been applied to the C-128 system) from its warmer counterpart (when the reset switch is pressed or the RESET code vectored through \$FFFC is called). During a true cold start, INIT__STATUS is first loaded with \$00; various bits of INIT__STATUS are set as the power-up sequence progresses. However, the SYSTEM vector and its "CBM" keysting are valid—and the value of INIT__STATUS is preserved instead of being set to \$00—whenever the C-128 is warm-started by pressing the reset switch or calling the RESET routine.

IOINIT can prevent a complete system initialization from occurring. For example, bit #7 of location \$0A04 is set after the 80-column video chip's character set has been downloaded from ROM to the 8563's independent video RAM. Any future calls to the RESET routine will not overwrite the character set currently in this 16K of RAM; thus, custom character sets—or a 640 × 200 ultrahigh-resolution display—can be preserved even after a system reset. (Of course, this does not apply if the power has been removed even momentarily!) INIT__STATUS' bit #6 determines whether or not both of the screen editors will be fully initialized by the CINT (\$FF81 or \$C000) routine. If clear, a full initialization takes place; however, if bit #6 is set, some editor initialization occurs but the keyboard decode matrix pointers are not reset and the programmable function

keys are not reassigned. The RUN-STOP/RESTORE key sequence, for example, calls both IOINIT and CINT routines without completely resetting I/O devices and editors, due to INIT__STATUS' bits #6 and #7 being set during normal system operation.

The Kernal, after calling IOINIT during a reset sequence, checks to see if the RUN-STOP or COMMODORE key has been pressed. If the RUN-STOP key is pressed during a system reset, control will later be passed to the monitor instead of BASIC. If the COMMODORE key is pressed, the C-128 will later switch over to C-64 mode by calling the Kernal's GO64 routine.

The Kernal's RAMTAS routine is the next call in the reset sequence. RAMTAS clears zero-page RAM, allocates the RS-232 and cassette buffers, and sets pointers to the top and bottom of RAM bank 0 (MEMSIZ, \$0A07/\$0A08 and MEMSTR, \$0A05/\$0A06). RAMTAS does not test RAM for bad storage locations, but it does set an important flag byte, DEJA-VU, located at \$0A02. DEJA-VU indicates to other Kernal routines that system RAM and important pointers have been initialized; it also indicates that the SYSTEM-VECTOR (not the SYSTEM vector!) is valid. This two-byte vector at \$0A00 is used by the RUN-STOP/RESTORE key sequence and normally points to BASIC 7.0's warm-start entry at \$4003. During the reset sequence, however, the SYSTEM-VECTOR points to BASIC's cold-start entry at \$4000.

After RAMTAS concludes its operations, control is next passed to the Kernal RESTOR (\$FF8A) routine. RESTOR sets up all the Kernal indirect vectors that are located in the \$0300 area of low RAM. However, it does not initialize any screen editor or BASIC 7.0 indirect vectors and does not affect the SYSTEM vector or the SYSTEM-VECTOR. The execution of RAMTAS will not take place in future resets because the DEJA-VU flag has been set. Thus, holding down the RUN-STOP key while triggering a warm reset in order to enter the monitor will enable the user to examine the system memory "untouched" after a crash. RESTOR returns to the reset code sequence, which in turn calls the CINT (\$FF81 or \$C000) screen

editor reset routine. Remember, CINT depends on the INIT—STATUS flag byte to determine if a complete editor reset is necessary.

On return from the CINT routine, the IRQ interrupt source is enabled, except in some foreign implementations of the C-128. Control will then be passed to the GO64 code, BASIC 7.0, or the monitor. And, once BASIC 7.0 is initialized, the Kernal PHOENIX (\$FF56) routine will call any non-autostart cartridges; additionally, BOOT—CALL (\$FF53) will query the disk drive for an autoboot disk by examining the contents of track 1, sector 0.

The Non-Maskable Interrupt

Non-maskable interrupts (NMIs) are vectored through locations \$FFFA/\$FFFB. Only two events can trigger a non-maskable interrupt: pressing the RESTORE key or handling RS-232 data I/O. The first few bytes of the NMI-handler code are also contained in all available RAM banks; an NMI event can occur while the C-128 is in one of its many memory configurations, so some code (residing in RAM and high ROM) must always be accessible to the processor. Whenever an NMI occurs, the following functions take place:

- 1.** Ordinary maskable interrupt requests (IRQs) are disabled.
- 2.** The contents of the 8502's .A, .X, .Y, and .P (status) registers are pushed onto the stack, along with the current memory configuration.
- 3.** The “system bank” is enabled—all ROMs, the 4K I/O block, and RAM bank 0 are in context.
- 4.** The execution path of the NMI routine then passes through the RAM vector (mnemonic: INMI) at \$0318/\$0319.

Normally, the INMI indirect vector points to the remaining ROM-resident code of the NMI routine. This code first clears the

8502 decimal-mode flag; it then determines the cause of the interrupt by examining the contents of the interrupt control register in CIA #2. (This register is often referred to as D2ICR, and is located at \$DD0D in the I/O block.) The ICR is cleared automatically upon a read operation on this register; the NMI line will be held low by CIA #2 until the ICR is read — providing that CIA #2 actually caused the NMI event in the first place.

If the NMI handler determines that the CIA caused the NMI, control is passed to the RS-232 serial I/O data handling routines in the Kernal ROM. These routines assemble the serially received data bits into a usable data byte, or sequentially transmit the bits that form a character. This software technique is employed instead of using a true ACIA chip. (ACIA stands for Asynchronous Communications Interface Adapter.) In fact, these routines actually emulate a 6551 ACIA chip, at data rates up to 1200 bits/second.

If the ICR indicates that CIA #2 was not responsible for the NMI event, then a RESTORE keystroke is assumed to have occurred. The RESTORE keystroke is the only other occurrence that can trigger an NMI, with the exception of C-128 systems that have special “out-board” I/O devices attached to the user or expansion ports. The NMI code then checks to see if the RUN-STOP key is pressed simultaneously with the RESTORE key; if not, then the 8502 register contents and memory configuration are restored with their original contents prior to the NMI event. An RTI instruction returns control to the original code being executed.

However, if the RUN-STOP keystroke occurred during the NMI, “warm-start” code is executed. First, the Kernal RESTOR (\$FF8A) routine is called to restore all Kernal (not BASIC 7.0 or screen editor) indirect RAM vectors in the \$0300 area of common RAM to their original cold-start values. IOINIT is called next, followed by a call to CINT to reset the screen editors. Since NMIs occur only when the system has been fully initialized, the INIT__STATUS byte at \$0A04 normally maintains bits #7 (for IOINIT) and #6 (for CINT) set high to prevent “full” initializations. Thus, 8563 RAM-based character defi-

nitions are not downloaded from ROM, and pointers to keyboard-decode-matrix tables are not changed. Programmable key definitions are not destroyed, either.

On return from CINT, control passes through the SYSTEM-VECTOR at \$0A00. In a fully initialized C-128 system, this vector normally points to \$4003, which is BASIC 7.0's "warm start" entry. Upon restoration of the 8502 .A, .X, .Y, and .P register contents and the original memory configuration before interruption, execution returns to the original code. Figure 2-5 lists the C-128 NMT handler's assembly code.

```

$FF05  SEI          ; disable interrupts.
        PHA          ; push .A, .X, & .Y register
        TXA          ; contents onto stack.
        PHA
        PHA
        PHA
        LDA $FF00    ; get MMU configuration data
        PHA          ; push it on stack, too!
        LDA #$00     ; select "system" bank (RAM0, ROM, I/O)
        STA $FF00
$FF14  JMP ($0318)    ; jump thru RAM-based INMI vector to $FA40.
        ; This is one place where the user can
        ; divert NMI event handling!
        ; *****
****   *****
$FA40  CLD          ; ensure 6502 decimal mode flag is clear.
        LDA #$7F
        STA $DDDD    ; disable NMI's by clearing D2ICR.
        LDY $DDDD    ; preserving latched ICR contents in .Y.
        BMI $FA5F    ; branch to RS-232 handler if NMI came from CIA#2.
        JSR $F63D    ; select last row of keyboard matrix and
        JSR $FFE1    ; check if <RUN-STOP> keystroke occurred;
        BNE $FA5F    ; if not, still go thru RS-232 handler!
        JSR $E056    ; call Kernal RESTOR routine directly.
        JSR $E109    ; call Kernal IOINIT routine directly.
        ; (Ordinarily, programmers should use the $FF8A
        ; and the $FF84 jump vectors for RESTOR and
        ; IOINIT, respectively.)
        JSR $C000    ; call CINT to initialize screen editor.

        JMP ($0A00)  ; go thru SYSTEM-VECTOR - normally points to
        ; BASIC 7.0 "warm-start" entry at $4003.

$FA5F  JSR $E805    ; else, call Kernal RS-232 I/O handler.
        JMP $FF33    ; pull old A/X/Y/config. contents off of stack,
        ; and return from the NMI.

```

Figure 2-5. The C-128 NMI handler routine

```
A 0B00 LDA #$08      ; device #8
   0B02 JSR $FF4A    ; CLOSE-ALL - closes all files on device #8.
   0B05 JMP $4003    ; jump to BASIC 7.0's "warm-start" entry.

>0A00 00 0B        : point SYSTEM-VECTOR to new code.
```

Upon exit from the monitor, this routine is called by the RUN-STOP/RESTORE key sequence.

Figure 2-6. An example of the diversion of the SYSTEM-VECTOR

Although NMIs are considered non-maskable by the 8502 processor, they can still effectively be disabled in a C-128 system because of assistance provided by the CIA chip. User code can bypass the reading of CIA #2's interrupt control register, resulting in its never being cleared. This action — or rather, inaction — causes CIA #2 to hold indefinitely the 8502 processor's NMI line to ground. Since the NMI line on 6500-family processor chips is negative-edge-triggered (meaning that only high-to-low transitions on the 8502's NMI pin cause interrupts), any further NMI events are ignored.

Two methods exist to “patch” into the NMI handler code. One is similar to the technique used on earlier Commodore machines: diverting the INMI vector at \$0318/\$0319 to point to user-written code. On completion of the NMI routine, control should return to the end of the ROM-based NMI handler, which restores original register contents, and so on. Otherwise, the user must write code to do so himself.

If the user wishes to solely redirect processing of the warm-start RUN-STOP/RESTORE keystroke, the SYSTEM-VECTOR at \$0A00/\$0A01 can be pointed to user-written code. Figure 2-6 shows an example of this technique. Once this routine has been entered and activated, the RUN-STOP/RESTORE key sequence will not only halt

BASIC 7.0 program execution, but will close all open files on the disk drive. This avoids the creation of “splat” file entries in a disk’s directory, and also can help the user avoid losing valuable data files if it is necessary to interrupt a program’s execution.

The IRQ Interrupt

The 8502 processor automatically routes all ordinary IRQ interrupt requests through the IRQ vector at \$FFFE/\$FFFF. Whenever an 8502 BRK (break) instruction is encountered, control is also passed through the IRQ vector.

The beginning of the C-128 IRQ handler is quite similar to the NMI handler; the 8502 accumulator and index register contents are stored on the stack, along with the current memory configuration. The majority of the C-128 IRQ interrupt code is listed in Figure 2-7. By examining the BRK flag in the status byte preserved on the stack, the front end of the IRQ handler determines whether an 8502 BRK instruction was encountered or an actual IRQ event occurred. Execution thus passes through either the RAM-based IRQ or BRK vectors (located at \$0314/\$0315 and \$0316/\$0317, respectively). Like the NMI handler, this “front-end” code is contained in both RAM banks and in high ROM due to the fact that an IRQ can occur while the C-128 system is in any memory configuration.

After system initialization, the BRK vector points to the machine language monitor’s BRK instruction entry point at \$B003. Interspersing BRK instructions throughout machine-language programs is thus a good practice—whenever a BRK instruction is encountered during program execution, control passes to the monitor, displaying the contents of the various 8502 registers. And, by diverting the C-128’s BRK vector at \$0317/\$0318, a programmer can create a phony, one-byte “JSR \$xxxx” instruction. For example, if an application program redirected the BRK vector to point to the Kernal BSOUT (also called CHROUT) routine at \$FFD2, JSR calls to \$FFD2 could be replaced by BRK instructions. Although BRK calls are time-consuming, they are quite useful in patching and debugging previously written programs because only one-byte BRK

```

$FF17  PHA          ; put contents of .A, .X, and .Y
        TXA          ; registers on stack.
        PHA
        TYA
        PHA
        LDA $FF00    ; get MMU configuration data and
        PHA          ; preserve it on stack, too.
        LDA #$00     ; select "system" bank (Bank #15- RAM0, ROM, I/O)
        STA $FF00    ; and store to MMU's CR.
        TSX          ; get stack pointer, then fetch preserved 8502
        LDA $0105,X  ; from stack and test
        AND #$10     ; if BRK instruction or IRQ occurred.
        BEQ $FF30    ; branch to IRQ vector if normal IRQ event occurred.
        JMP ($0316)  ; else jump thru BRK vector.
                    ; (normally points to $B000, the Monitor entry.)
$FF30  JMP ($0314)  ; IRQ vector normally points to IRQ handler @ $FA65.
        ....
$FA65  CLD          ; ensure 8502 decimal mode flag is cleared.
        JSR $C024    ; call Screen Editor's IRQ handler. (This routine
                    ; blinks the cursor, scans the keyboard, and
                    ; manipulates graphic/text split screens.)
        BCC $FA7D    ; .C was clear if only split screen needed attention.
                    ; If .C set, we have

```

Figure 2-7. Overall structure of the C-128 IRQ-handler code

instructions have to be inserted into the original program. The “phony JSR” can later be disabled by pointing the BRK vector to an RTI (not RTS) instruction.

Assuming that an IRQ has actually occurred, control will pass through the vector (called IIRQ) at \$0314/\$0315. This vector normally points to the ROM-resident IRQ-handler code at \$FA65, and can be diverted to user-written code if care is used. The first activity of the ROM routine is to ensure that the 8502 decimal-mode flag is clear. (If an applications program is being created that will use BCD—not true binary—arithmetic routines, then the programmer must create some new IRQ- and NMI-handler routines that do not clear the 8502’s decimal-mode flag bit. These routines could simply be mere copies of the ROM routines, minus the CLD instructions; of course, the IIRQ and INMI indirect RAM vectors would have to point to the new routines.) After this, the screen editor interrupt-driven routine, EDIRQ (at \$C024), is called. This section of code does all the house-

keeping that a split-screen graphics/text display requires. It scans the C-128's full keyboard and blinks the cursor of the 40-column screen editor. (The 8563 80-column chip can flash the cursor itself without software intervention.)

Programmers should note the more complex IRQ-based screen editor and the two-tiered interrupt scheme utilized by the C-128. All-graphic or all-text video displays present no problem; the complete IRQ handler code is normally executed every 1/60th of a second. However, when split-screen (graphics with text) video is displayed, IRQs occur at twice the standard rate, or every 1/120th of a second. There are really two "different" IRQs occurring when a split screen is being displayed.

In this dual IRQ system, the "main" IRQ routine simply runs through all the IRQ handler code. This updates the software-based "jiffy" clock, scans for a RUN-STOP keystroke, and checks if any cassette (Datasette) player keys are pressed. It also calls the BASIC 7.0 IRQ routine that checks for sprite collisions, handles light-pen triggers and music generation, and so forth. After this, the stack is restored by the PREND routine (at \$FF33), whose RTI instruction returns control to the interrupted routine.

The "middle" IRQ routine merely executes the screen editor's raster-interrupt code contained in the EDIRQ routine. EDIRQ returns to the main body of the IRQ handler with the 8502 carry bit set if complete execution of the IRQ-handler code is necessary — that is, if it is the "main" IRQ. If, however, the system is in the "middle" IRQ phase, EDIRQ returns to the IRQ handler with the 8502 carry bit clear, indicating that no more interrupt processing is necessary. A BCC instruction passes control to PREND, which cleans up the stack at the end of an IRQ or NMI event. Note that the "main" IRQ events are still separated by 1/60th of a second; the "jiffy" clock will still maintain correct timekeeping, and the keyboard will still be scanned at the same rate. Split-screen mode will take away some processing time from an application, however.

Timing is extremely critical during split-screen operations. The programmer's routines should not directly access the VIC-II chip

while split-screen mode is in use. Normally, the screen editor loads video chip registers (both VIC-II and 8563) from the low-memory locations listed in Table 2-8 during execution of the EDIRQ routine. Thus, “well-behaved” programs needing VIC-II access should write

Table 2-8. Important “Shadow” Video Registers Used by Screen Editor

| <i>Hex Address</i> | <i>Label</i> | <i>Comments</i> |
|--------------------|--------------|--|
| \$00D8 | GRAPHM | Screen editor does not interfere with VIC-II chip’s operation if GRAPHM set to \$FF. Bit #7 indicates Multicolor mode (MCM), Bit #6 indicates split-screen text and graphics, and Bit #5 is a Bit-Map mode (BMM) flag. |
| \$00D9 | CHAREN | Selects RAM or ROM character definition fetches, controlled by Bit #2. |
| \$0A2B | CURMOD | 8563 cursor mode (solid, none, blinking, etc.). |
| \$0A2C | VM1 | VIC-II text mode video matrix and character definition start address. (Similar to \$D018, text mode.) |
| \$0A2D | VM2 | VIC-II graphic mode color matrix and bitmap start address. (Similar to \$D018, bitmap mode.) |
| \$0A2E | VM3 | 8563 80-column text matrix start address (within 16K of independent video RAM). |
| \$0A2F | VM4 | 8563 80-column character attribute table start address (within 16K of independent video RAM). |
| \$0A34 | SPLIT | Raster scan-line value for “Middle” IRQ during a split-screen display. |

to these “shadow” registers instead of the VIC-II registers themselves. However, programs can manipulate split screens without screen editor intervention if GRAPHM (at location \$D8) contains \$FF. In fact, the programmer can directly access the VIC-II chip without having to use the shadow video registers if GRAPHM contains \$FF.

The “main” IRQ code, as mentioned before, also calls routines in the BASIC 7.0 ROM that handle IRQ-driven events such as detecting sprite collisions and reading the light pen. However, this code will be bypassed if bit #0 of location \$0A04 (INIT_STATUS) is zeroed. This occurs in a partially initialized system (one where the Kernal is operational but BASIC 7.0 hasn't yet been initialized). The IRQ_WRAP_FLAG also can disable the BASIC IRQ code (callable at \$4006); it is used to avoid “wraparound” IRQs. Setting IRQ_WRAP_FLAG (located at \$12FD) to \$01 effectively disables the BASIC 7.0 IRQ; it has the same effect as clearing bit #0 of location \$0A04 (INIT_STATUS).

Chapter 3

C-128 Memory Usage

The hardware and software architecture of the war-horse C-64 is clearly reflected in the C-128. In fact, segments of the current C-128 memory map often appear startlingly similar to the C-64's. However, the extended Kernal, a smarter DOS (just a DOS interface, really—DOS itself resides in the 1541/1571 disk drive controller), and the comprehensive BASIC 7.0 all take their toll on C-128 RAM usage.

The contents of zero-page RAM are much the same as they were on the C-64. At the very beginning of zero-page RAM, though, several bytes are reserved by the Kernal for complex bank-switching operations. Thus, standard C-64 zero-page locations are usually “shifted up” a few bytes in the C-128 memory map; other C-64 zero-page locations have been transferred to page 2, 3, or 4 because of lack of room.

The C-128's lowest 1K of RAM is considered *common RAM* because it is always available to the 8502 processor's address space, no matter what RAM bank or ROMs are currently "in context." Although common RAM can normally be considered part of any C-128 logical bank, in reality this shared RAM is always part of RAM bank 0. Because this region of memory is always available to the 8502, several small RAM-resident routines allow *cross-bank* operations—it is possible to JSR or JMP to a routine in another memory configuration, just as it is possible to store, fetch, and compare bytes across bank boundaries. Some of these routines are downloaded from ROM upon system initialization; others are downloaded when BASIC 7.0 completes its cold-start initialization.

The C-128 has an extensive arsenal of user-modifiable indirect vectors in page 3. This table is much more extensive than that of the C-64 and includes several extra BASIC 7.0 and screen editor indirect vectors. BASIC graphics and sound operations reserve much of the RAM from \$0A00 to \$12FF in bank 0. User-written machine-code programs can reside in the function-key software buffer at \$1300-\$1BFF, since this area is currently reserved by Commodore. BASIC 7.0 program storage begins at \$1C00 if normal text is being displayed. About 60K of program storage is available for a BASIC 7.0 program, unless BASIC creates a high-resolution bit-mapped display. In this case, the start of the BASIC program is moved to \$4000 in RAM bank 0.

So far, little mention has been made of RAM bank 1. It has over 60K of RAM that normally stores BASIC 7.0 numeric variables, arrays (string and numeric), and strings. However, variable storage begins at \$0400 in RAM bank 1.

The routines in the C-128 Kernal, BASIC, and editor ROMs are extremely useful as a software "library," since they contain a host of arithmetic functions, string handlers, pre-defined graphics and I/O functions, and so on. Using these routines can save a C-128 programmer many hours of effort.

Programmers should note, however, that Commodore may revise the C-128 ROM contents periodically to fix any bugs, to increase performance, or to add extra features to future C-128s. Whenever possible, a program should call ROM routines using the \$FFXX Kernal jump table, the \$AFxx BASIC 7.0 jump table, or the \$C0xx screen editor jump table. By avoiding direct calls to ROM routines and instead using the jump tables, your programs have a much greater chance of functioning on future C-128 systems. However, this advice should not deter programmers wishing to customize the BASIC interpreter or Kernal. Well-crafted programs should be easy to “patch” in the event revised C-128 systems appear on the market.

Zero-Page and Page 1 Memory Usage

As in all computers based on a 6500-compatible processor, the first 256 bytes of RAM (zero-page memory) are treated specially by the processor. These locations can usually be accessed more quickly than can other memory locations, due to the 6502's zero-page addressing mode. In addition, an operand for a 6502 opcode is only one byte long when accessing zero-page memory. Commodore 128 programmers will notice the extra zero-page overhead required by the BASIC 7.0 interpreter. The Kernal cross-bank routines (INDFET, INDSTA, JSRFAR, and so on) also reserve a small portion of zero-page RAM for their own use.

RAM in the page 1 region (\$0100-\$01FF) is primarily used as a stack, accessible by the 6502 stack push/pop instructions (such as PHA, PLA, and PHP). The lowest 55 bytes of this region, however, are used by the cassette tape I/O routines, the DOS interface (for filenames, and so on), and by BASIC 7.0's PRINT USING statement.

The following map assigns mnemonics to many zero-page and stack memory locations. Wherever possible, standard C-64 and PET/CBM mnemonics have been used. New locations pertinent only to the C-128 have been assigned mnemonics that either are easily understood by programmers or have appeared in various Commodore publications. Several mnemonics may occasionally be assigned to one memory location, implying that it is used for multiple (and often unrelated) purposes. The Commodore community owes much to “guru” Jim Butterfield, who has given many previously unnamed memory locations logical, sensible (and sometimes humorous) mnemonics. His early maps of the C-128 system helped blaze the path for later explorers.

Map of C-128 Zero-Page And Stack Usage

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|--|
| D6510 | 0000 | 8502 Data Direction Register Bit 7: (N/A). Bit 6: In. Bit 5: Out. Bit 4: In. Bit 3: Out. Bit 2: Out. Bit 1: Out. Bit 0: Out. |
| R6510 | 0001 | 8502 built-in I/O port Bit 7: Currently unused. Bit 6: CAPS LOCK key status reflected here. Bit 5: Cassette motor control; 1 = on. Bit 4: Cassette switch; low if NO button is pressed. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|--|
| | | Bit 3: Cassette data output line. Bit 2: CHAREN—ROM banking bits used in C-64 mode. Bit 1: HIRAM—ROM banking bits used in C-64 mode. Bit 0: LORAM—ROM banking bits used in C-64 mode. |
| BANK | 0002 | Current bank # (<i>not</i> MMU CR data) used by Kernal. |
| PC-HI | 0003 | Program counter MSB. |
| PC-LO | 0004 | Program counter LSB. |
| S-REG | 0005 | 8502 Processor status. |
| A-REG | 0006 | 8502 Accumulator storage. |
| X-REG | 0007 | 8502. X register storage. |
| Y-REG | 0008 | 8502. Y register storage. |
| STKPTR | 0009 | 8502 Stack pointer storage. |

NOTE: Locations \$02-\$09 are used by the Kernal during cross-bank operations. Data to be passed to another bank or to a routine hidden in another bank is placed here before calling the appropriate Kernal routine (for example, JSRFAR, INDFET, and INDCMP).

BASIC 7.0 Zero-Page Storage

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|--|
| INTEGR | 0009 | |
| CHARAC | 0009 | Temporary storage for search character in string operations. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| ENDCHR | 000A | Flag—quote at end of string? |
| TRMPOS | 000B | Screen column from last tab. |
| VERCK | 000C | Flag: 0=Load, 1=VERIFY (used only by BASIC; Kernal uses the “other” VERCK at \$0093). |
| COUNT | 000D | Input buffer pointer; # of subscripts in DIM and array references. |
| DIMFLG | 000E | Flag: default array dimension. |
| VALTYP | 000F | Variable type: \$FF=string, \$00=numeric. |
| INTFLG | 0010 | Variable type: \$80=integer, \$00=floating-point. |
| GARBFL | 0011 | Flag: DATA scan/LIST quote/ garbage collection. |
| SUBFLG | 0012 | Flag: subscript reference/user function call. |
| INPFLG | 0013 | Flag: \$00=INPUT, \$40=GET, \$98=READ |
| TANSGN | 0014 | Flag: TAN sign/comparison result. |
| CHANNL | 0015 | Current I/O device number. |
| POKER | 0016-17 | |
| LINNUM | 0016-17 | Temporary integer value (line #, GOTO, POKE, and so on). |
| TEMPPT | 0018 | Pointer: next string stack entry. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|--|
| LASTPT | 0019-1A | Pointer: current string stack entry. |
| TEMPST | 001B-23 | 9-byte stack for three string pointers. |
| INDEX | 0024-25 | |
| INDEX1 | 0024-25 | General-purpose pointer. |
| INDEX2 | 0026-27 | General-purpose pointer. |
| RESHO | 0028-2C | Workspace used by multiply and divide. |
| TXTTAB | 002D-2E | Pointer to start of BASIC (bank 0). |
| VARTAB | 002F-30 | Pointer to start of variables (bank 1). |
| ARYTAB | 0031-32 | Pointer to start of arrays. |
| STREND | 0033-34 | Pointer to (end of arrays + 1). |
| FRETO | 0035-36 | Pointer to string storage (decreases). |
| FRESPC | 0037-38 | Utility string pointer. |
| MAXMM1 | 0039-3A | Pointer to upper limit of numeric, string, and array variable storage in bank 1. |
| CURLIN | 003B-3C | Current BASIC line number. |
| TXTPTR | 003D-3E | BASIC work pointer (used by CHRGET). |
| FORM | 003F | Used by PRINT USING. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| FNDPTR | 003F-4A | Pointer to data found in search. |
| DATLIN | 0041-42 | Line number of current DATA statement. |
| DATPTR | 0043-44 | Address of current DATA statement. |
| INPPTR | 0045-46 | Vector to INPUT routine. |
| VARNAM | 0047-48 | Name of current BASIC variable. |
| | | NOTE: Remember, names of variables in Commodore BASIC 7.0 have only two significant characters; extra characters in variable names are ignored. Thus, GLEB and GLURP are regarded by all Commodore BASICs as referring to the same variable. The status of bit #7 in both characters of the variable's name reflects the variable type. See the next chapter for details of BASIC 7.0 operation. |
| VARPNT | 0049-4A | Address of current BASIC variable in bank 1 RAM. |
| FDECPT | 0049-4A | |
| FORPNT | 004B-4C | Pointer to index variable for FOR/NEXT loops. |
| LSTCPT | 004B-4C | |
| ANDMSK | 004B-4C | Mask used for AND function. |
| EORMSK | 004C | Mask used for EOR function. |
| OPPTR | 004D-4E | Pointer to BASIC's operator table (+, -, *, /, and so on). |
| VARTXT | 004D-4E | |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|--|
| OPMASK | 004F | Mask used for comparison. |
| GRBPNT | 0050-51 | Pointer: used for function definition. |
| TEMPF3 | | |
| DEFPNT | | |
| DSCPNT | 0052-54 | Pointer used for string operations. |
| HELPER | 0055 | Flag: HELP or LIST. |
| JMPER | 0056-58 | \$4C (JMP opcode) + function address. |
| TEMPF1 | 0059 | First temporary floating-point accumulator (also called FAC #3, 5 bytes long). |
| PTARG1 | 0059-5A | Used by BASIC 7.0's INSTR function. |
| PTARG2 | 005B-5C | |
| STR1 | 005D-5E | |
| STR2 | 0060-61 | |
| POSITN | 0063 | |
| MATCH | 0064 | |
| ARYPNT | 005A | Used to define arrays (DIM). |
| HIGHDS | 005A-5B | Pointer used for block transfer. |
| HIGHTR | 005C-5D | Pointer used for block transfer. |
| TEMPF2 | 005E | Temporary floating-point accumulator #2, 5 bytes long (also called FAC #4). |
| DECCNT | 005F | # decimal digits after decimal |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|--------------------|--|
| | | point in floating-point/binary conversions. |
| TENEXP | 0060 | Also used in floating-point math routines. |
| T0 | 0060 | |
| DPTFLG | 0061 | Flag: used to test for presence of decimal point in strings. |
| GRBTOP | 0061 | |
| LOWTR | 0061 | |
| EXPSGN | 0062 | Sign of FAC #1's exponent. |
| FACEX | 0063 | Floating-point accumulator #1: exponent. |
| FACHO | 0064-67 | Floating-point accumulator #1: mantissa. |
| FACSGN | 0068 | Floating-point accumulator #1: sign. |

NOTE: The primary floating-point accumulator is often called either FAC or FAC #1. If a second floating-point number is required (as in a dyadic operation), it is stored in the second floating-point accumulator; this region of zero-page memory is often referred to as ARG or FAC #2. There are two other temporary FACs, TEMPF1 and TEMPF2, used internally by BASIC math routines for intermediate storage during long calculations.

| | | |
|--------|---------|---|
| ARGEXP | 006A | Floating-point accumulator #2, similar structure to FAC #1. |
| ARGHO | 006B-6E | |
| ARGSGN | 006F | |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|--|
| ARISGN | 0070 | Sign comparison, FAC versus ARG. |
| STRNG1 | 0070 | |
| FACOV | 0071 | FAC #1 rounding byte. |
| FBUFPT | 0072-73 | Pointer to cassette buffer. |
| AUTINC | 0074-75 | AUTO command's increment value (\$00=off). |
| MVDFLG | 0076 | Flag: 9K high-resolution bit map (2K color matrix + 8K bit map) allocated. |
| KEYNUM | 0077 | Key chosen. |
| NOZE | 0077 | Leading zero counter. |
| SPRNUM | 0077 | MOVSPR temporary usage. |
| Z-P-TEMP-1 | 0077 | Used by MID\$. |
| HULP | 0078 | Counter. |
| KEYSIZ | 0078 | Function-key string length. |
| SYNTMP | 0079 | Used temporarily for indirect loads. |
| DSDESC | 007A-7C | Three-byte descriptor for disk error string, DS\$. |
| TOS | 007D-7E | Pointer to top of BASIC 7.0 512-byte run-time stack area at \$0800-\$09FF. |
| RUNMOD | 007F | Flag: RUN mode or Direct mode. |
| PARSTS | 0080 | Status word for DOS parser. |
| POINT | 0080 | Pointer to decimal point. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|-----------------|
| PARSTX | 0081 | |
| OLDSTK | 0082 | |

Zero-Page Storage for BASIC 7.0 Graphics Operations

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|---------------|------------------------|---|
| COLSEL | 0083 | Current color. |
| MULTI-COLOR-1 | 0084 | |
| MULTI-COLOR-2 | 0085 | |
| FORE-GROUND | 0086 | |
| SCALEX | 0087-88 | Horizontal scaling factor. |
| SCALEY | 0089-8A | Vertical scaling factor. |
| STOPNB | 008B | Terminate PAINT operation if wrong color. |
| GRAPNT | 008C-8D | Utility pointers for graphics operations. |
| VTEMP1 | 008E | |
| VTEMP2 | 008F | |

Zero-Page Storage for Kernal Routines

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|-----------------------|
| STATUS | 0090 | "ST" I/O status word. |
| STKEY | 0091 | RUN-STOP key flag. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|--|
| SVXT | 0092 | Tape timing constant. |
| VERCK | 0093 | Flag: \$00=LOAD, \$01=VERIFY (this location is used only by the Kernal; BASIC 7.0 uses the "other" VERCK at \$000C). |
| C3PO | 0094 | Buffered serial character awaiting output (flag). |
| BSOUR | 0095 | Actual serial character waiting for output. |
| SYNO | 0096 | Flag indicating cassette data- block-end. |
| XSAV | 0097 | Temporary store during input (tape and RS-232). |
| LDTND | 0098 | Number of currently open files; index to entry in active file table. |
| DFLTN | 0099 | Default input device (keyboard, 0). |
| DFLTO | 009A | Default output (CMD) device (screen, 3). |
| PRTY | 009B | Cassette write-operation parity. |
| DPSW | 009C | Cassette dipole switch. |
| MSGFLG | 009D | Kernal error-message flag: \$00=nil, \$40=errors, \$80=all. |
| PTR1 | 009E | Cassette error pass 1. |
| T1 | 009E | Temporary storage #1. |
| PTR2 | 009F | Cassette error pass 2. |
| T2 | 009F | Temporary storage #2. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|--|
| TIME | 00A0-A2 | “Jiffy” (1/60th second increments) clock: 3-byte, 24-hour clock. |
| PCNTR | 00A3 | Cassette temporary usage. |
| R2D2 | 00A3 | Serial bus temporary usage. |
| BSOUR1 | 00A4 | Serial routine temporary storage. |
| COUNT | 00A5 | Serial routine temporary storage. |
| CNTDN | 00A5 | Cassette sync-mark countdown timer. |
| BUFPNT | 00A6 | Cassette buffer pointer. |
| INBIT | 00A7 | RS-232 receiver, input bit storage. |
| SHCNL | 00A7 | Cassette short count. |
| BITCI | 00A8 | RS-232 receiver, bit count in. |
| RER | 00A8 | Cassette read error. |
| RINONE | 00A9 | RS-232 receiver, flag for start bit check. |
| REZ | 00A9 | Cassette reading zeroes. |
| RIDATA | 00AA | RS-232 receiver, byte buffer. |
| RDFLG | 00AA | Cassette read mode. |
| RIPRTY | 00AB | RS-232 receiver, parity store. |
| SHCNH | 00AB | Cassette short count. |
| SAL | 00AC-AD | Pointer to tape buffer. |
| EAL | 00AE-AF | Cassette end address, end of program. |
| C3PO | 00B0 | Tape timing constant. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| TEMP | 00B1 | Temporary storage. |
| TAPE1 | 00B2 | Address of tape buffer. |
| BITTS | 00B4 | RS-232 transmit bit count. |
| NXTBIT | 00B5 | RS-232 transmit next bit to send. |
| RODATA | 00B6 | RS-232 transmit byte buffer. |
| FNLEN | 00B7 | Length of filename. |
| LA | 00B8 | Current logical file number. |
| SA | 00B9 | Current secondary address number. |
| FA | 00BA | Current device number. |
| FNADR | 00BB-BC | Pointer to filename in bank 1. |
| ROPRTY | 00BD | RS-232 transmit parity. |
| OCHAR | 00BD | Output character. |
| FSBLK | 00BE | Number of blocks left to read/ write to cassette. |
| MYCH | 00BF | Serial word buffer. |
| DRIVE | 00BF | |
| CAS1 | 00C0 | Cassette motor control flag, updated during IRQ. |
| STAL | 00C1-C2 | I/O start address. |
| MEMUSS | 00C3-C4 | General-use pointer, used by Ker- nal LOAD, SAVE, and so on. |
| TMP2 | 00C5 | Tape read/write data. |
| BA | 00C6 | Destination bank for current LOAD/SAVE/VERIFY. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---------------------------------------|
| FN BANK | 00C7 | Bank containing filename (for FNADR). |
| RIBUF | 00C8-C9 | Pointer to RS-232 input buffer. |
| ROBUF | 00CA-CB | Pointer to RS-232 output buffer. |

Zero-Page Global Screen Editor Variables

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| KEYTAB | 00CC-CD | Keyboard decode table pointer (table in bank 15). |
| IMPARM | 00CE-CF | PRIMM utility string pointer. |
| NDX | 00D0 | Number of characters in keyboard buffer. |
| KYNDX | 00D1 | Pending number of characters in function-key string. |
| KEYIDX | 00D2 | Index into pending function-key string. |
| SHFLAG | 00D3 | Shift key flags (\$00 = no shifting). Bit 7: Unused. Bit 6: Unused. Bit 5: Unused. Bit 4: ALT key. Bit 3: SHIFT LOCK key (do not confuse with CAPS LOCK!). Bit 2: CTRL key. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|--------------------|---|
| | | Bit 1: COMMODORE key. Bit 0: SHIFT key. |
| SFDX | 00D4 | Current key index from keyscan. |
| LSTX | 00D5 | Last key pressed (keyscan index, not character value). |
| CRSW | 00D6 | Carriage return (CHR\$(13)) flag, triggers input. |
| MODE | 00D7 | 40/80 column flag (bit #7 high if 80-column). |
| GRAPHM | 00D8 | Flag: text/graphics mode. (\$FF = disable IRQ update of VIC-II) Bit 7: Multicolor mode flag bit. Bit 6: Split-screen (text with graphics) mode flag bit. Bit 5: Bit-map mode flag bit. Bits 4...0: Not used. |

NOTE: Because the C-128's screen editor is interrupt-driven, GRAPHM is used to select the desired VIC-II display mode. Programs should normally avoid writing directly to the VIC-II registers; instead, data should be written to the appropriate shadow registers at \$D8, \$D9, \$0A2C-\$0A2F, \$0A2B, or \$0A34. If the programmer wishes to control the VIC-II directly—and not update the VIC-II registers every IRQ cycle—then GRAPHM should contain \$FF.

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|--------------------|---|
| CHAREN | 00D9 | Character base. Bit #2 determines if character data is fetched from RAM (bit #2 high) or ROM (bit #2 clear). CHAREN is a shadow register, just like GRAPHM. It will be checked every IRQ cycle. |
| SEDSAL1 | 00DA-DB | Source pointer for MOVLIN routine, used in moving text on screen. |
| BITMSK | 00DA | Used in calculation of tab stops and line wraparounds. |
| SAVER | 00DB | Temporary storage. |
| SEDSAL2 | 00DC-DD | Destination pointer for MOVLIN routine. |
| SEDT1 | 00DE | Pointer for SAVPOS. |
| SEDT2 | 00DF | Pointer for SAVPOS. |
| KEYSIZ | 00DA | Length of definition string for PFKEY (alias KEYSET) function-key programming routine. |
| KEYLEN | 00DB | Total length of function-key definition strings. |
| KEYNUM | 00DC | Number 1 . . 10 associated with function keys F1-F8, SHIFT/RUN-STOP keystroke = 9, HELP = 10). |
| KEYNXT | 00DD | End address of string, start address of next string. |
| KEYBNK | 00DE | MMU CR configuration data (not bank #) for function key definition string. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| KEYTMP | 00DF | Temporary storage used while inserting new key definition string. |

Zero-Page Storage for Local Screen Editor Variables

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| PNT | 00E0-E1 | Pointer to current text-screen line. |
| USER | 00E2-E3 | Pointer to current color attribute line. |
| SCBOT | 00E4 | Window lower limit. Used by PLOT (\$FFFO). |
| SCTOP | 00E5 | Window upper limit. Used by PLOT (\$FFFO). |
| SCLF | 00E6 | Window left margin. Used by PLOT (\$FFFO). |
| SCRT | 00E7 | Window right margin. Used by PLOT (\$FFFO). |
| LSXP | 00E8 | Current screen input column start. |
| LSTP | 00E9 | Current screen input line start. |
| INDX | 00EA | Current input line end. |
| TBLX | 00EB | Current cursor screen line. |
| PNTR | 00EC | Current cursor column. |
| LINES | 00ED | Maximum number of screen lines. |
| COLUMN | 00EE | Maximum number of screen columns. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|--------------------|--|
| DATAX | 00EF | Current character to print. |
| LSTCHR | 00F0 | Previous character printed (used with test for ESC sequence). |
| COLOR | 00F1 | Current color print. |
| TCOLOR | 00F2 | Saved color (used with INST/DEL operations). |
| RVS | 00F3 | Reverse (RVS) mode flag. |
| QTSW | 00F4 | Quote mode flag. |
| INSRT | 00F5 | Pending inserts (if INSRT>0, INSRT = number of inserts). |
| INSFLG | 00F6 | Auto-insert mode flag (\$00=off). |
| LOCKS | 00F7 | Disable SHIFT/COMMODORE and CTRL-S key sequences. Bit #7 enables/disables case switching. Bit #6 enables/disables CTRL-S hold. |
| SCROLL | 00F8 | Disable screen scrolling and line linker. |
| BEEPER | 00F9 | Bit #7, when high, enables the CTRL-G bell tone. |
| FREKZP | 00FA-FE | Unused zero-page space for user vectors, pointers, and so on. |
| LOFBUF | 00FF | |

NOTE: Locations \$E0 through \$F9 are swapped with the contents of the RAM area at \$0A40 whenever the 40- or 80-column screen editor mode is changed. This changeover can be accomplished by an ESC-X key sequence or by calling the SWAPPER routine at \$FF5F (in the Kernal jump table) or at \$C02A (in the screen editor jump table).

Page 1 Memory Usage By BASIC 7.0 And BASIC's DOS Interface

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| BAD | 0100 | Cassette tape read errors. |
| FBUFFR | 0100-0F | DOS filename storage—16 bytes. |
| XCNT | 0110 | DOS loop counter. |
| DOSF1L | 0111 | DOS filename 1 length. |
| DOSDS1 | 0112 | DOS disk drive 1. |
| DOSF2L | 0113 | DOS filename 2 length. |
| DOSDS2 | 0114 | DOS disk drive 2. |
| DOSF2A | 0115-16 | DOS filename 2 address. |
| DOSOFL | 0117-18 | BLOAD/BSAVE start address (SA). |
| DOSOFH | 0119-1A | BSAVE ending address (EA). |
| DOSLA | 011B | DOS logical address. |
| DOSFA | 011C | DOS physical address. |
| DOSSA | 011D | DOS secondary address. |
| DOSRCL | 011E | DOS record length (REL files). |
| DOSBNK | 011F | DOS bank #. |
| DOSDID | 0120-21 | DOS disk ID. |
| DIDCHK | 0122 | Disk ID check. |
| BNR | 0123 | Beginning number pointer, PRINT USING. |
| ENR | 0124 | End number pointer. |
| DOLR | 0125 | Dollar sign flag. |
| FLAG | 0126 | Comma flag. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| SWE | 0127 | Counter. |
| USGN | 0128 | Exponent sign. |
| UEXP | 0129 | Exponent pointer. |
| VN | 012A | Number of digits before decimal point. |
| CHSN | 012B | Justify flag. |
| VF | 012C | Number of positions before decimal point. |
| NF | 012D | Number of positions after decimal point. |
| POSP | 012E | Plus or minus flag. |
| FESP | 012F | Exponent flag. |
| ETOF | 0130 | Switch. |
| CFORM | 0131 | Character counter. |
| SNO | 0132 | Sign number. |
| BLFD | 0133 | Blank/asterisk flag. |
| BEGFD | 0134 | Beginning-of-field pointer. |
| LFOR | 0135 | Length of format. |
| ENDFD | 0136 | End-of-field pointer. |
| SYSTK | 0137-FF | System stack. |

Low Memory, Pages 2 and 3

The C-128 Kernal and the BASIC 7.0 interpreter use pages 2 and 3 to store important vectors, system variables, and even a few small cross-bank transfer routines. The system input buffer and the

keyboard buffer are located in this region. Although use of this RAM area does not have the advantages of using zero-page RAM (fewer clock cycles and one-byte operands), space is still at a premium here because pages 2 and 3 are part of common RAM, always available to the system no matter what combinations of ROM, I/O devices, or RAM bank are in use.

Page 2 Storage

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|--------------------|---|
| BUF | 0200-A1 | BASIC input buffer, also used by ML monitor. 162 bytes long, can handle entry of a BASIC 7.0 statement or expression two screen lines long. |
| FETCH | 02A2-AE | Cross-bank LDA (addr, bank#), Y subroutine. |
| FETVEC | 02AA | Contains address of zero-page pointer used by FETCH. |
| STASH | 02AF-BD | Cross-bank STA (addr, bank#), Y subroutine. |
| STAVEC | 02B9 | Contains address of zero-page pointer used by STASH. |
| CMPARE | 02BE-CC | Cross-bank CMP (addr, bank#), Y subroutine. |
| CMPVEC | 02C8 | Contains address of zero-page pointer used by CMPARE. |
| JSRFAR | 02CD-E2 | Cross-bank JSR to routine in another bank. |
| JMPFAR | 02E3-FB | Cross-bank JMP to code in another bank. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|----------------|------------------------|---|
| ESC-FN- VEC | 02FC-FD | Vector: hook for additional functions. |
| BNKVEC | 02FE-FF | Vector: for function cartridge users. |

BASIC 7.0, Kernal, and Screen Editor Page 3 Vector Storage

The parentheses in “Comments” column of the following two tables contain addresses of corresponding ROM routines in current C-128 systems.

BASIC 7.0 Indirect Vectors

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| IERROR | 0300-01 | (\$4D3F) Output error message. |
| IMAIN | 0302-03 | (\$4DC6) Main system loop. |
| ICRNCH | 0304-05 | (\$430D) Tokenize (“crunch”) ASCII text to BASIC tokens. |
| IQPLOP | 0306-07 | (\$5151) LIST routine. |
| IGONE | 0308-09 | (\$4AA2) RUN routine. |
| IEVAL | 030A-0B | (\$78DA) FRMEVL formula/ expression evaluation routine. |
| IESCLK | 030C-0D | (\$4321) Escape-token crunch vector. |
| IESCPR | 030E-0F | (\$51CD) Escape-token print vector. |
| IESCEX | 0310-11 | (\$4BA9) Escape-token execution vector. |

Kernal Indirect Vectors

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| IIRQ | 0314-15 | (\$FA65) Vector to IRQ handler routine. |
| IBRK | 0316-17 | (\$B003) Break (BRK) vector, points to monitor's BRK entry. |
| INMI | 0318-19 | (\$FA40) Points to NMI interrupt handler code. |
| IOPEN | 031A-1B | (\$EFBD) OPEN routine, from \$FFC0. |
| ICLOSE | 031C-1D | (\$F188) CLOSE routine, from \$FFC3. |
| ICKIN | 031E-1F | (\$F106) CHKIN routine, from \$FFC6. |
| ICKOUT | 0320-21 | (\$F14C) CHKOUT routine, from \$FFC9. |
| ICLRCH | 0322-23 | (\$F226) CLRCHN routine, from \$FFCC. |
| IBASIN | 0324-25 | (\$EF06) CHRIN routine, from \$FFCF. |
| IBSOUT | 0326-27 | (\$EF79) CHROUT routine, from \$FFD2. |
| ISTOP | 0328-29 | (\$F66E) STOP routine, from \$FFE1. |
| IGETIN | 032A-2B | (\$EEEE) GETIN routine, from \$FFE4. |
| ICLALL | 032C-2D | (\$F222) CLALL routine, from \$FFE7. |
| EXMON | 032E-2F | (\$B006) Vector to monitor command processor. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|-------------------------------------|
| ILOAD | 0330-31 | (\$F26C) LOAD routine, from \$FFD5. |
| ISAVE | 0332-33 | (\$F54E) SAVE routine, from \$FFD8. |

Screen Editor Indirect Vectors

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| CTLVEC | 0334-35 | (\$C7B9) Print CTRL characters. |
| SHFVEC | 0336-37 | (\$C805) Print SHIFT characters. |
| ESCVEC | 0338-39 | (\$C9C1) Print ESC characters. |
| KEYVEC | 033A-3B | (\$C5E1) Points to keyboard matrix decoder logic. |
| KEYCHK | 033C-3D | (\$C6AD) Stores keystroke. |
| DECODE | 033E-49 | Keyboard matrix decode look-up tables, beginning at: \$FA80— Normal keyboard decode table. \$FAD9— Shifted keyboard decode table (righthand keycap graphic characters). \$FB32— COMMODORE keyboard decode table (lefthand keycap graphics characters). |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| | | \$FB8B—CTRL mode keyboard decode table. |
| | | \$FA80—Normal keyboard decode table, again! |
| | | \$FBE4—CAPS LOCK keyboard decode table. (There is a “bug” in the CAPS LOCK decode table on most C-128 systems—the Q key still produces a lower case letter even when the CAPS LOCK key is depressed.) |
| KEYD | 034A-53 | IRQ-driven keyboard buffer, 10 bytes long. |
| TABMAP | 0354-5D | Bit map of tab stops, 10 bytes. |
| BITABL | 035E-61 | Line-wrap bits. |
| | | NOTE: When screen editors are switched between 40- and 80-column mode, the SWAPPER routine (at \$C020 or \$FF5F) swaps tab-stop data and line-wrap data out to \$0A60. |
| LAT | 0362-6B | Logical file table. |
| FAT | 036C-75 | Device number table. |
| SAT | 0376-7F | Secondary address table. |
| CHRGET | 0380-9E | CHRGET subroutine, fetches tokens from BASIC 7.0 program or input buffer. See listing in Figure 3-1. |

```

CHRGET
$0380 INC $3D      ; Increment low byte of TXTPTR.
      BNE $0386   ; Not zero? Then continue.
      INC $3E     ; Else, increment TXTPTR's high byte.
CHRGET ; CHRGOT entry returns last token/byte
      ; previously read by CHRGET.
$0386 STA $FF01   ; Select PCRa: RAM Bank 0, no ROM, no I/O.
      LDY #$00    ; Zero counter.
      LDA ($3D),Y ; Fetch BASIC 7.0 token from RAM0.
      STA $FF03   ; Select PCB: RAM Bank 0, all ROMs, no I/O.
QNUM   ; QNUM entry determines whether or not
      ; a given byte represents an ASCII numeral.
$0390 CMP #$3A   ; .C set if token value > $39 (ASCII "9").
      BCS $039E   ; Yes, character not a numeral, so exit
      ; with .A = token's value.
      CMP #$20   ; Else test for space character.
      BEQ $0380   ; Yes, then execute CHRGET again to get next token.
      SEC        ; Set carry prior to subtraction.
      SBC #$30   ; Convert ASCII number to true binary number.
      SEC        ; Set carry prior to subtraction.
      SBC #$D0   ; If .A < ASCII "0" ($30) exit with carry set.
$039E RTS        ; .C clear only if CHRGET returns an ASCII number.

```

Figure 3-1. The C-128 CHRGET subroutine

Page 3 RAM-Resident Indirect-Load Subroutines

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|-----------------|------------------------|--|
| INDSUB- RAM0 | 0039F | Shared RAM fetch subroutine (\$03A6 = address of pointer). See Figure 3-2. |
| INDSUB- RAM1 | 03AB | Shared RAM fetch subroutine (\$03B2 = address of pointer). See Figure 3-3. |
| INDIN1- RAM1 | 03B7 | Indirect fetch via INDEX1 at \$24/\$25. (See Figure 3-4.) |
| INDIN2 | 03C0 | Indirect fetch via INDEX2 at \$26/\$27. (See Figure 3-5.) |
| INDTXT | 03C9 | Indirect fetch via TXTPTR at \$3D/\$3E. (See Figure 3-6.) |

NOTE: The indirect-load subroutines in common RAM are used primarily by the BASIC 7.0 interpreter to read data from either RAM bank. They are *not* replacements for the CHRGET routine, which additionally distinguishes between ASCII numerals and BASIC 7.0 tokens. INDIN1-RAM1, for example, is used by BASIC 7.0 to fetch string data from RAM bank 1,

```

INDSUB-RAM0 $039F 8D A6 03 STA $03A6 ; Self-modifying code! .A contains
                  ; address of pointer.
                  8D 01 FF STA $FF01 ; Select RAM0, no ROMs or I/O.
$03A5 B1 xx LDA ($xx),Y ; Fetch byte in RAM0 pointed to by
                  ; $xx, with index value in .Y.
                  8D 03 FF STA $FF03 ; Engage RAM0, system ROMs, no I/O.
                  60 RTS ; Return to caller.

```

Figure 3-2. Shared RAM fetch subroutine (\$03A6=address of pointer)

```

INDSUB-RAM1 $03AB 8D B2 03 STA $03B2 ; Self-modifying code! .A contains
                  ; address of pointer.
$03AE 8D 02 FF STA $FF02 ; Select RAM1, no ROMs or I/O.
$03B1 B1 xx LDA ($xx),Y ; Fetch byte in RAM0 pointed to by
                  ; $xx, with index value in .Y.
$03B3 8D 04 FF STA $FF04 ; Engage RAM1, system ROMs, no I/O.
$03B6 60 RTS ; Return to caller.

```

Figure 3-3. Shared RAM fetch subroutine (\$03B2=address of pointer)

```

INDIN1-RAM1 $03B7 8D 02 FF STA $FF02 ; Select RAM Bank 1.
$03BA B1 24 LDA ($24),Y ; Fetch byte in RAM0 pointed to by
                  ; $24/$25, with index value in .Y.
$03BC 8D 04 FF STA $FF04 ; Engage RAM1, system ROMs, no I/O.
$03BF 60 RTS ; Return to caller.

```

Figure 3-4. Indirect fetch via INDEX1 at \$24/\$25

```

INDIN2  $03C0  8D 01 FF STA $FF01    ; Select RAM Bank 0.
        $03C3  B1 26   LDA ($26),Y  ; Fetch byte in RAM0 pointed to by
        $03C5  8D 03 FF STA $FF03    ; $26/$27, with index value in .Y.
        $03C8  60      RTS           ; Engage RAM1, system ROMs, no I/O.
                                           ; Return to caller.

```

Figure 3-5. Indirect fetch via INDEX2 at \$26/\$27

while INDTXT and INDIN2 are able to fetch data from RAM bank 0 (for reading strings contained within program text, and so on). INDIN1 is particularly useful because it allows the user to have ready access to BASIC-created strings—the pointer at \$24/\$25 is properly set up by many of BASIC's string routines, and will contain the string's address in bank 1.

These indirect-load routines are merely variants of the Kernal's general-purpose INDFET routine. BASIC uses them instead of the Kernal's because they are already set up for given memory configurations. In other words, no logical bank number-to-MMU data look-up is necessary, saving clock cycles. No particular memory configuration is needed prior to the call to these indirect-load routines, either, since they are located in shared RAM.

If the programmer is careful to return these routines to their original state, they can be easily modified for special purposes. (Interrupts should be

```

INDTXT  $03C9  8D 01 FF STA $FF01    ; Select RAM Bank 0.
        $03CC  B1 3D   LDA ($3D),Y  ; Fetch byte in RAM0 pointed to by
        $03CE  8D 03 FF STA $FF03    ; TXTPTR, with index value in .Y.
        $03D1  60      RTS           ; Engage RAM0, system ROMs, no I/O.
                                           ; Return to caller.

```

Figure 3-6. Indirect fetch via TXTPR at \$3D/\$3E

disabled, because any interrupt-driven BASIC graphics or music I/O will probably result in a system crash.) For example, the \$B1 opcode for LDA (\$xx),Y could be replaced with a \$91 opcode in the INDIN2 routine (see Figure 3-5). This creates an STA (\$xx),Y instruction, allowing the programmer to have a customized indirect-store routine.

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|--------------------|--|
| ZERO | 03D2 | BASIC floating-point constant. |
| CUBNK | 03D5 | SYS, POKE, and PEEK functions store the current memory configuration as set by the BANK command. |
| TMPDES | 03D6 | INSTR uses this for temporary storage. |
| FNBNK | 03DA | BANK pointer for string/number conversion. |
| SAVSIZ | 03DB | SSHAPE temporary work area. |
| BITS | 03DF | Overflow bits for floating-point accumulator #1. |
| SPRTMP-1 | 03E0 | SPRSAB temporary storage. |
| SPRTMP-2 | 03E1 | |
| FG-BG | 03E2 | Packed foreground/background color nybbles. |
| FG-MC1 | 03E3 | Packed foreground/multicolor nybbles. |

Bank 0 RAM Usage

Much of the low RAM in bank 0 is occupied by the 40-column VIC-II text screen. Although the VIC-II screen is normally considered to be a 1K region, it really only occupies the first 1000 bytes out of the 1024 bytes allocated to it. The remaining 24 bytes can be used to store pointers to sprite definitions.

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|--|
| VICSCR | 0400-07E7 | 40-column VIC-II text screen memory. |
| SPRPTS | 07E8-07FF | Storage area for pointers to sprite definitions. |
| BASSTK | 0800-09FF | “Run-time” software-created stack for BASIC 7.0. |

Absolute Kernal Variables

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| SYSVEC | 0A00-01 | System vector, taken upon RUN-STOP/RESTORE key sequence. Normally points to \$4003, BASIC 7.0's warm-start entry. |
| DEJAVU | 0A02 | Kernal warm- and cold-start status vector. Indicates whether or not Kernal has already been initialized. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|-----------------|------------------------|---|
| PALNTS | 0A03 | PAL/NTSC system flag. |
| INIT_ STATUS | 0A04 | <p>Indicates the parts of the operating system that have already been initialized. Set bits indicate that a particular phase of the initialization has already occurred. The INIT_ STATUS bit pattern is as follows:</p> <p>Bit #7: 8563 character set downloaded from ROM to RAM. Custom character sets can be preserved even after a reset by ensuring that bit #7 of INIT_ STATUS is set.</p> <p>Bit #6: CINT has reset both the keyboard matrix decode tables and programmable key definitions.</p> <p>Bit #1: BASIC 7.0 has already been initialized, and the Kernal should call the BASIC IRQ handlers.</p> |
| MEMSTR | 0A05-06 | Memory-bottom pointer set by Kernal's MEMBOT. |
| MEMSIZ | 0A07-08 | Memory-top pointer set by Kernal's MEMTOP. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| IRQTMP | 0A09-0A | Save IRQ during tape. |
| CASTON | 0A0B | TOD sense during tape. |
| KIKA26 | 0A0C | Tape read, temporary storage. |
| STUPID | 0A0D | Tape read IRQ indicator. |
| TIMOUT | 0A0E | Fast serial time-out flag. |
| ENABL | 0A0F | RS-232 I/O enabled. |
| M51CTR | 0A10 | RS-232: control register. |
| M51CDR | 0A11 | RS-232: command register. |
| M51AJB | 0A12-13 | RS-232: user-determined baud rate. |
| RSSTAT | 0A14 | RS-232: status register. |
| BITNUM | 0A15 | RS-232: number of bits left to send. |
| BAUDOF | 0A16-17 | RS-232: baud rate full bit time. |
| RIDBE | 0A18 | RS-232: receive pointer. |
| RIDBS | 0A19 | RS-232: input pointer. |
| RODBS | 0A1A | RS-232: transmit pointer. |
| RODBE | 0A1B | RS-232: send pointer. |
| SERIAL | 0A1C | Fast serial I/O internal/external flag. |
| TIMER | 0A1D-1F | Decrementing "jiffy" clock register. |
| XMAX | 0A20 | Size of keyboard buffer, normally=10. |
| PAUSE | 0A21 | CTRL-S hold-off flag. Contains \$13 if CTRL-S is in effect. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| RPTFLG | 0A22 | Key repeat flag (\$80=all keys repeat, \$40=none repeat). |
| KOUNT | 0A23 | Delay between key repeats (number of jiffies). |
| DELAY | 0A24 | Hold time before key repeats (number of jiffies.) |
| LSTSHF | 0A25 | SHIFT/COMMODORE: delay between case-switches. |
| BLNON | 0A26 | Blinking-cursor flag (Blink enabled if BLNON=\$FF). |
| BLNSW | 0A27 | Cursor blink enable: \$00=flash. |
| BLNCT | 0A28 | Cursor blink counter. |
| GDBLN | 0A29 | Character under cursor. |
| GDCOL | 0A2A | Cursor color before blink. |

Shadow Register Area For VIC-II And 8563 Video Chips

Due to the nature of the C-128's IRQ-driven screen editors, direct writes to VIC-II or 8563 registers often will have no effect. This is because some VIC-II and 8563 registers are updated during every "middle" IRQ cycle. Although writing an \$FF byte to GRAPHM (at \$D8) allows the user to control the VIC and/or the 8563 directly (by forcing the EDIRQ routine to be bypassed), it is often easier to write to the shadow registers. Their contents are loaded into the video chips every time the EDIRQ code is executed.

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|--|
| CURMOD | 0A2B | 8563 cursor mode (none, solid, flashing, and so on). |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| VM1 | 0A2C | VIC-II text screen and character definition base address pointer. |
| VM2 | 0A2D | VIC-II bit-map base address. |
| VM3 | 0A2E | 8563 text screen base address. |
| VM4 | 0A2F | VDC attributes base. |
| LINTMP | 0A30 | Temporary pointer. |
| SAV80A | 0A31 | Temporary storage for 8563 80-column routines. |
| SAV80B | 0A32 | Temporary storage for 8563 80-column routines. |
| CURCOL | 0A33 | 8563 cursor color before cursor blink occurs. |
| SPLIT | 0A34 | VIC-II raster value for split-screen changeover. |
| FNADRX | 0A35 | Save .X during banking operations. |
| PALCNT | 0A36 | Jiffy-cycle (1/60 sec.) timing adjustment for PAL (non-U.S.) C-128 systems. |
| SPEED | 0A37 | System clock speed saved here during serial and tape I/O operations. |
| SPRITES | 0A38 | Save sprite enables during serial and tape I/O. |
| BLANKING | 0A39 | Save video blanking status during tape operations. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|--|
| HOLD-OFF | 0A3A | Flag allowing user to gain full control of VIC-II (programmers should set GRAPHM to \$FF to gain full control of VIC-II chip instead). |
| LDTB1SA | 0A3B | High byte of start address of VIC-II text screen. |
| CLR-EA-LO | 0A3C | Low and high byte of address used in 8563 block-fill. |
| CLR-EA-HI | 0A3D | |
| SWAPAREA | 0A40-7F | Area reserved for screen variable swapouts during 40/80-column editor changeover. |

Monitor RAM Area

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|--|
| XCNT | 0A80-9F | Buffer for monitor's (C)ompare command, 32 bytes long. |
| HULP | 0AA0 | |
| FORMAT | 0AAA | |
| LENGTH | 0AAB | Used by assembler and disassembler. |
| MSAL | 0AAC | Used by assembler. |
| SXREG | 0AAF | Temporary storage for 8502 .X register. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|--|
| SYREG | 0AB0 | Temporary storage for 8502 .Y register. |
| WRAP | 0AB1 | Temporary storage. |
| XSAVE | 0AB2 | .X saved here during indirect subroutine calls. |
| DIRCTN | 0AB3 | Direction flag for monitor's (T)ransfer command. |
| COUNT | 0ABA | Parse number conversion. |
| NUMBER | 0AB5 | Parse number conversion. |
| SHIFT | 0AB6 | Parse number conversion. |
| TEMPS | 0AB7 | |

External ROM Cartridge Variables

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| CURBNK | 0AC0 | Current function-key ROM bank. |
| PAT | 0AC1-C4 | Physical address table; contains IDs of cartridges. |
| DK-FLAG | 0AC5 | Reserved for foreign systems. |
| | 0AC6-0AFF | Reserved for system. |
| TBUFFER | 0B00-BF | Cassette buffer, 192 bytes long. |
| RS232I | 0C00-0CFF | RS-232 input buffer. |
| RS232O | 0D00-0DFF | RS-232 output buffer. |
| PKYBUF | 1000-09 | Programmed function key string lengths. |
| PKYDEF | 100A-FF | Programmed key definition strings. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|--------------------|--|
| DOSSTR | 1100-1130 | DOS output buffer. |
| | 1131-114E | BASIC 7.0 graphics variables. |
| | 1139-1148 | Line-drawing variables. |
| | 1149-114F | Angle routine variables. |
| | 1150-1161 | Circle-drawing, shape, and general-use variables. |
| CHRPAG | 1168 | High byte of start address of character ROM; used by BASIC's CHAR command. |
| BITCNT | 1169 | Temporary use for GSHAPE statement. |
| SCALEM | 116A | SCALE mode flag. |
| WIDTH | 116B | Double-width (graphics) flag. |
| FILFLG | 116C | Shape area-fill flag. |
| BITMSK | 116D | Bit-mask temporary storage. |
| NUMCNT | 116E | |
| TRCFLG | 116F | Trace-mode-on flag. |
| RENUM1 | 1170-71 | Re-number temporary storage. |
| RENUM2 | 1172-73 | Re-number temporary storage. |
| ADRAY1 | 117A-7B | Vector to FP-to-integer routine; points to \$84B4. |
| ADRAY2 | 117C-7D | Vector to integer-to-FP routine; points to \$793C. |
| SPRITE-DATA | 117E-D5 | Sprite speed and direction table. |
| VICSAVE | 11D6-EA | Register contents used to update VIC-II during retrace. |
| LPEN | 11E9-EA | Light-pen X and Y values. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|-------------------|------------------------|---|
| UPPER- LOWER | 11EB | Pointer to upper/graphic character set. |
| DOSSA | 11ED | Temporary storage for file secondary addresses when using RECORD command. |
| OLDIN | 1200-01 | Previous BASIC line number. |
| OLDTXT | 1202-03 | Pointer to BASIC statement to be executed by CONT. |
| PUCHRS | 1204-07 | PRINT USING characters (*,.\$). |
| ERRNUM | 1208 | ER = error number of last BASIC 7.0 error. |
| ERRLIN | 1209-0A | EL = error line number (\$FFFF = no error). |
| TRAPNO | 120B-0C | RESUME—line to go to on error (\$FFxx none). |
| TMPTRP | 120D | Hold trap number temporarily. |
| ERRTXT | 120E | |
| TEXTOP | 1210-11 | End of BASIC (bank 0). |
| MAXMEMO | 1212-13 | BASIC program limit (\$FF00). |
| TMPTXT | 1214-15 | Used by DO loop. |
| USRPOK | 1218-1A | USR program jump—\$4C (JMP), \$lo, \$hi. |
| RNDX | 121B-1F | RND seed value. |
| | 1220-1271 | Music variables. |
| INT-TRIP- FLAG | 1276 | Interrupt storage. |
| INT-ADR- LO | 1279 | Interrupt storage. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|-------------------------|------------------------|---|
| INT-ADR- HI | 127C | Interrupt storage. |
| INTVAL | 127F | Interrupt storage. |
| COLTYP | 1280 | Interrupt storage. |
| | 1281-12B2 | Sound variables. |
| IRQ__ WRAP__ FLAG | 12FD | Disables wraparound BASIC 7.0 IRQ events. |
| | 1300-18FF | Application program area. |
| START | 1C00 | Normal start of BASIC text. |
| BASIC | 1C00-1FFF | Color matrix for high-resolution bit map. |
| | 2000-3FFF | Screen memory (in high-resolution bit-map mode). |
| | 4000-FF00 | BASIC's RAM memory (in high- resolution bit-map mode). |

NOTE: BANK 1 \$0400-\$FF00: BASIC variable, array, and string storage.

ROM Bank \$4000-\$AFFF: BASIC 7.0 Interpreter ROM code (see Chapter 4 for map).

Monitor ROM Entry Points

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| MONITR | B000 | JMP to monitor startup and initialization entry at \$B021. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| MONBRK | B003 | JMP to Break (BRK) instruction handler; vectored through indirect at \$0316. |
| JMONCMD | B006 | JMP entry to command parser routine. |
| DSPREG | B050 | Display contents of 8502 registers (“R” command). |
| MAIN | B08B | Command-line entry routine. Leading spaces are skipped, and MAIN can accept up to 160 characters. Control passes through the EXMON vector at \$032E to the actual MONCOMD routine, comparing the entered command with a list of valid monitor commands. |
| ERROR | B0BC | Error routine—prints CURSOR RIGHT character and question mark, then reenters MAIN via a JMP instruction. |
| | B0E3 | EXIT command passes control through the SYSTEM-VECTOR at \$0A00; normally, this indirect vector points to BASIC 7.0’s warm-start entry at \$4003. |
| CMDCHR | B0E6-FB | Table of valid monitor commands (A, C, D, F, G, H, J, M, R, T, X, “@”, “.”, “>”, “;”, \$, +, &, %, L, S, V). |
| CMDQTY | B0FC | Number of valid commands (22). |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| FETCH | B11A | Read a byte from any bank. |
| STASH | B12A | Write a byte to any bank. |
| CMPARE | B13D | Cross-bank memory comparison. |
| DSPMEM | B152 | Memory display command. |
| SETREG | B194 | “;” command — modify register contents. |
| SETMEM | B1AB | “>” command — modifies memory. |
| GO | B1D6 | GO command — control will not return to ML monitor unless 8502 BRK instruction (opcode \$00) is encountered. |
| GOSUB | B1DF | Jump-to-subroutine command. Upon a return routine’s RTS instruction, control is passed back to MAIN routine. Uses Kernal’s JSRFAR routine. |
| DMPONE | B1E8 | Dumps one line of memory (16 byte has display with ASCII equivalents displayed also) to current output device. |
| COMPAR | B231 | Compare command. |
| TRANSFR | B234 | Transfer command. Both COMPAR and TRANSFR use the same code at \$B236-\$B2CB. A flag is set to determine the operation; if VERCK (at \$0093) contains \$80, a Transfer is executed. If VERCK contains |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|--------------------|--|
| | | \$00, then a Compare operation occurs. |
| | | T2 (at \$66/\$67) contains start address. |
| | | T1 (at \$63/\$64) contains number of bytes to be transferred or compared. |
| | | T0 (at \$60/\$61) contains destination of transfer or start of second area of memory to be compared. |
| HUNT | B2CE | Hunt command. |
| LODSAV | B337 | Load, save, and verify. |
| FILL | B3DB | Fill command. |
| ASSEM | B406 | Assemble command. |
| DISASM | B599 | Disassemble command. |
| DIS300 | B5D4 | Print disassembled instruction from opcode. |
| DSET | B659 | Classify opcode type. |
| NMODE | B6C3-B7A4 | Compressed instruction mnemonic tables and data for disassembler. |
| PARGOT | B7A5 | Input parameter after monitor command already read. |
| EVAL | B7CE | Get value(s) and put into T0, T1, T2 (at \$60...\$67). |
| MAKHEX | B8D2 | Convert byte in .A to two ASCII characters. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| GNC | B8E9 | Get next character from input buffer. |
| T0TOT2 | B901 | Copy addresses in monitor's temporary storage area. |
| INCT2 | B950 | Increment pointer. |
| CONVERT | B9B1 | \$, +, &, % commands—convert between bases. |
| DISK | BA90 | "@" command passes command string to DOS. |
| DISKDIR | BB03 | Display disk's directory. |
| | BB73-BFFF | Unused ROM space—filled with \$FF bytes. |

Screen Editor Jump Table

Addresses in parentheses at the beginning of the "Comments" column contain the location of the actual screen editor code.

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|--|
| CINT | C000 | (\$C07B) Initialize editor/screen, also available at \$FF81. |
| DISPLY | C003 | (\$CC34) Show character in .A, color in .X. |
| LP2 | C006 | (\$C234) Get key during IRQ cycle into .A. |
| LOOP5 | C009 | (\$C29B) Get character from screen into .A. |
| PRINT | C00C | (\$C72D) Print character in .A. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|--------------------|---|
| SCRORG | C00F | (\$CC5B) Get screen in .X/.Y, also available at \$FFED. |
| SCNKEY | C012 | (\$C55D) Scan keyboard, also available at \$FF9F. |
| REPEAT | C015 | (\$C651) Repeat keystroke. |
| PLOT | C018 | (\$CC6A) Read/set cursor location in .X/.Y. Also available at \$FFF0. |
| CURSOR | C01B | (\$CD57) Read or set 8563 cursor (set if .C is high). |
| ESCAPE | C01E | (\$C9C1) Perform ESC function, with character in .A. |
| KEYSET | C021 | (\$CCA2) Redefine programmable key. |
| EDIRQ | C024 | (\$C194) Editor's IRQ entry— handles split screens, reads keyboard, and so on. |
| INIT80 | C027 | (\$CE0C) Downloads 8563 character definitions from ROM to independent 16K 8563 video RAM. |
| SWAPPER | C02A | (\$CD2E) 40/80-column editor local variable swap. |
| WINDOW | C02D | (\$CA1D) Set window corners. |
| LDTB | C033-C064 | Table of screen-line low and high bytes. |
| | C065-C07A | CTRL and ESC handler vectors and keyboard decode matrix pointers. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|--------------------|--|
| CINT | C07B | Initialize screen (CINT) routine here. |
| STUPT | C15C | Set up new line. |
| SCOLOR | C17C | 80-column color settings. |
| IRQ | C194 | Editor's IRQ entry. |
| LP2 | C234 | Get a key (called by EDIRQ). |
| LOOP5 | C29B | Input from screen. |
| QTSWC | C2FF | Check for quotes. |
| LOOP2 | C30C | Final code for screen print. |
| SCRDWN | C37C | Scroll down screen. |
| SCRUP | C3A6 | Scroll up screen. |
| IRQK | C55D | Scan keyboard. |
| KEYLOG | C5E1 | Keyboard read. |
| REPEAT | C651 | Get key and repeat. |
| KEYPUT | C6AD | Keyboard decode tables. |
| BLINK | C6E7 | VIC-II cursor blink routine. |
| PRINT | C72D | Print to screen. |
| | C77D | ESC-O handler—reset all screen modes (Quote, Insert, and so on). |
| | C7B9 | Print CTRL character (vectored through \$0334). |
| | C805 | Print SHIFT character (vectored through \$0336). |
| | C854 | CHR\$(29): cursor right. |
| | C85A | CHR\$(17): cursor down. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|--------------------|--|
| | C880 | CHR\$(14): text mode (lower case select). |
| | C8A6 | CHR\$(11): lock character set. |
| | C8AC | CHR\$(12): unlock character set. |
| | C8B3 | CHR\$(19): home cursor. |
| | C8BF | CHR\$(146): reverse video off. |
| | C8C2 | CHR\$(18): Reverse video on. |
| | C8C7 | CHR\$(2): underline on—8563 only. |
| | C8CE | CHR\$(130): underline off—8563 only. |
| | C8D5 | CHR\$(15): flash on—8563 only. |
| | C8DC | CHR\$(143): flash off—8563 only. |
| | C8E3 | Screen line opened up (insert). |
| | C91B | CHR\$(20): delete. |
| | C94F | CHR\$(9): tab character. |
| | C961 | CHR\$(24): tab switch (set or clear tab stop positions). |
| | C980 | ESC-Z: clear all tabs. |
| | C983 | ESC-Y: set default tabs. |
| | C98E | CHR\$(7): bell tone. |
| | C9B1 | CHR\$(10): linefeed. |
| ESCAPE | C9C1 | ESC key sequence handler (vectored through \$0338, IESCVEC). |
| | CA14 | ESC-T: set window top. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|--------------------|---------------------------------|
| | CA16 | ESC-B: set window bottom. |
| WINDOW | CA1B | Set window (\$C02D). |
| | CA3D | ESC-I: insert blank line. |
| | CA52 | ESC-D: delete line. |
| | CA76 | ESC-Q: erase to line end. |
| | CA8B | ESC-P: erase to line start. |
| | CA9F | ESC (at): clear rest of screen. |
| | CABC | ESC-V: scroll up. |
| | CACA | ESC-W: scroll down. |
| | CAE2 | ESC-L: scroll on. |
| | CAE5 | ESC-M: scroll off. |
| | CAEA | ESC-C: auto-insert off. |
| | CAED | ESC-A: auto-insert on. |
| | CAF2 | ESC-S: block cursor. |
| | CAFE | ESC-U: underline cursor. |
| | CB0B | ESC-E: nonflashing cursor. |
| | CB21 | ESC-F: flashing cursor. |
| | CB37 | ESC-G: bell enable. |
| | CB3A | ESC-H: bell disable. |
| | CB3F | ESC-R: screen reverse. |
| | CB48 | ESC-N: screen normal. |
| | CB52 | ESC-K: go to end of line. |
| | CBB1 | ESC-J: go to start of line. |
| DISPLY | CC34 | Send character to screen. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|--------------------|---|
| SCRORG | CC5B | Return current window size. |
| PLOT | CC6A | Read/set cursor (\$C018). |
| KEYSET | CCA2 | Define function key (\$C021). |
| SWAPPR | CD2E | ESC-X: switch 40/80 screen (\$C02A). |
| CURSOR | CD57 | Position 80-column cursor (\$C01B). |
| VDCPUT | CDCA | Set 8563 register 31 with VDC data. |
| VDCOUT | CDCC | Set screen register. |
| VDCGET | CDD8 | Read register 31 containing 8563 output data. |
| VDCIN | CDDA | Read screen register. |
| GETPNT | CDE6 | Set CRT to screen address. |
| INIT80 | CE0C | Initialize 80 columns—download ROM to RAM. |

Kernal ROM Routine Entry Points

Addresses of the form \$FFxx in parentheses in the following table indicate Kernal jump table entries. Programmers should normally use these jump addresses instead of calling the ROM routines directly. Addresses of the form \$03xx in parentheses indicate Kernal indirect vectors. These RAM-resident vectors can be directed to user-created code.

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|--|
| START | E000 | System initialization. |
| RESTOR | E056 | Reset standard I/O devices (from \$FF8A). |
| VECTOR | E05B | Set/store I/O vectors (from \$FF81). |
| RAMTAS | E093 | Clear RAM regions, including zero-page and cassette buffer, and allocate memory pointers. RAMTAS does <i>not</i> test memory. Available at \$FF87 in jump table. |
| | E0CD | Transfer IRQ code and vectors to top of RAM in all RAM banks. |
| IOINIT | E109 | Initialize I/O (\$FF84). |
| SECURE | E1F0 | Check for SYSTEM vector and “CBM” keystring. |
| POLL | E242 | Check for C-64 ROM and C-128 ROM cartridges. Logs in C-128 cartridge IDs. |
| C64MODE | E24B | GO64 code, from \$FF4D. |
| C64BEG | E263 | Code that moves GO64 code to \$02. |
| DATTBL | E2C7-E339 | Register data loaded into I/O chips (VIC, 8563, and so on). |
| TALKK | E33B | Send TALK on serial bus (\$FFB4). |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|----------------|--------------------|---|
| LSTNK | E33E | Send LISTEN on serial bus (\$FFB1). |
| SENDR | E348 | Send buffered character over serial bus. |
| ACPTR | E43E | Input a byte from serial bus. (\$FFA5). |
| TKSAK | E4D2 | Send secondary address (SA) (\$FF93). |
| CIOUTK | E503 | Output byte on serial bus (\$FFA8). |
| SCNDK | E4E0 | Send SA after LISTEN (\$FF96). |
| UNTLKK | E515 | Send UNTALK on serial bus (\$FFAB). |
| UNLSNK | E526 | Send UNLISTEN on serial bus (\$FFAE). |
| SLOWB | E5BC | Wait for D2ICR bit #3 to go high. |
| SPINP | E5C3 | Engage fast serial input mode. |
| SPOUT | E5D6 | Engage fast serial output mode. |
| SPIN_ SPOUT | E5FB | Select SPINP (.C=0) or SPOUT (.C=1) (\$FF47). |
| TR232 | E5FF | RS-232 Transmit routines. |
| RCV232 | E69D | RS-232 Receive routines. |
| BSO232 | E75C | Output a character from RS-232 port. |
| BSI232 | E7CE | Input a character from RS-232 port. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| BAUDO | E850-E863 | Baud rate table for NTSC (United States) C-128 systems. |
| BAUDOP | E864-E877 | Baud rate table for PAL (European) C-128 systems. |
| NMI232 | E878 | NMI RS-232 bit input handler. |
| CASTAPE | E8D0 | Cassette tape routines. |
| NGETIN | EEEE | Get character, usually from keyboard (\$032A) (\$FFE4). |
| NBASIN | EF06 | Get character from input device (\$FFCF) (\$0324). |
| JTGET | EF48 | Get character from tape buffer. |
| NBSOUT | EF79 | From \$FFD2, \$0326: send character to output channel. |
| NOPE | EFBD | From \$FFC0, \$031A: open file for read/write. |
| NCHKIN | F106 | From \$FFC6, \$031E: prepare a file for input. |
| NCKOUT | F14C | From \$FFC9, \$0320: prepare a file for output. |
| NCLOSE | F188 | From \$FFC3, \$031C: close a file. |
| LOOKUP | F202 | Search for file in logical file table. |
| GETLFS | F212 | Fetch file parameters from logical file table entries. |
| NCLALL | F222 | From \$FFE7, \$032C: abort I/O and close one file. |
| NCLRCH | F226 | From \$FFCC, \$0322: restore default I/O devices. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|----------------|------------------------|---|
| CLOSE__ ALL | F23D | From \$FF4A: close all files on device except for current I/O channel. |
| LOADSP | F265 | From \$FFD5: load memory from device. |
| NLOAD | F26C | Load (\$0330) or Verify setup code. |
| LDDISK | F27B | Serial load. |
| LDTAPE | F326 | Tape load. |
| BURST | F3A1 | 1571 disk "burst" load, high-speed. Uses DOS's "U0"+chr\$(31) burst-load command. |
| BURSTBYT | F4BA | Get "burst" byte from fast serial bus. |
| BURSTBLK | F4C5 | Burst-load block of data. |
| WIGGLE | F503 | Toggle clock line. |
| LUKING | F50F | Print "SEARCHING FOR" <filename> message. |
| LOADING | F533 | Print "LOADING" <filename> message. |
| SAVESP | F53E | From \$FFD8: save memory to disk or tape. |
| NSAVE | F54E | Save (through \$0332) setup routines. |
| BREAK | F5B5 | Terminate serial operations. |
| SAVING | F5BC | Print "SAVING" message. |
| SVTAPE | F5C8 | Save to tape. |
| UDTIMK | F5F8 | Update clock by 1 "jiffy" = 1/60 second (\$FFEA). |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| RDTIMK | F65E | Read TI jiffy clock (\$FFDE). |
| SETMK | F665 | Set TI clock (\$FFDB). |
| NSTOP | F66E | Scan RUN-STOP key (\$FFE1) (\$0328). |
| ERROR1 | F67C | “TOO MANY FILES” |
| ERROR2 | F67F | “FILE OPEN” |
| ERROR3 | F682 | “FILE NOT OPEN” |
| ERROR4 | F685 | “FILE NOT FOUND” |
| ERROR5 | F688 | “DEVICE NOT PRESENT” |
| ERROR6 | F68B | “NOT INPUT FILE” |
| ERROR7 | F68E | “NOT OUTPUT FILE” |
| ERROR8 | F691 | “MISSING FILE NAME” |
| ERROR9 | F694 | “ILLEGAL DEVICE NUMBER” |
| ERROR16 | F697 | “OUT OF MEMORY” |
| MSGTBL | F6B0-F71D | Rest of Kernal messages. |
| SPMSG | F71E | Print Kernal messages to screen only if output enabled by MSGFLG at \$9D. |
| SETNAM | F731 | From \$FFBD: set filename. |
| SETLFS | F738 | From \$FFBA: set file#, device#, secondary address. |
| SETBNK | F73F | From \$FF68: set load/save bank. |
| READSS | F744 | From \$FFB7: set ST byte into .A. |
| SETMSG | F75C | From \$FF90: error messages on/ off. |
| MEMTOP | F763 | From \$FF99: read/set upper limit of BASIC. |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|---------------|------------------------|--|
| MEMBOT | F772 | From \$FF9C: read/set lower limit of BASIC. |
| IOBASE | F781 | From \$FFF3: get start of I/O block in .X/.Y. |
| LKUPSA | F786 | From \$FF5C: search for a file's secondary address. |
| LKUPLA | F79D | From \$FF59: search for a file's logical address. |
| DMA__ CALL | F7A5 | From \$FF50: trigger DMA (watch for bugs—see Chapter 6). |
| FNADRY | F7AE | Get filename character from memory. |
| SALSY | F7BC | Memory operation used in LOAD and SAVE. |
| EALLY | F7C9 | Memory operation used in LOAD and SAVE. |

Kernal Indirect Cross-Bank Operations

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|--------------------------------------|
| INDFET | F7D0 | Get byte from bank (\$FF74). |
| INDSTA | F7DA | Store byte to bank (\$FF77). |
| INDCMP | F7E3 | Cross-bank byte comparison (\$FF7A). |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|--------------------|---|
| GETCFG | F7EC | Get MMU data corresponding to a bank number (\$FF6B). |

Code Transferred to Common RAM

| | | |
|----------------|------|--|
| | F800 | Routines to go to \$02A2-\$02FB. |
| | F85A | DMA code to go to \$03F0. |
| PHOENIX | F867 | Check for autostart ROM (\$FF56); falls into BOOT. |
| BOOT__ CALL | F890 | Check for autoboot disk—examine track 1, sector 0 (\$FF53) and load sector(s) if necessary. |
| BLOCK- NEXT | F9B3 | Increment boot sector. |
| BLOCK- READ | F9D5 | Execute disk block-read operation by sending “U1: 13 0 tt ss” command string to disk drive and transferring bytes to cassette buffer (\$0B00). |
| DEC-ASC | F9FB | Convert byte to two-digit ASCII decimal number. |
| PRIMMS | FA17 | Print message (\$FF7D). |
| NNMI | FA40 | NMI interrupt processing (\$0318). |
| NIRQ | FA65 | Normal IRQ processing (\$0314). |
| NMI | FF05 | Start of NMI handler—saves such things as register contents and memory configuration. |
| IRQ | FF17 | Start of IRQ handler—saves |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| | | registers, checks for BRK versus IRQ, and so on. |
| PREND | FF33 | Return from either interrupt; restore 8502 registers from stack, restore memory configuration, and so on. Both the IRQ and NMI routines “clean up” using this PREND code. |

Kernal Jump Table

Addresses in parentheses are the actual addresses of the particular Kernal routine.

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|-----------------|------------------------|---|
| SPIN__ SPOUT | FF47 | Fast disk toggle (\$E5FB). |
| CLOSE__ ALL | FF4A | Clear I/O (\$F23D). |
| C64MODE | FF4D | GO64 (\$E24B). |
| DMA__ CALL | FF50 | Trigger DMA (\$F7A5). Please note that there is a bug in the current DMA__CALL routine. See Chapter 7 for a description of the problem and how to work around it. |
| BTCALL | FF53 | Boot load program from disk (\$F890). |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|--|
| PHENIX | FF56 | Call card cold-start routines (\$F867). |
| LKUPLA | FF59 | Search tables for LA (\$F79D). |
| LKUPSA | FF5C | Search tables for SA (\$F786). |
| SWAPPER | FF5F | Swap to ALT display device (\$C02A). |
| DLCHR | FF62 | Initialize 80-column character RAM (\$C027). |
| PFKEY | FF65 | Program function key (\$C021). |
| SETBNK | FF68 | Set bank # for LOAD/SAVE/VERIFY (\$F73F). |
| GETCFG | FF6B | Convert bank # to MMU CR data byte (\$F7EC). |
| JSRFAR | FF6E | JSR to any bank, RTS to calling bank (\$02CD). |
| JMPFAR | FF71 | JMP to any bank (\$02E3). |
| INDFET | FF74 | LDA (FETVEC),Y from any bank (\$F7D0). |
| INDSTA | FF77 | STA (STAVEC),Y to any bank (\$F7DA). |
| INDCMP | FF7A | CMP (CMPVEC),Y to any bank (\$F7E3). |
| PRIMM | FF7D | Print immediate (\$FA17 — always JSR here). |
| RELEASE | FF80 | Release # of Kernal ROM (data byte only). |

Standard CBM Kernel Table

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| | FF81 | Initialize screen editor (\$C000). |
| | FF84 | Initialize input/output (\$E109). |
| | FF87 | RAM test (\$E093). |
| | FF8A | Reset vectors to default values (\$E056). |
| | FF8D | Change vectors for user (\$E05B). |
| | FF90 | Control OS messages (\$F75C). |
| | FF93 | Send SA after LISTEN (\$E4D2). |
| | FF96 | Send SA after TALK (\$E4E0). |
| | FF99 | Set/read top of memory (\$F763). |
| | FF9C | Set/read bottom of memory (\$F772). |
| | FF9F | Scan keyboard (\$C012). |
| | FFA2 | Not used in C-128. |
| | FFA5 | Handshake byte in (\$E43E). |
| | FFA8 | Handshake byte out (\$E503). |
| | FFAB | Send UNTALK (\$E515). |
| | FFAE | Send UNLISTEN (\$E526). |
| | FFB1 | Send LISTEN (\$E33E). |
| | FFB4 | Send TALK (\$E33B). |
| | FFB7 | Get I/O status ST (\$F744). |
| | FFBA | Set LA FA SA (\$F738). |
| | FFBD | Set length, FN address (\$F731). |

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| | FFC0 | Open logical file (\$031A,\$EFBD). |
| | FFC3 | Close logical file (\$031C,\$F188). |
| | FFC6 | Open channel in (\$031E,\$F106). |
| | FFC9 | Open channel out (\$0320,\$F14C). |
| | FFCC | Close I/O channels (\$0322,\$F226). |
| | FFCF | Input from channel (\$0324, \$EF06). |
| | FFD2 | Output to channel (\$0326,\$EF79). |
| | FFD5 | Load from file (\$F265). |
| | FFD8 | Save to file (\$F53E). |
| | FFDB | Set internal clock (\$F665). |
| | FFDE | Read internal clock (\$F65E). |
| | FFE1 | Scan RUN-STOP key (\$0328, \$F66E). |
| | FFE4 | Get character from queue (\$032A, \$EEEE). |
| | FFE7 | Close all files (\$032C,\$F222). |
| | FFEA | Increment clock (\$F5F8). |
| | FFED | Read screen size (\$C00F). |
| | FFF0 | Read/set X,Y on screen (\$C018). |
| | FFF3 | Return I/O base (\$F781). |
| NMI | FFFA | \$FF05=NMI. |
| RESET | FFFC | \$FF3D=reset. |
| IRQ | FFFE | \$FF17=IRQ. |

Chapter 4

The C-128 BASIC 7.0 Interpreter

The C-128's advanced BASIC 7.0 interpreter has much of the same internal structure as the older BASIC 2.0 interpreters available on the Commodore VIC-20 and C-64 computers. Additional features have been added, and there have been improvements in the internal string-handling functions — C-128 BASIC doesn't suffer from interminable "garbage collection" delays, unlike its predecessors. ("Garbage collection" refers to the BASIC interpreter's rearrangement of strings in memory and the deletion of old strings that have since been updated.)

The internal routines of previous Commodore BASIC interpreters were often a mystery even to advanced machine language programmers. Entry points to standard routines differed between the various CBM systems, even when their BASICs were quite similar.

The C-128 does not have this problem. BASIC 7.0 provides a *jump table*, similar to the Kernal's, in the \$AFxx region of memory. Programs that call any of the important routines within BASIC 7.0 using this jump table are guaranteed to function on future C-128 systems. Most of the calls within this jump table relate to floating-point arithmetic, memory transfer routines for the arithmetic functions, and data format conversions. Major BASIC 7.0 operations such as statement tokenization, line execution, and expression evaluation also have entries in the BASIC jump table. A complete map of C-128 BASIC 7.0 ROM routine entry points will be provided at the end of the chapter.

C-128 BASIC 7.0 Variable Storage

C-128 BASIC uses the same arrangement and format to store its variables as previous CBM BASICs have used. However, bank 1 RAM (\$0400-\$FEFF) is completely devoted to numeric variable, array, and string storage. There is no floating boundary between the end of the BASIC program text area and the start of variable storage, since BASIC reserves RAM bank 0 from \$1C00 (\$4000 if a BASIC 7.0-created high-resolution bit map is displayed) to \$FBFF. Thus, the BASIC system variable VARTAB, at \$2F/\$30—indicating the start of BASIC variable storage—will normally point to \$0400. BASIC 7.0 “understands” that this is RAM bank 1. The system variable ARYTAB (at \$31/\$32) points to the start of BASIC arrays. Both ARYTAB and VARTAB are stored in standard 6502 low-byte/high-byte format. (Although VARTAB normally points to \$0400, room in bank 1 RAM can be reserved for machine language programs by increasing VARTAB. Figure 4-1 shows how to reserve \$0400-\$07FF for a user-created routine by protecting it from being overwritten by BASIC variables.) The address of the end of BASIC array storage (actually, end of arrays + 1) is contained in the pointer STREND, at \$33/\$34. This pointer also indicates the bottom of the string storage area.

To understand how BASIC stores variables in C-128 RAM, type

```
POKE 47,0: POKE 48,8: REM VARTAB POINTS TO $0800.
```

Figure 4-1. Reserve \$0400-\$07FF for user-created routine residing in Bank1 RAM

in the short BASIC program contained in Figure 4-2 after resetting the system. After running this program, use the monitor to examine the contents of the storage area by typing **M 10400 10430**. The memory dump should look like that in Figure 4-3.

Each non-array variable declaration occupies seven bytes in RAM. The first two bytes of each entry contain the variable's name, and the next five bytes contain either a floating-point number, a two-byte integer, or a string descriptor consisting of a length value (one byte) and a pointer (two bytes).

In the case of true integer variables (those with a percent sign in their names), the first two bytes of the five-byte data field contain the integer in standard 6502 low-byte/high-byte format. The next three locations contain \$00 filler bytes. Therefore, no space is saved by using integer variables (there is one exception to this, to be covered later). Use of integer variables also slows down a BASIC program's operation. Since BASIC usually works with floating-point numbers, integer variables have to be converted to floating-point format, which is a time-consuming task. Anytime the value of an integer variable needs to be updated, the floating-point number must be reconverted back to integer format, which takes even more time.

```
10 A= 1.234: BC= -5.889 E04: DE= 3.578 E-04
20 HHS="ABCDE....TEST STRING....ABCDE"
30 JJS= MID$(HHS,10,11)
40 XQX= 123: YYX= -464
50 DIM QQ(4,4)
```

Figure 4-2. BASIC program to create several types of variables

```

>10400 41 00 81 1D F3 B6 46 42 43 90 E6 0A 00 00 44 45:A...s6FBC.f...DE
>10410 75 3B 97 1A 64 48 C8 1D E1 FE 00 00 4A CA 0B D4:u;..dHH.a@..JJ.T
>10420 FE 00 00 D8 D1 00 7B 00 00 00 D9 D9 FE 30 00 00:a..Xq.a...YY@0..
>10430 00 51 51 86 00 02 00 05 00 05 00 00 00 00 00 00:..qq.....

```

Figure 4-3. Monitor memory dump of beginning of C-128 variable storage area (contents generated by BASIC program in Figure 4-2)

Generally, Commodore BASIC programs do not benefit from the use of integer variables. There are several BASIC 7.0 compilers available from various software manufacturers that do make full use of integer variables without the speed penalties or filler-byte overhead. A BASIC compiler converts a whole BASIC program into machine language before the program is executed, whereas interpreters—as in the C-128—continually fetch BASIC tokens at run-time, executing the appropriate machine-code routines immediately. The “fetch” part of the interpretation, as well as the process of looking up the appropriate routines, takes a substantial amount of time. Thus compiled programs usually run much faster than interpreted ones. In fact, compiled BASIC programs run (on the average) 10 times faster than their uncompiled brethren, and sometimes have a spectacular speed-increase factor of 45 or 50. In addition, compiled programs do not have to search for a variable’s name in the variable table every time that variable is used; during the compilation process (that is, not during run-time), references to variable names are replaced with the actual addresses of the variables themselves, eliminating the time-consuming variable search process.

The name of the variable is stored in two bytes at the beginning of the entry in the variable table. This is the reason that Commodore BASIC variables have only two “significant” characters in their names—GLEP and GLORP both refer to the same variable. The high bits (bit #7) of each character in the variable’s name determine the variable’s type: floating-point, integer, or string. Ordinary floating-point variable names (A, BC, and DE in the example in

Table 4-1. Internal Storage of BASIC Variable Names

| <i>Variable or Array Type</i> | <i>Example Variable Name</i> | <i>Hex Bytes of Name</i> | | <i>Bit #7 Status</i> | |
|---------------------------------------|--------------------------------------|----------------------------------|------|----------------------------|-----------------------------|
| | | | | <i>First Character</i> | <i>Second Character</i> |
| Floating-point | AB | \$41 | \$42 | low | low |
| Integer | CD% | \$C3 | \$C4 | high | high |
| String | EF\$ | \$45 | \$C6 | low | high |

Figure 4-2) are stored with bit #7 low in both characters. Integer variable names are stored with bit #7 high in both characters, while string variable names are stored with bit #7 high only in the last character. Table 4-1 gives examples of Commodore BASIC variable naming. These conventions also apply to the naming of array variables.

A string variable entry does not contain the string itself, but rather a byte representing the length of the string and a two-byte pointer to the string's starting address (in bank 1). Actual strings are stored in high bank 1 RAM, beginning at \$FEFF and working downward. (The characters in the string itself are stored in normal low-to-high order). This string descriptor occupies only three of the available five bytes of the variable's data field, so the remaining two locations contain \$00 filler bytes. The monitor's memory dump in Figure 4-3, for example, contains an entry in the variable table for HH\$, beginning at location \$0415 in bank 1. The byte sequence \$48 \$C8 contains the variable's name, HH; because bit #7 of the variable name's second character is set, it is a string variable. The \$1D byte indicates that HH\$ is 29 bytes long, while the \$E1 \$FE sequence denotes the string's starting address at \$FEE1. Two \$00 filler bytes follow. Figure 4-4 shows a monitor memory dump of string storage RAM after the BASIC program in Figure 4-2 has been executed. Programmers should note the presence of unused "garbage" string

```

>1FEC0 2C 30 30 00 00 00 00 00 00 56 41 52 53 56 41:.,00.....VAR$VA
>1FED0 52 53 28 FF 54 45 53 54 20 53 54 52 49 4E 47 1E:RS(@TEST STRING.
>1FEED 04 41 42 43 44 45 2E 2E 2E 2E 54 45 53 54 20 53:.ABCDE....TEST S
>1FEFD 54 52 49 4E 47 2E 2E 2E 2E 41 42 43 44 45 28 FF:TRING....ABCDE(Ø
>1FF00 7F 3F 7F 01 41 78 48 8A 48 98 48 AD 00 FF 48 A9:Ø?Ø.AxH.H.H-ØH)

```

Figure 4-4. String storage in upper bank 1 RAM

data; programs that create and manipulate many strings (for example, a data base program) will have many of these discarded strings in RAM. When the bottom of string memory “collides” with the top of numeric variable and array storage, the previously mentioned garbage collection takes place, rearranging strings in memory and overwriting old, unused strings.

Array storage follows the variable table, with the start address of this region contained in the zero-page pointer ARYTAB at \$31/\$32. Each BASIC array begins with several bytes of header information about the array’s size and number of dimensions. The actual array data follows immediately after the header information. This header/data scheme is repeated for every array.

Unlike entries in the variable table, array table entries have variable lengths, depending upon the number of dimensions of the array. For example, array QQ (a 4×4 array created by the program in Figure 4-2) has its array table entry at \$0431 in bank 1 RAM. The entry can be seen in the monitor dump in Figure 4-3. It contains the byte sequence \$51, \$51, \$86, \$00, \$02, \$00, \$05, \$00, \$05. The array name, QQ, is stored in the first two bytes and follows the variable-name conventions in Table 4-1. The next two bytes (\$86 and \$00) tell the interpreter that the combined length of the array’s header information and data block is \$0086 (134 decimal) bytes; this value is stored in standard 6502 low-byte/high-byte format.

The fifth byte in any array header contains the number of dimensions of the array, as specified by BASIC’s DIM command. Since

only one byte contains this value, there is a maximum of 255 possible array dimensions. In the example here, \$02 indicates that array QQ has two dimensions.

Starting with the sixth byte and progressing in two-byte steps, the array header contains the number of elements in each dimension of the array. Since array elements with a value of zero are allowed by Commodore BASIC — QQ(0,0) is legal — the number of elements for any particular dimension of the array is one greater than its corresponding parameter declared in the DIM statement. Thus, the array QQ is really a 5×5 array, since BASIC also accounts for array indexes of zero.

Contrary to normal practice in a 6502-based environment, these two-byte values (denoting the number of array elements per dimension) are stored in high-byte/low-byte format. Thus, the sequence of bytes \$00, \$05, \$00, \$05 (see the monitor dump in Figure 4-3) indicates that both the first and second dimensions of the array QQ contain five entries each. If the array had four dimensions, for example, then this part of the array header would be eight bytes long, with each two-byte value in reverse order.

The actual array data begins immediately after the array header. Floating-point values require five bytes, while storage of integer values requires only two bytes. Of course, there is no two-byte overhead for a variable name. A string array contains three-byte descriptors indicating the string length, followed by its starting address in conventional 6502 low-byte/high-byte format; strings themselves are stored in high RAM.

Integer and string (descriptor) data stored in non-array variables require less storage than their floating-point counterparts. Because all non-array variables contain a five-byte data field and since integer and string variables require less than five bytes of storage, \$00 filler bytes are appended to the integer or string descriptor data. Integer and string arrays do not require any such padding; each integer or string descriptor consumes only two or three bytes of storage, respectively. This is the *only* case in which use of integer variables in Commodore

BASIC actually reduces memory consumption. Remember, there is still a speed penalty (time taken for floating-point/integer conversions) associated with the use of any integer variable, array or not. Integer arrays should be used only when there is danger of running out of memory using ordinary floating-point arrays.

Floating-Point Numbers

Briefly, a floating-point number is the product of a mantissa multiplied by a power of two. Commodore BASIC floating-point numbers occupy five bytes, with the power of two contained in an exponent byte and the mantissa occupying the last four bytes.

The exponent byte of a Commodore floating-point number really contains the true exponent value with a +129 offset added to it so that both positive and negative powers of two can be represented. (A zero value of the exponent byte has a special meaning: the complete floating-point number is to be regarded as zero no matter what the four mantissa bytes contain. Storing zero to the exponent byte of any floating-point number is a quick way of zeroing the number itself.) For example, if a number's exponent byte contains \$83, the true power of two is $2^{83-129} = 2^{-46}$, or $1/2^{46}$. Two is raised to the second power, and the mantissa is thus multiplied by four. As another example, if the exponent byte contains \$71, the true power of two is $2^{71-129} = 2^{-58}$, or $1/2^{58}$. Two will be raised to the power of -15, resulting in the mantissa being multiplied by $1/32768$, or $1/2^{15}$.

The four-byte mantissa (32 bits) represents a number between 1 and 1.99999999... and is arranged in the order of most significant byte to least significant byte. Bit #7 of the most significant mantissa byte is, in turn, the most significant mantissa bit, and has a value of 2^{31} , or 1. Bit #0 of the least significant mantissa byte has a value of 2^{-31} . Thus, the mantissa is merely a sum of negative powers of two, just as an ordinary binary integer is a sum of positive powers of two. See Table 4-2 for a depiction of mantissa structure.

Table 4-2. Structure of Floating-Point Mantissa Bytes

| | | | | | | | | |
|----------|----------------|-----------------|-----------------|------------------|------------------|------------------|------------------|------------------|
| Mantissa | | | | | | | | |
| Byte #: | 1 | 2 | 3 | 4 | | | | |
| Bit #: | 7.....0 | 7.....0 | 7.....0 | 7.....0 | | | | |
| Bit | | | | | | | | |
| value: | 2 ⁰ | 2 ⁻⁷ | 2 ⁻⁸ | 2 ⁻¹⁵ | 2 ⁻¹⁶ | 2 ⁻²³ | 2 ⁻²⁴ | 2 ⁻³¹ |

With this combination of both exponent and mantissa, there are many ways of expressing a particular number in floating-point format. For example, the decimal number 10 can be expressed (using decimal numerals) as 1.25×2^3 , 2.50×2^2 , or even as $.15625 \times 2^6$.

Commodore BASIC arithmetic routines always adjust the value of a mantissa *after* math operations, so that its absolute value is greater than or equal to 1 (it can never be greater than 1.999999999). This “normalization” operation is done because numbers less than 1 contain leading zeroes in the mantissa, reducing the accuracy of the stored number; the leading zeroes can be thought of as pushing the least significant bits out of the mantissa. Of course, the exponent must be also be adjusted appropriately; every time the mantissa is doubled in order to bring it within the range 1 to 1.999999999, the exponent byte must be decremented by one. These normalization operations take place deep within the BASIC interpreter and need not concern programmers because all variable storage and calculation results have already been normalized.

Since a normalized floating-point number is guaranteed to be greater than 1, bit #7 of the first mantissa byte will always be high. This is a real benefit, because it means that this bit can be set to reflect the sign of the mantissa; thus, all math routines in BASIC ROM assume that this sign bit is high whenever the number is required for a calculation. If the sign needs to be taken into account, bit #7 of the

mantissa's MSB (most significant bit) will be checked; this bit will be high if the mantissa is negative. Table 4-3 shows how a decimal value of a floating-point number can be calculated if the exponent byte and (signed) mantissa bytes are known.

To obtain the true power of two by which the mantissa is multiplied, subtract \$81 (129 decimal) from the exponent byte. If this byte were zero originally, no further calculations would be needed, since

Table 4-3. Converting a Floating-Point Number to its Decimal Equivalent

| <i>Exponent</i> | <i>Mantissa</i> | | |
|--|---|------|----------------|
| | \$82 | \$49 | \$0F \$DA \$A3 |
| 1. Exponent byte = \$82. | "True" power of two = (\$82 - \$81) = \$01. Therefore, mantissa is multiplied by $2^1 = 2$. | | |
| 2. Mantissa sign is positive. | Bit #7 of the mantissa's first (most significant) byte is low, so mantissa is positive. | | |
| 3. "True" mantissa bytes used in conversion: Decimal equivalents: | \$C9, \$0F, \$DA, \$A3. 201, 15, 218, 163. | | |
| 4. Intermediate mantissa value | $= .7853981637 =$ $15 + \frac{218 + (163/256)}{256}$ $201 + \frac{\quad}{256}$ <hr/> 256 | | |
| 5. Actual mantissa value | = $(2 * .785398163) = 1.570796327$ | | |
| 6. Value of floating-point number | $= (\text{mantissa}) \times 2^{(\text{exponent} - 129)}$ $= 1.570796327 \times 2^1$ $= 3.141592654$ | | |

the whole floating-point number is regarded as zero. Next, bit #7 of the first mantissa byte is examined to check for the mantissa sign. If this byte is greater than \$80, then the mantissa is negative. All subsequent calculations will assume that this bit is set, since these calculations must account for that bit representing 2^0 instead of the mantissa sign. In other words, if the first mantissa byte is less than \$80, then \$80 (or decimal 128) should be added to it.

To calculate the mantissa value, the least significant (fourth) mantissa byte should be divided by 256. The preceding mantissa byte should be added to this quotient, and the resulting sum should be divided by 256. Continuing, the second mantissa byte is added to this quotient and the resulting sum is again divided by 256. This third add-and-divide operation should be repeated for the first mantissa byte in order to obtain the final result. The continued fraction shown in step 4 of Table 4-3 shows this operation more symbolically.

The mantissa value just calculated assumes that the 32 bits comprising the mantissa are aligned on byte boundaries. This is not the case, because the most significant mantissa bit (bit #7 of the first mantissa byte) has a place value of 2^0 , not 2^{-1} . The mantissa's calculated value thus falls between 0.5 and 1. For this discrepancy, the calculated value of the mantissa should be multiplied by two, effectively shifting the mantissa's 32 bits left by one bit position. The mantissa's most significant bit will then have a place value of 2^0 (which is 1) instead of 2^{-1} (0.5).

To convert a decimal number into five-byte floating-point format, the process outlined in Table 4-3 is reversed. First, the number is repeatedly multiplied or divided by two, as appropriate, until the absolute value of the result falls between 1 and 1.999999999. The number of divisions required to scale the decimal number is added to 129 (\$81 hex), whereas the number of multiplications required would be subtracted from 129. This result is the actual exponent byte.

The scaled value is again divided by two to make the mantissa value fall between 0.5 and .999999999 so that the mantissa's most

significant bit has a value of 2^{-1} . This value is then multiplied by 256, and the integer portion of the result becomes the first mantissa byte. Bit #7 of this byte should be set if the mantissa is negative or be clear if positive. The fractional portion of the preceding product is again multiplied by 256, and the result's integer portion becomes the second mantissa byte. In turn, the fractional portion of the previous product is multiplied by 256, and this result's integer portion is the third mantissa byte. Finally, the fractional portion of this product is multiplied by 256. This result, with the integer part rounded upward to the next integer if its fractional portion is greater than 0.5, will become the fourth byte of the mantissa. Figure 4-5 shows the steps necessary to convert a decimal number to a five-byte floating-point number.

The relative amounts of storage required by different numerical bases can be easily calculated. Comparing base 10 numbers with binary (base 2) by using the formula $\log(10)/\log(2)$, you can see that each decimal digit requires approximately 3.32 bits of storage. Since the mantissa is 32 bits long, the division $32/3.32$ indicates that the mantissa is accurate to over 9 decimal places.

BASIC Program Storage

The C-128's BASIC 7.0 interpreter reserves much of bank 1 RAM for storage of tokenized BASIC program text. Ordinarily, storage of BASIC program text begins at location \$1C01 (7169 decimal) and continues upward to location \$FEFF. Location \$1C00 should always contain a zero; otherwise, "syntax error" messages will frequently appear after legitimate BASIC commands have been entered. The start of BASIC program storage is moved up to \$4001 (16385) when-

Decimal number to be converted = 3.141592654

1. Scale the number into the 1 to 1.999999999 range by dividing it or multiplying it by two. The number of divisions or multiplications required should be recorded and added to (for divisions) or subtracted from (for multiplications) an offset of 129 (\$81 hex).

| | |
|--|---|
| Scaled value: | 1.570796327 |
| Number of times value scaled up/down by 2: | 1 |
| Value of exponent byte: | 129 + 1 = 130, decimal. = \$82, hex. |

2. Divide the scaled value (now within 1 to 1.999999999 range) by 2 so that the mantissa's most significant bit has a place value of 2^{-1} instead of 2^0 . Do not try to account for this division by adjusting the exponent byte again!

| | |
|-------------------|-------------|
| New scaled value: | .7853981635 |
|-------------------|-------------|

3. Calculate first mantissa byte by multiplying the result of step 2 by 256. The integer part of the product should have bit #7 clear if the mantissa is positive. Negative mantissas require bit #7 to be set.

| | |
|--|---------------|
| a. (Result from step 2) * 256 = | 201.061929900 |
| b. INT (result) = | 201 |
| c. FRAC(result) = | .061929900 |
| d. Calculated first mantissa byte = | \$C9 |
| e. Positive mantissa, so bit #7 cleared: | \$89 |

Figure 4-5. Converting a decimal number to its floating-point equivalent

-
4. Find second mantissa byte by multiplying fractional result obtained in step 3c by 256. The product's integer portion becomes mantissa byte #2. The functions INT(x) and FRAC(x) return the integer and fractional portions of a decimal number, respectively.
- a. $.061929900 * 256 = 15.854054400$
 b. $\text{INT}(x) = 15$
 c. $\text{FRAC}(x) = .854054400$
 d. Second mantissa byte: \$0F
5. Repeat step 4 to obtain the third mantissa byte, starting with the fractional portion obtained in step 4c.
- a. $.085405440 * 256 = 218.637926400$
 b. $\text{INT}(\text{result}) = 218$
 c. $\text{FRAC}(\text{result}) = .637926400$
 d. Third mantissa byte: \$DA
6. Multiply fractional result from step 5c by 256 to get fourth mantissa byte. Round the number upwards if a fractional portion is greater than 0.5.
- a. $.637926400 * 256 = 163.30915840$
 b. Rounded integer result = 163
 c. Fourth mantissa byte = \$A3

Thus, the floating-point equivalent of 3.141592654 is this sequence of bytes: \$81 \$89 \$0F \$DA \$A3.

Figure 4-5. Converting a decimal number to its floating-point equivalent
(continued)

ever BASIC 7.0-created bit-mapped graphics are being displayed; the \$1C00-\$1FFF region is then allocated for color storage, while the \$2000-\$3FFF region contains the actual bit map.

Each BASIC 7.0 program line contains a two-byte line “link” and a two-byte line number, both in standard 6502 low-byte/high-byte order. The actual tokens that comprise the program line follow, with the end of the line marked by a \$00 byte.

The link address stored at the start of every program line contains the address of the next line’s link bytes. These addresses are updated every time a new line is modified, and they are used by GOTO and GOSUB ROM routines to move through the BASIC program text quickly without having to examine every token byte. Figure 4-6 shows a memory dump of the simple program contained in Figure 4-7. The first two-byte line link can be seen at \$1C01 in the dump and indicates that the next line of BASIC begins at \$1C21. In turn, the link bytes at \$1C21 indicate that the third line of program text begins at \$1C2F. If any line of the BASIC program had been modified or deleted, all the lines would have been relinked — new link

```
>01C00 00 21 1C 0A 00 FE 25 3A EE 3A 81 20 49 B2 31 20:!....|X:n:. I21
>01C10 A4 20 35 3A 20 99 20 22 48 45 4C 4C 4F 22 3A 82:$ 5: . "HELLO":.
>01C20 00 2F 1C 14 00 41 41 B2 31 32 33 34 35 37 00 4F:./...AA2123457.0
>01C30 1C 1E 00 5A 5A 24 B2 22 2E 2E 2E 2E 2E 54 45 53:...ZZ$2".....TES
>01C40 54 20 53 54 52 49 4E 47 2E 2E 2E 2E 2E 2E 22 00 5C:T STRING.....".a
>01C50 1C 28 00 50 B2 CE 0A 28 41 41 29 00 00 00 00 00:.(.P2N.(AA).....
```

Figure 4-6. Monitor dump of tokenized BASIC program from Figure 4-7

```
10 FAST:DIRECTORY:FOR I=1 TO 5: PRINT "HELLO":NEXT
20 AA=123457
30 ZZ$=".....TEST STRING....."
40 P=POINTER(AA)
```

Figure 4-7. Simple BASIC 7.0 program used to demonstrate program storage

addresses would have been calculated for every line. This is one of the reasons for the slight delay in system response when lines are added to a long BASIC program.

All Commodore BASIC functions and commands are stored in tokenized form, which means that only one or two bytes are required to store any Commodore BASIC 7.0 function or command. Previous implementations of Commodore BASIC needed only one token byte per function or command, but the large set of C-128 BASIC 7.0 instructions requires that two bytes, \$FE and \$CE, are set aside as *escape tokens*. (These are not to be confused with screen editor escape sequences.) When either of these tokens are encountered by BASIC as it is executing either a program or a line entered in direct mode, BASIC switches to an alternate table of functions (using the \$FE token) or statements (using the \$CE token). For example, the FAST instruction in line 10 of the BASIC program in Figure 4-7 is stored as a two-byte token \$FE \$25 at location \$1C05. Similarly, the POINTER(x) function in line 40 is stored as the two-byte sequence \$CE \$0A at location \$1C55. Table 4-4 contains a list of all Commodore BASIC 7.0 tokens, including those prefixed by a \$FE or \$CE escape token.

The commands QUIT and OFF, although not documented by Commodore in their *System Guide*, both return an “UNIMPLEMENTED COMMAND ERROR” message. Apparently these statements will be used in future C-128 systems, and Commodore is merely providing room for them in the BASIC 7.0 instruction table. However, the *System Guide* makes no mention of the useful RREG statement. Its syntax is

```
RREG a,b,c,d
```

Not all parameters are required; a shortened RREG a or RREG a,b are also legal. RREG returns the contents of the 8502's .A, .X, .Y, and status registers (in that order) to ordinary BASIC numeric variables. It is normally used after returning from a SYS call to pass parameters and/or data from a machine language routine back to BASIC variables.

Table 4-4. The BASIC 7.0 Token Set

| <i>One-byte tokens</i> | | <i>One-byte tokens</i> | |
|------------------------|---------|------------------------|--------------------|
| 128 | END | 165 | FN |
| 129 | FOR | 166 | SPC(|
| 130 | NEXT | 167 | THEN |
| 131 | DATA | 168 | NOT |
| 132 | INPUT# | 169 | STEP |
| 133 | INPUT | 170 | + |
| 134 | DIM | 171 | - |
| 135 | READ | 172 | * |
| 136 | LET | 173 | / |
| 137 | GOTO | 174 | † (exponentiation) |
| 138 | RUN | 175 | AND |
| 139 | IF | 176 | OR |
| 140 | RESTORE | 177 | > (greater than) |
| 141 | GOSUB | 178 | = |
| 142 | RETURN | 179 | < (less than) |
| 143 | REM | 180 | SGN |
| 144 | STOP | 181 | INT |
| 145 | ON | 182 | ABS |
| 146 | WAIT | 183 | USR |
| 147 | LOAD | 184 | FRE |
| 148 | SAVE | 185 | POS |
| 149 | VERIFY | 186 | SQR |
| 150 | DEF | 187 | RND |
| 151 | POKE | 188 | LOG |
| 152 | PRINT# | 189 | EXP |
| 153 | PRINT | 190 | COS |
| 154 | CONT | 191 | SIN |
| 155 | LIST | 192 | TAN |
| 156 | CLR | 193 | ATN |
| 157 | CMD | 194 | PEEK |
| 158 | SYS | 195 | LEN |
| 159 | OPEN | 196 | STR\$ |
| 160 | CLOSE | 197 | VAL |
| 161 | GET | 198 | ASC |
| 162 | NEW | 199 | CHR\$ |
| 163 | TAB(| 200 | LEFT\$ |
| 164 | TO | | |

Table 4-4. The BASIC 7.0 Token Set (*continued*)

| <i>One-byte tokens</i> | | <i>One-byte tokens</i> | |
|------------------------|--------------------------------|------------------------|-------------------------------|
| 201 | RIGHT\$ | 235 | DO |
| 202 | MIDS | 236 | LOOP |
| 203 | GO | 237 | EXIT |
| 204 | RGR | 238 | DIRECTORY |
| 205 | RCLR | 239 | DSAVE |
| 206 | function escape token, \$CE | 240 | DLOAD |
| 207 | JOY | 241 | HEADER |
| 208 | RDOT | 242 | SCRATCH |
| 209 | DEC | 243 | COLLECT |
| 210 | HEX\$ | 244 | COPY |
| 211 | ERR\$ | 245 | RENAME |
| 212 | INSTR | 246 | BACKUP |
| 213 | ELSE | 247 | DELETE |
| 214 | RESUME | 248 | RENUMBER |
| 215 | TRAP | 249 | KEY |
| 216 | TRON | 250 | MONITOR |
| 217 | TROFF | 251 | USING |
| 218 | SOUND | 252 | UNTIL |
| 219 | VOL | 253 | WHILE |
| 220 | AUTO | 254 | command escape token, \$FE |
| 221 | PUDEF | 255 | pi (π) token |
| 222 | GRAPHIC | | |
| 223 | PAINT | | |
| 224 | CHAR | | |
| 225 | BOX | | |
| 226 | CIRCLE | | |
| 227 | GSHAPE | | |
| 228 | SSHAPE | | |
| 229 | DRAW | | |
| 230 | LOCATE | | |
| 231 | COLOR | | |
| 232 | SCNCLR | | |
| 233 | SCALE | | |
| 234 | HELP | | |

Table 4-4. The BASIC 7.0 Token Set (*continued*)

Function tokens preceded by \$CE escape token

| | |
|----|----------|
| 2 | POT |
| 3 | BUMP |
| 4 | PEN |
| 5 | RSPPOS |
| 6 | RSPRITE |
| 7 | RSPCOLOR |
| 8 | XOR |
| 9 | RWINDOW |
| 10 | POINTER |

Statement tokens preceded by \$FE escape token

| | |
|----|-----------|
| 2 | BANK |
| 3 | FILTER |
| 4 | PLAY |
| 5 | TEMPO |
| 6 | MOVSPR |
| 7 | SPRITE |
| 8 | SPRCOLOR |
| 9 | RREG |
| 10 | ENVELOPE |
| 11 | SLEEP |
| 12 | CATALOG |
| 13 | DOPEN |
| 14 | APPEND |
| 15 | DCLOSE |
| 16 | BSAVE |
| 17 | BLOAD |
| 18 | RECORD |
| 19 | CONCAT |
| 20 | DVERIFY |
| 21 | DCLEAR |
| 22 | SPRSV |
| 23 | COLLISION |
| 24 | BEGIN |
| 25 | BEND |
| 26 | WINDOW |

Table 4-4. The BASIC 7.0 Token Set (*continued*)

Statement tokens preceded by \$FE escape token

| | |
|----|--|
| 27 | BOOT |
| 28 | WIDTH |
| 29 | SPRDEF |
| 30 | QUIT (unimplemented command) |
| 31 | STASH |
| 32 | (unused, in order to bypass spaces continued in program text) |
| 33 | FETCH |
| 34 | (unused, in order to bypass quote marks continued in program text) |
| 35 | SWAP |
| 36 | OFF (unimplemented command) |
| 37 | FAST |
| 38 | SLOW |

BASIC 7.0 Dictionary

This dictionary explains the syntax of all the C-128 BASIC statements. They are presented in alphabetical order and include both functions and I/O commands. Each entry in this “dictionary” also lists the corresponding token value, as well as the ROM entry address in hex. The information will be presented in this format:

| Name | Token Value | ROM Entry Address |
|-------------|--------------------|--------------------------|
|-------------|--------------------|--------------------------|

The token value will also contain the appropriate escape-function byte (\$CE or \$FE) where necessary. Optional parameters are usually enclosed within square brackets.

BASIC 7.0 Statements

| | | |
|---------------|-----------------|---------------|
| APPEND | \$FE\$OE | \$A134 |
|---------------|-----------------|---------------|

The APPEND# statement opens an existing sequential diskette file and allows new data to be added at the end of the file.

Format:

APPEND#lf,"filename"[,Dd][ON Uz]

The APPEND# statement opens sequential data file *filename* on the diskette in drive *d* and positions file pointers beyond the current end of file. Subsequent PRINT# statements referencing logical file *lf* can then write additional data, which gets appended to the end of the file. If no disk drive is specified (*d* is absent), drive 0 is assumed.

Example:

```
APPEND#1,"CALC"  
PRINT#1,A
```

Open sequential file "CALC" as logical file #1 on drive 0. Write variable A contents to the end of the file

```
APPEND#3,"TALK",D1  
PRINT#3,"123"
```

Open sequential file "TALK" as logical file #3. The string "123" is added to the end of the file

AUTO \$DC \$5975

Initiates auto line numbering. The number that follows AUTO indicates the line increments.

Example:

```
AUTO 10
```

This will start incrementing line numbers by 10 after you enter the first line of your program. AUTO entered without parameters disables automatic numbering.

BACKUP \$F6 \$A37C

The BACKUP statement duplicates an entire diskette. The duplicate and original have the same header, disk name, identification number, directory, and files.

Format:

BACKUP Ds TO Dd [ON Uz]

The diskette in drive *s* is duplicated. The duplicate diskette is generated in drive *d*. Duplicating the entire diskette takes a couple of minutes.

Example:

| | |
|-----------------|---|
| BACKUP D0 to D1 | <i>Duplicate contents of diskette in drive 0 to diskette in drive 1</i> |
| BACKUP D1 to D0 | <i>Duplicate contents of diskette in drive 1 to diskette in drive 0</i> |

Caution: All files on the diskette must be properly closed before the diskette is backed up. Additionally, it can only be used on a dual disk drive unit.

BANK \$FE \$02 \$6BC9

Switches the system from one bank to another. Options (0-15). The default bank is 15. The only BASIC commands affected by BANK are PEEK, POKE, SYS, and USR. See Chapter 2 for more information on C-128 memory banks and configurations.

Example:

BANK 12

Switches the system from the current memory bank to bank 12.

BEGIN \$FE \$18 \$796C

Signals the start of a conditional subroutine. Similar to IF/THEN, but allows you to include several line numbers between BEGIN and BEND. (The ROM code for BEGIN is actually part of the IF code, and begins at \$52D8.)

Example:

```
10 IF A<=12 THEN BEGIN: PRINT "THIS IS VALID
MONTH"
```

```

20 PRINT A$(A)
30 BEND: GOTO 50
40 PRINT "NOT A VALID MONTH"
50 END

```

BEND \$FE \$19 \$528F

Signals the end of a BEGIN/BEND subroutine (see BEGIN). (The ROM code for BEND is really part of that used by the DATA statement, whose code begins at \$528F.)

BLOAD \$FE \$11 \$A218

Loads a binary file from disk or tape. If a string variable contains the filename, the variable must be enclosed within parentheses.

Format:

BLOAD "*filename*" [,Ddrive#][,Udevice#][,Bbank#][,Pstart
address]

Example:

BLOAD "CHARACTERS"

BOOT \$FE \$27 \$7335

Loads a binary program and runs it.

Format:

BOOT "*filename*" [,Ddrive#][,Udevice#]

Example:

BOOT "WRDPROC"

This loads and runs a machine language program named WRDPROC.

BOX **\$E1** **\$62B7**

Draws a box in the selected position and size.

Format:

BOX *C, XA, YA, XB, YB, R, F*

where *C* is the box color, *XA* and *YA* locate the upper left-hand corner of the box, *XB* and *YB* locate the lower right-hand corner of the box, *R* is the rotation angle (0 is no rotation), and *F* is fill. (0 = no; 1 = yes; default is 0.)

BSAVE **\$FE \$10** **\$A1 C8**

Saves a binary program or data file on disk or tape.

Format:

BSAVE "*filename*", *B bank#*, *P start address*, *TO P end address*

CATALOG **\$FE \$0C** **\$A07E**

Same as directory command—displays the contents of the disk.

Format:

CATALOG

CHAR **\$E0** **\$67D7**

The CHAR statement can be used to display characters on either a bit-mapped or a text screen. It has the following syntax:

CHAR (*color source*) , *X*, *Y* (*,string*) (*,Reverse flag*)

The *color source* parameter can be either 0 (foreground color) or 1

(background color). The *X* and *Y* parameters contain the character column and row position of the text string's first character; they can range from 0 to 79 for the *X* dimension and from 0 to 24 for the *Y* dimension. The string can contain cursor movements and other special characters, but these—except for upper- and lowercase selection—will only have an effect on a text screen. The *Reverse flag* (0 = off, 1 = on) functions similarly to the RVS On function available from the C-128 keyboard.

Example:

CHAR 10, 20, "Hello
there!!!"

*Prints the string in quotes to the current
screen selected by the GRAPHIC n
statement*

CHAR is used slightly differently in multicolor bit-mapped mode. The GRAPHIC and COLOR statements are used to select this mode with its two corresponding foreground colors. To display the text in the foreground color #1, set the *color source* parameter to 0 and the *Reverse flag* to 0. If text needs to be displayed in foreground color #2, ensure that the *color source* parameter is again set to zero, but set the *Reverse flag* to 1.

CIRCLE

\$E2

\$668E

Draws a circle in the selected position and size.

Format:

CIRCLE *C*, *X*, *Y*, *XR*, *YR*, *B*, *E*, *L*

where *C* is the box color, *X* and *Y* locate the center of the circle, *XR* and *YR* specify the x and y radii, *B* is the beginning arc angle, *E* is the ending arc angle, *R* is the rotation of the circle, and *L* is the length of the sides (if any).

NOTE: If you specify a side length (*L*) when the circle will become a polygon with the number of sides determined by 360/*L*.

CLOSE

\$AO

\$919A

The CLOSE statement closes a logical file.

Format:

CLOSE *lf*

The CLOSE statement closes logical file *lf*. Every file should be closed after all file accesses have been completed. An open logical file may be closed only once. The particular operations performed in response to a CLOSE statement depend on the open file's physical device and the type of access that occurred.

Example:

CLOSE 1
CLOSE 14

Close logical file 1
Close logical file 14

CLR

\$9C

\$51F8

The CLR statement sets all numeric variables to zero and assigns null values to all string variables. All array space in memory is released. This is equivalent to turning the computer off, then turning it back on and reloading the program into memory. CLR closes all logical files that are currently open within the executing program.

Format:

CLR

A program will continue to run following execution of a CLR statement if the statement's execution does not adversely affect program logic.

Example:

100 CLR

CMD \$9D \$5540

The CMD statement sends all output that would have gone to the display to another specified unit. Output goes to that unit, instead of the display, until an empty PRINT# statement specifying the same logical file number that was opened is executed. At least one PRINT# statement must follow a CMD statement.

Format:

CMD lf

The CMD statement assigns a line printer output channel to logical file *lf*. After execution of a CMD statement, PRINT and LIST both print data instead of displaying it.

Example:

| | |
|----------|---|
| OPEN 5,4 | <i>Open logical file 5 selecting the printer</i> |
| CMD 5 | <i>Direct subsequent output to the printer</i> |
| LIST | <i>Print the program listing</i> |
| PRINT#5 | <i>Print a carriage return and deselect the printer</i> |
| CLOSE 5 | <i>Close logical file 5</i> |

COLLECT \$F3 \$A32F

The COLLECT statement recreates a block availability map (BAM) for all files on the diskette. Improperly closed files are closed or deleted.

Format:

COLLECT [D*d*][ON U*y*]

The diskette on drive *d* is collected. If the *Dd* parameter is absent, drive 0 is assumed.

Example

| | |
|------------|--|
| COLLECT | <i>Collects space on diskette in last drive accessed</i> |
| COLLECT D0 | <i>Collects space on diskette in drive 0</i> |
| COLLECT D1 | <i>Collects space on diskette in drive 1</i> |

COLLISION \$FE \$17 \$7164

System variable. Detects sprite-sprite collisions and sprite-object collisions on the screen. There are two options:

- 0 = Detect sprite to sprite collisions
- 1 = Detect sprite to background collisions

Format:

COLLISION T,*line#*

where T is the type of collision to detect and *line#* is the line you want to execute if a collision occurs.

COLOR \$E7 \$69E2

Allows you to specify the color of various screen objects.

Format:

COLOR O, C

where O is the object and C is the color. The options for O are

- O = 0: Color of 40-column background
- O = 1: Color of 40-column foreground
- O = 2: Multi-color #1
- O = 3: Multi-color #2
- O = 4: Color of 40-column border
- O = 5: Character color
- O = 6: Color of 80-column background

The color options in 40-column mode are

- 1 = BLACK
- 2 = WHITE
- 3 = RED
- 4 = CYAN
- 5 = PURPLE
- 6 = GREEN
- 7 = BLUE
- 8 = YELLOW
- 9 = ORANGE
- 10 = BROWN
- 11 = LT RED
- 12 = DARK GREY
- 13 = MED GREY
- 14 = LT BLUE
- 15 = LT GREY

The color options in 80-column mode are

- 1 = BLACK
- 2 = WHITE
- 3 = DK RED
- 4 = LT CYAN
- 5 = LT PURPLE
- 6 = DK GREEN
- 7 = DK BLUE
- 8 = LT YELLOW
- 9 = DK PURPLE
- 10 = DARK YELLOW
- 11 = LT RED
- 12 = DARK CYAN
- 13 = MED GREY
- 14 = LT BLUE
- 15 = LT GREY

CONCAT \$FE \$13 \$A362

The CONCAT statement concatenates two data files.

Format:

CONCAT[Ds]"*sourcefile*" TO D*d*,]"*destfile*"[ON Uz]

The contents of *sourcefile* on the diskette in drive *s* are concatenated onto the end of *destfile* on the diskette in drive *d*. The file named *sourcefile* does not change. The file named *destfile* keeps its original contents, with the contents of *sourcefile* tacked on at the end. If drive numbers *s* and/or *d* are not specified, then drive 0 is assumed.

Caution: Files must be closed before they are concatenated.

Example:

CONCAT "FIRST" TO
"SECOND"

The contents of file FIRST is concatenated on the end of file SECOND. Both files are on the diskette in drive 0

CONCAT D1,"ABC"
TO D0,"XYZ"

The contents of file ABC on the diskette in drive 1 is concatenated on the end of file XYZ on the diskette in drive 0

CONT \$9A \$5A60

The CONT statement, typed at the keyboard in immediate mode, resumes program execution after a BREAK.

Format:

CONT

A break is caused by execution of a STOP statement or an END statement that has additional statements following it. Depressing the RUN-STOP key while a program is running also causes a break.

Program execution continues at the exact point where the break occurred.

Typing **CONT** after this break reexecutes the **INPUT** statement.

Example:

CONT

COPY \$F4 \$A346

The **COPY** statement copies a single diskette file, or all the files on a diskette.

Format:

COPY [D*s*,]“*sourcefile*”] **TO** [D*d*,] (“*destfile*”)[**ON** U*z*]

If the **COPY** statement is used to copy a single file, then the file named *sourcefile* on the diskette on drive *s* is copied to a new file named *destfile* on the diskette in drive *d*; the filenames *sourcefile* and *destfile* must be present, but if drive numbers *s* and/or *d* are absent, drive 0 is assumed.

The **COPY** statement can also be used to copy all files from the diskette in one drive to the diskette in the other drive. To use the **COPY** statement in this fashion, filenames *sourcefile* and *destfile* must be absent, but drive numbers *s* and *d* must be present and different.

If the name of a source file that is being copied exists on the destination diskette, then the copy will be aborted at that file, and a “**FILE ALREADY EXISTS**” error will be reported.

COPY does not modify any files previously on the destination diskette.

Caution: A file must be closed before it is copied.

Example:

| | |
|---------------------------------|---|
| COPY D1 TO D0 | <i>Copy all files on the diskette in drive D1 to the diskette in drive D0. (DOS 2.0 and higher releases only)</i> |
| COPY D1, "MAJOR" TO D1, "MINOR" | <i>Create MINOR file on the diskette in drive D1</i> |

DATA \$83 \$528F

The DATA statement declares constants that are assigned to variables by READ statements.

Format:

DATA *constant*[,*constant,constant,....,constant*]

DATA statements may be placed anywhere in a program. The DATA statement specifies either numeric or string constants. String constants are usually enclosed in double quotation marks; the quotes are not necessary unless the string contains graphic characters, blanks (spaces), commas, or colons. Blanks, commas, colons, and graphic characters are ignored unless the string is enclosed in quotes. A double quotation mark cannot be represented in a DATA string; it must be specified using a CHR\$(34) function. The DATA statement is valid in program mode only.

Example:

| | |
|---------------------|--|
| 10 DATA NAME,"C.D." | <i>Defines two string variables</i> |
| 50 DATA 1E6,-10,XYZ | <i>Defines two numeric variables and one string variable</i> |

Refer to the READ statement for a description of how DATA statement constants are used within a program. The ROM code used by the DATA statement is also used by BEND.

DCLEAR \$FE \$15 \$A322

Closes all open disk channels and files on the selected device.

Example:

DCLEAR DD

Closes all the files and open channels on drive 0.

DCLOSE \$FE \$OF \$A16F

DCLOSE closes a single file or all the files currently open on a disk unit. (Also see CLOSE.)

*Format:*DCLOSE#*lf* [ON *Uz*]The DCLOSE statement closes logical file *lf*. If the logical file number is not specified, all currently open diskette files are closed.*Example:*

| | |
|--------------|--|
| DCLOSE | <i>Closes all open diskette files</i> |
| DCLOSE#1 | <i>Closes the diskette file identified by logical file 1</i> |
| DCLOSE ON UB | <i>Closes all open diskette files on physical unit #8</i> |

DEF FN \$96 \$84FA

The DEF function (DEF FN) allows special purpose functions to be defined and used within BASIC programs. The token byte for the FN function is \$A5.

*Format:*DEF FN*nvar*(*arg*)=*expression*Floating-point variable *nvar* identifies the function, which is subsequently referenced using the name FN*nvar*(*data*). (A syntax error is reported if *nvar* is a string or integer variable.)The function is specified by *expression*, which can be any arithmetic expression containing any combination of numeric constants,

variables, or operators. The dummy variable name *arg* can (and usually does) appear in expression.

The *arg* is the only variable in *expression* that can be specified when `FNnvar(data)` is referenced. Any other variables in *expression* must be defined before `FNnvar(data)` is referenced for the first time. `FNnvar(data)` evaluates *expression* using *data* as the value for *arg*.

The entire DEF FN statement must appear on a single program line; however, a previously defined function can be included in *expression*, so user-defined functions of any complexity can be developed.

The function name *var* can be reused and therefore redefined by another DEF FN statement appearing later in the same program.

The DEF FN definition statement is illegal in immediate mode. However, a user-defined function that has been defined by a DEF FN statement in the current stored program can be referenced in an immediate mode statement.

Example:

```
10 DEF FNC(R)=
  π*R*2
```

Defines a function that calculates the circumference of a circle. It takes a single argument R, the radius of the circle, and returns a single numeric value, the circumference of the circle

```
?FNC(1)
55 IF FNC(X)>60 GOTO
150
```

*Prints 3.141159265 (the value of pi)
Uses the value calculated by the user-defined function FNC as a branch condition. The current contents of variable X are used when calculating the user-defined function*

DELETE

\$F7

\$5E87

Deletes a range of BASIC lines from the current program. The options are:

DELETE *L#*

This deletes a single line as specified by *L#*.

DELETE L#-L#

deletes the range of lines between the two *L#*'s.

DELETE L#-

deletes all of the lines that follow the *L#*.

DELETE -L#

deletes all of the line prior to the *L#*.

Example:

DELETE 1000-2000

will delete lines 1000 through 2000.

DIM **\$86** **\$587B**

The dimension statement DIM allocates space in memory for array variables.

Format:

DIM *var(sub)*[,*var(sub)*,...,*var(sub)*]

The DIM statement identifies arrays with one or more dimensions as follows:

| | |
|---|----------------------------|
| <i>var(sub_i)</i> | Single-dimensional array |
| <i>var(sub_isub_j)</i> | Two-dimensional array |
| <i>var(sub_isub_jsub_k)</i> | Multiple-dimensional array |

Arrays with more than 11 elements must be dimensioned in a DIM statement. Arrays with 11 elements or less (subscripts 0 through

10 for a one-dimensional array) may be used without being dimensioned by a DIM statement; for such arrays, 11 array spaces are automatically allocated in memory when the first array element is encountered in the program. An array with more than 11 elements must occur in a DIM statement before any other statement references an element of the array.

If an array is dimensioned more than once, or if an array having more than 11 elements is not dimensioned, an error occurs and the program is aborted. A CLR statement allows a DIM statement to be reexecuted.

Example:

| | |
|-----------------------------|---|
| 10 DIM A(3) | <i>Dimension a single-dimensional array of 3 elements</i> |
| 45 DIM X\$(44,2) | <i>Dimension a two-dimensional array of 88 elements</i> |
| 1000 DIM MU(X,3*B),N(12) | <i>Dimension a two-dimensional array of X times 3*B elements and a single-dimensional array of 12 elements. X and B must have been assigned values before the DIM statement is executed</i> |

DIRECTORY \$EE \$A07E

The DIRECTORY statement displays directories for diskettes in one or both drives. The word CATALOG may be used instead of DIRECTORY.

Format:

DIRECTORY[Dd][ON Uz]

The directory for the diskette in drive *d* is displayed. If the *Dd* parameter is absent, directories for the diskettes in both drives are displayed. If a selected drive contains no diskette, an error status is reported.

The DIRECTORY statement is usually executed if immediate mode.

Example:

| | |
|--------------|--|
| DIRECTORY | <i>Displays the directory of drive 0 and drive 1</i> |
| DIRECTORY D1 | <i>Displays the directory of drive 1</i> |

Printing a Directory A directory can be printed instead of being displayed by opening a printer channel before executing the DIRECTORY statement. Here is the required immediate mode statement sequence:

| | |
|-----------|---|
| OPEN 4,4 | <i>Open the printer specifying logical file 4</i> |
| CMD 4 | <i>Deflect display output to the printer</i> |
| DIRECTORY | <i>Print directories for diskettes in both drives</i> |
| PRINT#4 | <i>Deflect output back to the display</i> |
| CLOSE 4 | |

DLOAD \$FO \$A1A7

The DLOAD statement loads a BASIC program from a diskette into memory (also see LOAD).

Format:

DLOAD "filename" [,Ddrive#][(ON), Udevice#]

The DLOAD statement loads program file "filename" from the diskette in drive *d* into computer memory. If *d* is not present, drive 0 is assumed.

Example:

| | |
|-------------------------|---|
| DLOAD "CALC" | <i>Load CALC file from drive 0</i> |
| DLOAD "TIME",D1 | <i>Load TIME file from drive 1</i> |
| DLOAD (A\$) | <i>Load file whose name is in A\$. Note parentheses required.</i> |
| DLOAD"PROG",DO ON UB | <i>Load PROG file from drive 0 on the disk unit</i> |

Note that if the filename is contained within a string variable, the

variable must be surrounded by parentheses in DLOAD or DSAVE statements.

DOPEN \$F\$OD \$A11D

DOPEN opens a data file for a read and/or write access.

Format:

DOPEN#*lf*,"*filename*"[,*Ly*][,*Dd*][ON Uz][,*W*]

The DOPEN statement opens data file *filename* on the diskette in drive *d*, assigning to it logical file number *lf*. If *d* is not specified, then drive 0 is assumed. If *Ly* is not present, then a sequential file is assumed. The sequential file is opened for a write access if *W* is not present; it is opened for a read access if *W* is present.

If *Ly* is present, then a relative file is assumed with a record length of *y* bytes. Relative files are opened for read or write accesses; therefore the *W* parameter cannot be present.

Example:

| | |
|---------------------------|--|
| DOPEN#1,"PRIZES" | <i>Opens the sequential file named PRIZES on drive 0 for a read access</i> |
| DOPEN#6,"SNAKE" L30,D1 | <i>Opens the relative file named SNAKE, with a record length of 30, for read and write accesses. The file is on drive D1</i> |

DRAW \$E5 \$6797

Draws a line on the graphic screen.

Format:

DRAW *C*, *XA*, *YA* TO *XB*, *YB*

where *C* is the color of the line, *XA* and *YA* specify one end of the line, and *XB* and *YB* specify the other end of the line. Any (or all) of the

coordinate parameters (XA, YA, \dots) can be expressed as relative, instead of absolute, coordinates by the addition of a + or - sign.

Options:

`DRAW C, XA, YA, R; A`

This is the polar form of the DRAW statement. XA and YA are the Cartesian coordinates of the initial point, while R is a radius and A is an angle value.

`DRAW C, XA, YA TO XB, YB TO XC, YC . . .`

In this case, a continuous line is drawn between the points specified by each X and Y pair. Again, relative coordinates are allowed.

DSAVE \$EF SA18C

The DSAVE statement writes a BASIC program file from memory onto a diskette (also see SAVE).

Format:

`DSAVE"filename"[,Dd][ON Uz]`

The DSAVE statement saves the BASIC program currently in memory, writing it to a new file named *filename* on the diskette in drive d . If d is not present, drive 0 is assumed.

Example:

| | |
|------------------------------|---|
| <code>DSAVE"TRUE"</code> | <i>Write program file TRUE to diskette in drive 0</i> |
| <code>DSAVE"FALSE",D1</code> | <i>Write program file FALSE to diskette in drive 1</i> |
| <code>DSAVE(A\$)</code> | <i>Save file whose name is stored in A\$. Note that parentheses are required.</i> |

END \$80 \$4BCD

The END statement terminates program execution and returns the computer to immediate mode.

Format:

END

The END statement can provide a program with one or more termination points at locations other than the physical end of the program. END statements can be used to terminate individual programs when more than one program is in memory at the same time. An END statement at the physical end of the program is optional. The END statement is used in program mode only.

Example:

20001 END

ENVELOPE \$FE \$0A \$70C1

Specifies a set of sound parameters for the PLAY command.

Format:

ENVELOPE *N,A,D,S,R,W,P*

where *N* is the envelope number (0-9), *A* is the attack value (0-15), *D* is the decay value (0-15), *S* is the sustain value (0-15), *R* is the release value (0-15), *W* is the waveform number (0-4), and *P* is the pulse width (0-4095).

EXIT \$ED \$6039

When encountered in a DO-WHILE/UNTIL loop, EXIT transfers program execution to the first statement following the LOOP statement following the LOOP statement. Normal program execution resumes.

Format:

EXIT

FAST \$25 \$77B3

Switches the computer from slow mode (1 MHz) to fast mode (2 MHz). Does not work properly with I/O functions or VIC-II graphics, although sounds can be produced if the SID sound chip is accessed directly—either with PEEK/POKE statements or with machine language. BASIC sound and music statements will not function properly in fast mode.

Format:

FAST

FETCH \$FE \$21 \$AA24

The FETCH statement transfers contents of RAM in an expansion cartridge to C-128 memory.

Format:

FETCH #bytes, intsa, expb, expsa

The #bytes parameter indicates the number of bytes to be transferred to C-128 RAM and can range from 1 to 65536. The parameter *intsa* is the starting address of the region of C-128 RAM that will receive data. The *expb* parameter is the bank number (0-3) of the desired 64K segment within the expansion RAM cartridge; the *expsa* parameter is the starting address of the region in expansion RAM that contains the data to be transferred.

Care should be taken to avoid FETCH transfers that result in data overlaying the \$D000-\$DFFF 4K I/O block. Register contents of the various I/O chips will probably be changed if the address range of a FETCH transfer includes this I/O block, and may even lock up the system. See the description of the DMA_CALL routine in Chapter 6 for more information.

FILTER \$FE \$03 \$7046

Specifies a set of filter parameters for the SID chip.

Format:

FILTER *F, L, B, H, R*

where *F* is the cut-off frequency, *L* selects the low pass filter, *B* selects the band pass filter, *H* selects the high pass filter, and *R* selects resonance.

FOR \$B1 \$5DF9
NEXT \$B2 \$57F4
STEP \$A9 \$5E4D

All statements between the FOR statement and the NEXT statement are reexecuted the same number of times.

Format:

FOR *nvar* = *start* TO *end* STEP *increment*
[*statements in loop*]
NEXT[*nvar*]

where

| | |
|--------------|--|
| <i>nvar</i> | is the index of the loop. It holds the current loop count. <i>nvar</i> is often used by the statements within the loop. |
| <i>start</i> | is a numeric constant, variable, or expression that specifies the beginning value of the index. |
| <i>end</i> | is a numeric constant, variable, or expression that specifies the ending value of the index. The loop is completed when the index value is equal to the end value, or when the index value |

is incremented or decremented past the end value.

increment

if present, is a numeric constant, variable, or expression that specifies the amount by which the index variable is to be incremented with each pass. The step may be incremental (positive) or decremental (negative). If STEP is omitted, the increment defaults to 1.

The *nvar* may optionally be included in the NEXT statement. A single NEXT statement is permissible for nested loops that end at the same point. The NEXT statement then takes the form

NEXT *nvar*₁,*nvar*₂...

although this executes more slowly than a series of NEXT statements without parameters.

The FOR/NEXT loop will always be executed at least once, even if the beginning *nvar* value is beyond the end *nvar* value. If the NEXT statement is omitted and no subsequent NEXT statements are found, the loop is executed once.

The *start*, *end*, and *increment* values are read only once, on the first execution of the FOR statement. You cannot change these values inside the loop. You can change the value of *nvar* within the loop. This may be used to terminate a FOR/NEXT loop before the *end* value is reached: set *nvar* to the *end* value and on the next pass the loop will terminate itself. Do not jump out of the FOR/NEXT loop with a GOTO. Do not start the loop outside a subroutine and terminate it inside the subroutine.

FOR/NEXT loops may be nested. Each nested loop must have a different *nvar* variable name. Each nested loop must be wholly contained within the next outer loop; at most, the loops can end at the same point.

The ROM code for STEP is actually part of that for the FOR statement; if no STEP instruction and increment value are given, the code at \$5E4D assumes a STEP value of one.

GET \$A1 \$5612

The GET statement receives single characters as input from the keyboard.

Format:

GET *var*

The GET statement can be executed in program mode only. When a GET statement is executed, *var* is assigned a 0 value if numeric, or a null value if a string. Any previous value of the variable is lost. Then GET fetches the next character from the keyboard buffer and assigns it to *var*. If the keyboard buffer is empty, *var* retains its 0 or null value.

GET is used to handling one-character responses from the keyboard. GET accepts the RETURN key as input and passes the value (CHR\$(13)) to *var*.

If *var* is a numeric variable and no key has been pressed, 0 is returned. However, a 0 is also returned when 0 is entered at the keyboard.

If *var* is a numeric variable and the character returned is not a digit (0-9), a “?SYNTAX ERROR” message is generated and the program aborts.

The GET statement may have more than one variable in its parameter list, but it is hard to use if it has multiple parameters.

GET *var,var,...,var*

Example:

```
10 GET CS
10 GET D
10 GET A,B,C
```

GETKEY \$A1 \$F9 \$5612

Gets a keystroke from the keyboard. The difference between GETKEY and GET is that GETKEY pauses the computer while it waits

for entry and GET does not. The following lines would be identical in operation:

```
10 GET A$: IF A$="" THEN GOTO 10
10 GETKEY A$
```

GETKEY uses much of the same ROM code as GET; the beginning of the machine-code routine distinguishes between GET, GETKEY, and GET#.

GET# \$A1 \$23 \$5612

The GET external statement (GET#) receives single characters as input from an external storage device identified via a logical file number.

Format:

GET#lf,var

The GET# statement can only be used in program mode. GET# fetches a single character from an external device and assigns this character to variable *var*. The external device is identified by logical file number *lf*. This logical file must have been previously opened by an OPEN statement.

GET# and GET statements handle variables and data input identically. For details see the GET statement description. GET# uses much of the same ROM code as GET; the beginning of the machine code for the GET routine distinguishes among GET, GETKEY, and GET#.

Example:

```
10 GET#4,C$:IF
C$="" GOTO 10
```

Get a keyboard character. Reexecute if no character is present

G064 \$CB + "64" \$5A3D

Switches the C-128 to C-64 mode. In direct mode, the user will be prompted with an "ARE YOU SURE (Y/N)?" message; in program

mode, no prompts are given. The GO64 routine is a part of the GO routine, since the possibility of a GO_TO statement (instead of a GO64) must be checked.

Format:

GO64

GOSUB \$8D \$59CF

The GOSUB statement branches program execution to a specified line and allows a return to the statement following GOSUB. The specified line is a subroutine entry point.

Format:

GOSUB *ln*

The GOSUB statement calls a subroutine. The subroutine's entry point must occur on line *ln*. A subroutine's entry point is the beginning of the subroutine in a programming sense; that is to say, it is the line containing the statement (or statements) that are executed first. The entry point need not necessarily be the subroutine line with the smallest line number.

Upon completing execution the subroutine branches back to the line following the GOSUB statement. The subroutine uses a RETURN statement in order to branch back in this fashion.

A GOSUB statement may occur anywhere in a program; in consequence a subroutine may be called from anywhere in the program.

Example:

| | |
|----------------|--|
| 100 GOSUB 2000 | <i>Branch to subroutine at line 2000</i> |
| 110 A=B*C | <i>Subroutine branches back here</i> |
| . | |
| . | |
| 2000 | <i>Subroutine entry point</i> |

2090 RETURN

Branch back to line 110

GOTO **\$89** **\$59DB**
GO_TO **\$CB...\$A4** **\$5A3D**

The GOTO statement branches unconditionally to a specified line. It can also be expressed as GO__TO. The ROM code for the GO routine (at \$5A3D) tests for either a GO64 statement or a GO__TO statement.

*Format:*GOTO *ln**Example:*

10 GOTO 100

Executed in immediate mode, GOTO branches to the specified line in the stored program without clearing the current variable values. GOTO cannot reference immediate mode statements, since they do not have line numbers.

GRAPHIC **\$DE** **\$6B5A**

Selects one of the graphics modes of the C-128. The options are:

- 0 = 40 column text
- 1 = bit mapped graphics
- 2 = split screen graphics/text
- 3 = multicolor graphics
- 4 = multicolor split
- 5 = 80 column text

HEADER **\$F1** **\$A267**

The HEADER statement formats a diskette, assigning it a disk name and identification number.

Format:

HEADER "*diskname*",D*d*[,*lvv*][ON Uz]

When formatting a diskette, the HEADER statement marks off sectors on each track, then initializes the directory and block availability map. The formatted diskette must be in drive *d*. The diskette is given the name *diskname* and the number *vv*. This name and number appears in the reverse field at the top of a diskette directory display.

The HEADER statement is usually executed in immediate mode. The diskname can be contained within a string variable if the variable name is enclosed within parentheses when used with HEADER.

The HEADER statement can be used to format a blank diskette or to reformat and clear a used diskette. Because the changes are permanent, this command requires caution in its use. If executed in immediate mode, the question "ARE YOU SURE?" is displayed. You must respond by typing YES to continue.

If a media error occurs when the HEADER statement is executed, a "?BAD DISK" message is displayed on the screen. Media errors occur when a diskette is missing from the drive, the write/protect tab is in place, or the diskette magnetic surface is defective.

If no disk ID characters (*lvv*) are given, only the directory of the disk is cleared. This can only be done to an already formatted disk. It is faster than a full format.

Example:

```
HEADER "MASTER",D0,I02
```

Prepare and format a diskette, giving it the name "MASTER" and the number 02. The diskette is in drive 0

HELP**\$EA****\$5986**

This is a debugging tool. When HELP is activated, any error will be listed and displayed in reverse characters on the screen.

| | | |
|-------------|-------------|---------------|
| IF | \$8A | \$52C5 |
| THEN | \$A7 | \$52C5 |
| ELSE | \$D5 | \$5391 |

The IF/THEN/ELSE statement provides conditional execution of statements based on a relational expression.

Format:

If *condition* THEN *statement*[:*statement* . . .] Conditionally execute statement(s)

If *condition* { THEN } *line* *Conditionally branch*
 { GOTO }

If the specified condition is true, then the statement or statements following the THEN are executed. If the specified condition is false, control passes to the statement(s) on the next line and the statement or statements following the THEN are not executed. For a conditional branch, the branch line number is placed after the word THEN or after the word GOTO. The compound form THEN GOTO is also acceptable.

| | | |
|-----------------------|---|-------------------|
| IF A = 1 THEN 50 | } | <i>Equivalent</i> |
| IF A = 1 GOTO 50 | | |
| IF A = 1 THEN GOTO 50 | | |

If an unconditional branch is one of many statements following THEN, the branch must be the last statement on the line, and it must have "GOTO line" format. If the unconditional branch is not the last statement on the line, then statements following the unconditional branch can never be executed.

An ELSE statement is also provided by BASIC 7.0, and can follow an IF/THEN statement. Since it is a separate statement, it must be separated from the IF/THEN statement by a colon.

The following statements cannot appear in an immediate mode IF/THEN statement: DATA, GET, GET#, INPUT, INPUT#, REM, RETURN, END, STOP, WAIT.

If a line number is specified, or any statement containing a line number, there must be a corresponding statement with that line number in the current stored program.

The CONT and DATA statements cannot appear in a program mode IF/THEN statement. If a FOR/NEXT loop follows the THEN, the loop must be completely contained on the IF/THEN line. Additional IF/THEN statements may appear following the THEN as long as they are completely contained on the original IF/THEN line. However, Boolean connectors are preferred to nested IF/THEN statements. For example, the two statements below are equivalent, but the second is preferred. BEGIN and BEND can be used with the IF/THEN/ELSE for a more structured program. See their entries in this dictionary.

```
10 IF A$="X" THEN IF B=2 THEN IF C>D THEN 50
10 IF A$="X" AND B=2 AND C>D THEN 50
```

Example:

```
400 IF X>Y THEN A=1
500 IF M+1 THEN AG=4.5 GOSUB 1000
```

INPUT **\$85** **\$5662**

The INPUT statement receives data input from the keyboard.

Format:

```
INPUT { (blank) } var[,var,...,var]
      { "message" }
```

INPUT can be used in program mode only. When the INPUT statement is executed, C-128 BASIC displays a question mark on the screen requesting data input. The user must enter data items that agree exactly in number and type with the variables in the INPUT

statement parameter list. If the INPUT statement has more than one variable in its parameter list, then keyboard entries must be separated by commas. The last entry must be terminated with a carriage return.

| | |
|----------------------|------------------------------------|
| ?1234<CR> | <i>Single data item response</i> |
| ?1234,567.89,NOW<CR> | <i>Multiple data item response</i> |

If *message* is present, it is displayed before the question mark. A “*message*” can have as many as 80 characters.

If more than one but fewer than the required number of data items are input, C-128 BASIC requests additional input with double question marks (??) until the required number of data items have been input. If too many data items are input, the message “?EXTRA IGNORED” is displayed. The extra input is ignored, but the program continues execution.

Example:

| Statement | Operator Response | Result |
|------------------|---------------------------|-----------------------------|
| 10 INPUT A,B,C\$ | ? 123,456,NOW | A=123, B=456, C\$="NOW" |
| 10 INPUT A,B,C\$ | ? 123 ?? 456 ?? NOW | A=123 B=456 C\$="NOW" |
| 10 INPUT A,B,C\$ | ? NOW ?REDO FROM START | |
| | ? 123 ?? 456 ?? 789 | A=123 B=456 C="789" |
| 10 INPUT "A= ";A | A= ? 123 | A=123 |

Note that you must input numeric data for a numeric variable, but you can input numeric or string data for a string variable.

INPUT# \$84 \$5648

The INPUT external statement (INPUT#) inputs one or more data items from an external device identified via a logical file number.

Format:

INPUT#*lf* var[,var,...,var]

The INPUT# statement inputs data from the selected external device and assigns data items to variable(s) *var*. Data items must agree in number and kind with the INPUT# statement parameter list.

If an end-of-record is detected before all variables in the INPUT# statement parameter list have received data, then an “OUT OF DATA” error status is generated, but the program continues to execute.

INPUT# and INPUT statements execute identically, except that INPUT# receives its input from a logical file. Also, INPUT# does not display error messages; instead, it reports error statuses that the program must interrogate and respond to.

Input data strings may not be longer than 160 characters (159 characters plus a carriage return) because the input buffer has a maximum capacity of 80 characters. Commas and carriage returns are treated as item separators by the computer when processing the INPUT# statement; they are recognized, but are not passed on to the program as data. INPUT# is valid in program mode only.

The ROM code for the INPUT# routine shares most of its code with the normal INPUT routine.

Example:

| | |
|-------------------|--|
| 1000 INPUT#10,A | <i>Input the next data item from logical file 10. A numeric data item is expected; it is assigned to variable A.</i> |
| 946 INPUT#12,A\$ | <i>Input the next data item from logical file 12. A string data item is expected; it is assigned to variable A\$.</i> |
| 900 INPUT#5,B,C\$ | <i>Input the next two data items from logical file 5. The first data item is numeric; it is assigned to numeric variable B. The second data item is a string; it is assigned to string variable C\$.</i> |

KEY

\$F9

\$610A

The KEY statement is used to display the current settings for the eight function keys and to assign new output strings to them. KEY without

parameters simply lists the current settings. KEY followed by a number in the range 1 through 8 and a string results in the assignment of that string to the function key.

Format:

KEY

Lists current key values.

KEY n , "string"

Redefines function key.

Example:

```
KEY7, "OPEN 1,4,7: CMD1: LIST: CLOSE1" + CHR$(13)
```

Assigns a command string that LISTs the current BASIC program in memory everytime the F7 key is pressed. The CHR\$(13) character simulates the RETURN key being pressed. Double quotation marks cannot be directly included in the string; instead, the string must be broken up and have CHR\$(34) functions inserted.

LET **\$88** **\$53C6**

The assignment statement LET=, or simply =, assigns a value to a specified variable.

Format:

$$\left. \begin{array}{l} \text{\textit{(blank)}} \\ \text{LET} \end{array} \right\} \quad \textit{var} = \textit{data}$$

Variable *var* is assigned the value computed by resolving *data*.

The word “LET” is optional; it is usually omitted, although an *implied* LET statement will still result in calling the ROM routine at \$53C6.

Example:

```
10 A=2
300 M(1,3)=SGN(X)
310 XX$(I,J,K,L)="STRINGALONG"
```

LIST **\$9D** **\$50E2**

LIST displays one or more lines of a program. Program lines displayed by the LIST statement may be edited.

Format:

$$\text{LIST} \left\{ \begin{array}{l} \text{(blank)} \\ \text{line} \\ \text{line}_1 - \text{line}_2 \\ - \text{line} \\ \text{line} - \end{array} \right.$$

The entire program is displayed in response to LIST. Use line-limiting parameters for long programs to display a section of the program that is short enough to fit on the screen.

Example:

| | |
|-------------|---|
| LIST | <i>List entire program</i> |
| LIST 50 | <i>List line 50</i> |
| LIST 60-100 | <i>List all lines in the program from lines 60 to 100, inclusive</i> |
| LIST -140 | <i>List all lines in the program from the beginning of the program through line 140</i> |
| LIST 20000- | <i>List all lines in the program from line 20000 to the end of the program</i> |

Listed lines are reformatted as follows:

1. ?'s entered as a shorthand for PRINT are expanded to the word PRINT. Example:

?A becomes PRINT A

2. Lines whose keywords have been entered using abbreviations (for example, **PO SHIFT K** for POKE) will be listed in their full readable form with no abbreviations.

3. Blanks preceding the line number are eliminated. Example:

| | | |
|-----------|----------------|-----------|
| 50 A=1 | | 50 A=1 |
| | <i>becomes</i> | |
| 100 A=A+1 | | 100 A=A+1 |

4. A space is inserted between the line number and the rest of the statement if none was entered. Example:

55A=B-2 becomes 55 A=B-2

LIST is always used in immediate mode. A LIST statement in a program will list the program but then exit to immediate mode. Attempting to continue program execution via CONT simply repeats the LIST indefinitely.

Printing a Program Listing To print a program listing instead of displaying it, OPEN a printer logical file and execute a CMD statement before executing the LIST statement. Here is the necessary immediate mode sequence:

| | |
|----------|---|
| OPEN 4,4 | <i>Open the printer specifying logical file 4</i> |
| CMD 4 | <i>Deflect display output to the printer</i> |
| LIST | <i>Print the program listing</i> |
| PRINT#4 | <i>Deflect output back to the display</i> |
| CLOSE 4 | |

LOAD \$93 \$912C

The LOAD statement loads a program from an external device into memory. If a string variable contains the filename, the variable's name must be enclosed within quotes when used with LOAD.

Cassette Program Format

LOAD ["filename"][,dev]

The LOAD statement loads into memory the program file specified by *filename* from the cassette unit selected by device number *dev*. If no device is specified, device 1 is assumed by default; cassette unit 1 is then selected. If no filename is given, the next file detected on the selected cassette unit is loaded into memory.

Example:

| | |
|--------------|---|
| LOAD | <i>Load into memory the next program found on cassette unit #1. If you start a LOAD when the cassette is in the middle of a program, the cassette will read past the remainder of the current program, then load the next program</i> |
| LOAD "",2 | <i>Load into memory the next program found on cassette unit #2</i> |
| LOAD "EGOR" | <i>Search for the program named EGOR on tape cassette #1 and load it into memory</i> |
| N\$="WHEELS" | <i>Search for the program named WHEELS on cassette unit #1 and load it into memory</i> |
| LOAD (N\$) | <i>Search for a program named X on cassette unit #1 and load it into memory</i> |
| LOAD "X" | |

Diskette Drive Program Format

LOAD "dr:file name",dev

The LOAD statement loads into computer memory the program file with the *filename* on the diskette in drive *dev*. The device number for the diskette drive unit is 8 in the C-128. If *dev* is not present, the

default value is 1, which selects the primary tape cassette unit.

A single asterisk can be included instead of the filename, in which case the first program found on the selected diskette drive is loaded into memory.

Example:

| | |
|--------------------|---|
| LOAD"0:*",8 | <i>Load the first program found on disk drive 0</i> |
| LOAD"0:FIREBALL",8 | <i>Search for the program named FIREBALL</i> |
| | <i>on disk drive 0 and load it into memory</i> |
| T\$="0:METEOR" | <i>Search for the program named METEOR on</i> |
| | <i>disk drive 0 and load it into memory</i> |
| LOAD T\$,8 | |

When a LOAD is executed in immediate mode, C-128 BASIC automatically executes CLR before the program is loaded. Once a program has been loaded into memory, it can be listed, updated, or executed.

The LOAD statement can also be used in program mode to build program overlays. A LOAD statement executed from within a program causes that program's execution to stop and another program to be loaded. In this case the C-128 computer does not perform a CLR; therefore, the old program can pass on all of its variable values to the new program.

When a LOAD statement accessing a cassette unit is executed in program mode, LOAD message displays are suppressed unless the tape PLAY key is up (off). If the PLAY key is off, the "PRESS PLAY ON TAPE #1" message is displayed so that the load can proceed. All LOAD messages are suppressed when loading programs from a diskette in program mode.

LOCATE \$E6 \$6955

Positions the pixel cursor in the specified location on the graphics screen.

Format:

LOCATE X, Y

where X and Y are the new x and y coordinates of the pixel cursor. Either coordinate can be an absolute or relative screen location. Thus `LOCATE 20,50` moves the pixel cursor to the point (20,50) while `LOCATE +20, -50` moves the pixel cursor 20 pixels to the right and 50 units upward. Absolute and relative positions can be used within the same `LOCATE` statement.

MONITOR \$FA \$B000

Enters the machine language monitor.

Format:

MONITOR

MOVSPR \$FE \$06 \$6CC6

This command is used to either position a sprite on the screen or specify a direction and speed for sprite movement.

Format:

MOVSPR S , X , Y

where S is the sprite number and X and Y specify the location on the screen. If a + or - sign precedes both X and Y coordinates, relative positioning is used. A + indicates rightward or downward movement, while a - indicates leftward or upward movement.

MOVSPR X , A , # SP

where X is the sprite number, A is the angle of movement, and SP is the speed of the sprite (0-15).

NOTE: The # sign is necessary for the polar form of `MOVSPR`.

NEW \$A2 \$51D6

The NEW statement clears the current program from memory.

Format:

NEW

When a NEW statement is executed, all variables are initialized to zero or null values and array variable space in memory is released. The pointers that keep track of program statements are reinitialized, which has the effect of deleting any program in memory; in fact the program is not physically deleted. NEW operations are automatically performed when a LOAD statement is executed.

If there is a program in memory, you should execute a NEW statement in immediate mode before entering a new program at the keyboard. Otherwise, the new program will overlay the old one, replacing lines if their numbers are duplicated, but leaving other lines. The result is a scrambled mixture of two unrelated programs.

Example:

NEW

NEW is always executed in immediate mode. If a NEW statement is executed from within a program, the program will self-destruct, or clear itself out.

ON-GOSUB \$91...\$8D \$53A3

The ON-GOSUB statement provides conditional subroutine calls to one of several subroutines in a program, depending on the current value of a variable.

Format:

ON *byte* GOSUB *line*₁[,*line*₂,...,*line*_n]

ON-GOSUB has the same format as ON-GOTO. Refer to the ON-GOTO statement description for branching rules. The *byte* is evaluated and truncated to an integer number, if necessary.

For *byte*=1, the subroutine beginning at *line*₁ is called. That subroutine completes execution with a RETURN statement that causes program execution to continue at the statement immediately following ON-GOSUB. If *byte*=2, the subroutine beginning with *line*₂ is called, and so on.

ON-GOSUB is normally executed in program mode. It may be executed in immediate mode as long as there are corresponding line numbers to branch to in the current stored program.

The ROM code that handles ON-GOSUB also handles ON-GOTO statements.

Example:

```
10 ON A GOSUB 100,200,300
```

ON-GOTO \$91...\$89 \$53A3

The ON-GOTO statement causes a conditional branch to one of several points in a program, depending on the current value of a variable.

Format:

```
ON byte GOTO line1[,line2,...,linen]
```

Byte is evaluated and truncated to an integer number, if necessary. If *byte*=1, a branch to line number *line*₁ occurs. If *byte*=2, a branch to line number *line*₂ occurs, and so on.

If *byte*=0, no branch is taken. If *byte* is in the allowed range but there is no corresponding line number in the program, then no branch is taken. If a branch is not taken, program control proceeds to the statement following the ON-GOTO; this statement may be on the

same line as the ON-GOTO (separated by a colon) or on the next line.

If index has a nonzero value outside of the allowed range, the program aborts with an error message. As many line numbers may be specified as will fit on the 80-character line.

ON-GOTO is normally executed in program mode. It may be executed in immediate mode as long as there are corresponding line numbers in the current stored program that may be branched to.

ON-GOTO ROM code also handles ON-GOSUB statements.

Example:

```
40 A=B<10
50 ON A+2 GOTO
100,200
50 X=X+1
60 ON X GOTO
500,600,700
```

*Branch to statement 100 if A is true (-1) or
branch to statement 200 if A is false (0)*

*Branch to statement 500 if X=1, statement
600 if X=2, or to statement 700 if X=3. No
branch is taken if X>3*

OPEN \$9F \$918D

The OPEN statement opens a logical file and readies the assigned physical device.

Cassette Data File Format

OPEN *lf* [,*dev*] [,*sa*] [, "*filename*"]

The file named *filename* on the tape cassette unit identified by *dev* is opened for the type of access specified by the secondary address *sa*; the access is assigned the logical file number *lf*.

If no filename is specified, the next file encountered on the selected tape cassette is opened. If no device is specified, device number 1 is selected by default; this device number selects cassette unit 1. If no secondary address is specified, a default value of 0 is assumed and the file is opened for a read access only. A secondary

address of 1 opens the file for a write access, while a secondary address of 2 opens the file for a write access with an end-of-tape mark written when the file is subsequently closed.

Example:

| | |
|----------------------------|---|
| OPEN 1 | <i>Open logical file 1 at cassette drive #1 (default) for a read access (default) from the first file encountered on the tape (no file name specified)</i> |
| OPEN 1,1 | <i>Same as above</i> |
| OPEN 1,1,0 | <i>Same as above</i> |
| OPEN 1,1,0,"DAT" | <i>Same as above but access the file named DAT</i> |
| OPEN 3,1,2 | <i>Open logical file 3 for cassette #1 for a write with EOT (End of Tape) access. The new file is unnamed and will be written at the current physical tape location</i> |
| OPEN 3,1,2, "PENTAGRAM" | <i>Same as above but access the file named PENTAGRAM</i> |

Disk Data File Format

OPEN *lf,dev,sa,"dr:file name,type[,access]"*

The file named *filename* on the diskette in drive *dr* is opened and assigned logical file number *lf*. *type* identifies the file as sequential (SEQ), program (PRG), or random (USR). If the file is sequential, access must be WRITE to specify a write access or READ to specify a read access. Access is not present for a program or random access file.

An existing sequential file can be opened for a write access if *dr* is preceded by an @ sign. The existing sequential file contents are replaced entirely by new written data.

The device number *dev* must be present; it is 8 for all standard disk units. If *dev* is absent, a default value of 1 is assumed and the primary tape cassette unit is selected.

For a data file, the secondary address *sa* can have any value between 2 and 14, but every open data file should have its own unique secondary address. A secondary address of 15 selects the disk unit command channel. Secondary addresses of 0 and 1 are used to access

program files. Secondary address 0 is used to load a program file; secondary address 1 is used to save a program file.

Example:

| | |
|-----------------------|--|
| OPEN 1,8,2, | <i>Open logical file 1 on a diskette in drive 0.</i> |
| "0:DAT,SEQ,READ" | <i>Read from sequential file DAT.</i> |
| OPEN 5,8,3, | <i>Open logical file 5 on a diskette in drive 1.</i> |
| "1:NEWFILE,SEQ,WRITE" | <i>Write to sequential file NEWFILE.</i> |

PAINT \$DF \$61A8

Paints a selected area the specified color.

Format:

PAINT *C, X, Y, M*

where *C* is the color, *X* and *Y* specify any point inside the area to be painted, and *M* selects the area mode.

PLAY \$FE \$04 \$6DE1

Plays a note or series of notes.

Format:

PLAY "*V#, O#, E#, U#, X#, M*"

where *V#* is the voice number, *O#* is the octave, *E#* is the envelope number, *U#* is the volume, and *X* is the filter/switch. *M* is a string of notes (A,B,C...). All parameters are concatenated into one string.

POKE \$97 \$80E5

The POKE statement stores a byte of data in a specified memory location.

Format:

POKE *memadr,byte*

A value between 0 and 255, provided by *byte*, is loaded into the memory location with the address *memadr* in the current bank.

Example:

| | |
|------------------------|---|
| 10 POKE 1,A | <i>POKE value of variable A into memory at address 1</i> |
| POKE 32768,ASC("A")-64 | <i>POKE 1 (the value of ASC("A")-64) into memory at address 32768</i> |

PRINT **\$99** **\$555A**

The PRINT statement displays data; it is also used to print to the line printer.

Format:

```
PRINT
      ?      data      ' ; data ... ' ; data ...
```

PRINT Field Formats Numeric fields are displayed using standard numeric representation for numbers greater than 0.01 and less than or equal to 999999999. Scientific notation is used for numbers outside of this range. Numbers are preceded by a sign character and are followed by a blank character.

| | |
|-----------------------|-------|
| Sign | Blank |
| Number | |
| SNNN,...NNb | |
| Numeric field display | |

The sign is a blank for a positive number and a minus sign (−) for a negative number.

Strings are displayed without additions or modifications.

PRINT Formats *First data item.* The first data item is displayed at the current cursor position. The PRINT format character (comma or semicolon) following the first data item specifies the location of the second data item's display. The location of each subsequent data item's display is determined by the punctuation following the preceding data item. Data items may be in the same PRINT statement or in a separate PRINT statement.

New line. When no comma or semicolon follows the last data item in a PRINT statement, a carriage return occurs after the last data item is displayed.

Tabbing. A comma following a data item causes the next data item to be displayed at the next default tab column. Default tabs are at columns 1, 11, and 21. If a comma precedes the first data item, a tab will precede the first item display.

Continuous. A semicolon following a data item causes the next display to begin immediately in the next available column position. Numeric data always has one trailing blank character. For string data, items are displayed continuously with no forced intervening spaces.

Example:

```
40 PRINT A
40 PRINT A,B,C
40 PRINT A;B;C
40 PRINT, A;B;C
40 PRINT "NUMBERS",A;B;C
40 PRINT "NUM";"BER";
41 PRINT "S",A;B;C
```

PRINT#

\$98

\$553A

The print external statement (PRINT#) outputs one or more data items from the computer to an external device (cassette tape unit, disk unit, or printer) identified by a logical file number. Much of the ROM code used by an ordinary PRINT statement is also used by PRINT#.

Format:

```
PRINT#lf,data;CHR$(13);data;CHR$(13), . . . ,CHR$(13);data
```

Data items listed in the PRINT# statement parameter list are written to the external device identified by logical unit number *lf*.

Very specific punctuation rules must be observed when writing data to external devices. A brief summary of punctuation rules is given below.

PRINT# Output to Cassette Files Every numeric or string variable written to a cassette file must be followed by a carriage return character. This carriage return character is automatically output by a PRINT# statement that has a single data item in its parameter list. But a PRINT# statement with more than one data item in its parameter list must include characters that force carriage returns. For example, use CHR\$(13) to force a carriage return, or a string variable that has been equated to CHR\$(13) such as *c\$=CHR\$(13)*.

PRINT# Output to Diskette Files The cassette output rules described above apply also to diskette files, with one exception: groups of string variables can be separated by comma characters (CHR\$(44)). The comma character separators, like the carriage return separators, must be inserted using CHR\$. String variables written to diskette files with comma character separators must subsequently be read back by a single INPUT# statement. The INPUT# statement reads all text from one carriage return character to the next.

PRINT# Output to the Line Printer When the PRINT# statement outputs data to a line printer, CHR\$ must equal CHR\$(29). No punctuation characters should separate CHR\$ from data items, as illustrated in the PRINT# format definition.

Caution: The form ?# cannot be used as an abbreviation for PRINT#.

PRINT USING \$99 \$FB \$555A to \$9520
PRINT# USING \$98 \$FB \$553A to \$9520

The PRINT USING statement (and its corresponding file-oriented cousin, PRINT# USING) configure output of text strings. A format list that indicates how the output fields are configured is included after the USING keyword. Auxiliary characters such as decimal points, commas, dollar signs, plus signs, and minus signs can be appropriately positioned in the output field by including them within the format list.

Format:

PRINT(*#filename*) USING "*format list*"; *print list*

A typical format list's field specifier would be "\$##,###.##". The pound signs indicate locations in the output field that can contain a digit; if the number

1.0354679 E+04

were printed with this format list, the output field would appear as follows:

\$10,354.68

If the number 1234567.881 were to be printed with the above field specifier, the PRINT USING "overflow" indication

\$*****

would appear. Leading zeroes normally do not appear unless a fractional number appears; thus, .16 will appear as 0.16 if the format list "##.##" is used.

Dollar signs appear in a fixed position if they are the first character in the field specifier. If a pound sign precedes the dollar sign

in the field specifier, the dollar sign will “float” within the output field; the dollar sign’s position will vary with the number of digits (or string characters) output within that field.

Plus and minus signs can be included in the first or last position in a field; when a number is later printed using that format, the sign will be displayed according to the following conventions:

1. A sign can appear either at the beginning or ending position of an output field, but not in both positions.
2. If a minus sign is included in the format list and obeys the preceding convention in step 1, it will only appear if the number to be printed is negative.
3. If a plus sign is included in the format list and obeys the convention in step 1 above, the plus sign will appear if a positive number is to be output. When a negative number is to be output—despite the plus sign in the format list—a minus sign will appear in the selected print position.

Commas and decimal points can be included in a format list. Only one decimal point can be printed; if it is not included, the output value will be rounded to the nearest whole number and displayed without a decimal point. If commas are also included in a format list, they will be printed only between the number’s digits—leading commas (those to the left of the digits) are replaced by the filler character. This filler character is normally a space, but it can be changed through the use of BASIC 7.0’s PUDEF statement. Four up-arrow symbols indicate that a number is to be printed in scientific notation (that is, E format) and must also be accompanied by pound signs to specify a field width.

String text output with PRINT USING also uses pound signs to specify the field. Additionally, the presence of an equal sign in the format list will cause the output string to be centered within the field. The field width is then determined by both the number of pound signs

and equal signs (only one, normally) in the field specifier. If a string to be centered in an output is longer than the specified field width, the field will be filled with the leftmost characters of the string—truncating the string by dropping enough rightmost characters in order to make the string fit the field. A greater-than sign (>) is used to flag a string that is to be right-justified; the > character is also part of the field width specifier.

PUDEF \$DD \$5F34

The PUDEF statement lets the user redefine four special characters output by PRINT USING. Its format is

Format:

PUDEF "cccc"

The string following PUDEF should be four (or fewer) bytes long. Each character *c* is one of PRINT USING's special format characters.

The first character in the PUDEF string is PRINT USING's filler character and is normally a space. The second and third characters in this string are the comma and decimal point characters, respectively. They can be replaced with other characters, if desired. The final character of the PUDEF string is the dollar sign. Changing this character to a [symbol would result in PRINT USING displaying output in pounds, not dollars.

READ \$87 \$56A9

The READ statement assigns values from a DATA statement to variables named in the READ parameter list.

Format:

READ var[,var,...,var]

READ is used to assign values to variables. READ can take the place of multiple assignment statements (see LET=).

READ statements with variable lists require corresponding DATA statements with lists of constant values. The data constants and corresponding variables have to agree in type. A string variable can accept any type of constant; a numeric variable can accept only numeric constants.

The number of READ and DATA statements can differ, but there must be an available DATA constant for every READ statement variable. There can be more data items than READ statement variables, but if there are too few data items the program aborts with an “?OUT OF DATA” error message.

READ is generally executed in program mode. It can be executed in immediate mode as long as there are corresponding DATA constants in the current stored program to read from.

Example:

```
10 DATA 1,2,3
20 READ A,B,C
```

On completion, A=1,B=2,C=3

```
150 READ C$,D,F$
160 DATA STR
170 DATA 14.5,"TM"
```

On completion, C\$="STR",D=14.5,F\$="TM"

RECORD

\$FE \$12

\$A2D7

The RECORD statement is used to set up pointers in relative file I/O operations. The pointer can be directed to any character in any record of a relative file.

Format:

RECORD#*logical file number, record number, byte number*

The *logical file number* can range from 0 to 255, while the *record*

number can exist in the range 0-65535. The byte number, an index into the actual record, can range from 1 to 254.

If the record number is greater than the last record number in the file, the file will be expanded to the desired record number if a PRINT# operation takes place. However, if an INPUT# statement occurs instead—with the record pointer greater than the current number of records in the file—a “RECORD NOT PRESENT ERROR” is output by DOS. RECORD can use numeric variables for any of its parameters.

REM \$8F \$529D

The remark statement (REM) allows comments to be placed in the program for program documentation purposes.

Format:

REM *comment*

where *comment* is any sequence of characters that will fit on the current logical line.

REM statements are reproduced in program listings, but they are otherwise ignored. A REM statement may be placed on a line of its own, or it may be placed as the last statement on a multiple-statement line.

A REM statement cannot be placed ahead of any other statements on a multiple-statement line, since all text following the REM is treated as a comment. REM statements may be placed in the path of program execution, and they may be branched to. Even if BASIC statements or functions appear after the REM keyword, they are not tokenized.

Example:

```
10 REM *** * * * * ***
```

```
20 REM ***PROGRAM EXCALIBUR***  
30 GOTO 55: REM BRANCH IF OUT OF DATA
```

RENAME \$F5 \$A36E

RENAME merely changes the name of a disk file. (Note: if the file is currently open, RENAME will not change the filename.)

Format:

RENAME "old file" TO "new file" (,Ddrive#)(,Udevice#)

RENAME can accept string variables for the filenames if the variables are enclosed in parentheses. An example:

Example:

RENAME (QQ\$) TO (ZZ\$)

RENUMBER \$F8 \$5AF8

Renumbers a BASIC program according to the following parameters:

RENUMBER *N, I, O*

where *N* is the new starting line number, *I* is the increment between the lines, and *O* is the old starting line number. The RENUMBER command without parameters will renumber a BASIC program in increments of 10; the first line will become line 10.

RESTORE \$8C \$5ACA

The RESTORE statement resets the DATA statement pointer to the beginning of data.

Format:

RESTORE [line#]

RESTORE may be given in immediate or program mode. It resets the DATA pointer to the first DATA item in the program if no parameter follows the RESTORE statement. If a parameter does follow, the DATA pointer will be reset to the first DATA item in the line number parameter.

Example:

```

10 DATA 1,2,N44
20 READ A,B,B$      A=1,B=2,B$="N44"
30 RESTORE
40 READ X,Y,Z$      X=1,Y=2,Z$="N44"

```

RESUME \$D6 \$5F62

The RESUME statement returns control back to a given line of a BASIC 7.0 program after an error has been TRAPped. It only functions when error trapping has been engaged by the TRAP statement, since errors normally terminate program execution if unTRAPped. RESUME only functions in program mode; it can three different formats:

- RESUME attempts to reexecute the line in which the error occurred. If the error occurred in the middle or end of a multiple statement program line, RESUME will reexecute the complete program line.
- RESUME NEXT returns program control to the program line after the one in which the error occurred. Thus, if line 70 immediately follows line 60, and an error has been TRAPped in line 60, RESUME next will return control to line 70.
- RESUME *nnnnn* returns program control, after an error has been TRAPped, to line number *nnnnn*.

RETURN \$8E \$5262

The RETURN statement branches program control to the statement

in the program following the most recent GOSUB call. Each subroutine must terminate with a RETURN statement.

Format:

RETURN

Example:

100 RETURN

Note that the RETURN statement returns program control from a subroutine, whereas the RETURN key moves the cursor to the beginning of the next display line. The two are not related in any way.

RREG \$FE \$09 \$58BD

The RREG statement is used primarily after a SYS call to a machine language routine.

Format:

RREG *n1* (*,n1*)(*,n3*)(*,n4*)

RREG returns the contents of the 8502's .A, .X, .Y, and .S registers (respectively) to up to four BASIC variables after a SYS call. Only the first parameter is necessary; the additional three can be used as needed.

RUN \$8A \$5A9B

RUN begins execution of the program currently stored in memory. RUN closes any open files and initializes all variables to 0 or null values.

Format:

RUN[*line*]

or

RUN "*filename*"[,*Ddrive#*][,*Udevice#*]

When **RUN** is executed in immediate mode, the computer performs a CLR of all program variables and resets the data pointer in memory to the beginning of data (see **RESTORE**) before executing the program.

If **RUN** specifies a line number, the computer still performs the CLR and **RESTOREs** the data, but execution begins at the specified line number. **RUN** specifying a line number should not be used following a program break—use **CONT** or **GOTO** for that purpose.

RUN may also be used in program mode. It restarts program execution from the beginning of the program with all variables cleared and data pointers reinitialized.

Example:

| | |
|-----------------|--|
| RUN | <i>Initialize and begin execution of the current program</i> |
| RUN 1000 | <i>Initialize and begin execution of the program starting at line 1000</i> |

SAVE **\$94** **\$9112**

The **SAVE** statement writes a copy of the current program from memory to an external device. A string variable can contain a filename if the variable name is in parentheses.

Cassette Unit Format **SAVE**["*filename*"][,*dev*][,*sa*]

The **SAVE** statement writes the program that is currently in memory to the tape cassette drive specified by *dev*. If the *dev* parameter is not present, the assumed value is 1 and the primary cassette drive is selected. The *filename*, if specified, is written at the beginning of the program. If a nonzero secondary address (*sa*) is specified, an end-of-file mark is written on the cassette after the saved program.

Although no SAVE statement parameters are required when writing to a cassette drive, it is a good idea to name all programs. A named program can be read off cassette tape either by its name or by its location on the cassette tape. A program with no name can be read off cassette tape by its location only.

The SAVE statement is most frequently used in immediate mode, although it can be executed from within a program.

Example:

| | |
|-----------------------|--|
| SAVE | <i>Write the current program onto the cassette in drive 1, leaving it unnamed</i> |
| SAVE "RED" | <i>Write the current program onto the cassette in drive 1, assigning the file name of RED</i> |
| A\$="RED" | <i>Same as above</i> |
| SAVE (A\$) | |
| SAVE "BLACKJACK", 2,1 | <i>Write the current program onto the cassette in drive 2, naming the program BLACKJACK. Write an end-of-file mark after the program</i> |

Diskette Drive Format SAVE "*dr;filename*",*dev*

The SAVE statement writes a copy of the current program from memory to the diskette in the drive specified by *dr*. The program is given the name *filename*. The *dev* must be present; normally, it has the value 8. If *dev* is absent, a default value of 1 is assumed and the cassette is selected.

The filename assigned to the program must be new. If a file with the same name already exists on the diskette, a syntax error is reported. However, a program file can be replaced; if an @ sign precedes *dr* in the SAVE statement text string, the program replaces the contents of a current file named *filename*.

The diskette SAVE statement is also used primarily in immediate mode although it can be executed out of a program.

SCALE

\$E9

\$6960

Changes the graphics screen coordinate from 320×200 to 32767×32767 (default SCALEed size).

Format:

SCALE N [, $Xmax$] [$Ymax$]

where N is either scaling on (1) or off (0). $Xmax$ and $Ymax$ are optional parameters that let the programmer change the bit-map scaling.

| | |
|--------------------------|-------------------------|
| Monochrome bit-map mode: | $320 \leq Xmax < 32767$ |
| | $200 \leq Ymax < 32767$ |
| Multicolor bit-map mode: | $160 \leq Xmax < 32767$ |
| | $200 \leq Ymax < 32767$ |

SCNCLR **\$E8** **\$6A79**

Clears the current text screen or (with parameters) other screens.

Format:

SCNCLR [*param*]

The options are

- 0 = 40-column text
- 1 = bit-mapped
- 2 = split screen
- 3 = multicolor bit mapped
- 4 = split screen multicolor
- 5 = 80-column text

SCRATCH **\$F2** **\$A2A1**

The SCRATCH statement erases a single file from a diskette.

Format:

SCRATCH [Dd], "*filename*" [ON Uz]

The file named *filename* on the diskette in drive d is deleted. If the d parameter is not present, drive 0 is assumed. The filename can

be contained in a string variable if the variable's name is in parentheses.

The SCRATCH statement is used in immediate mode and in program mode. In immediate mode the statement is used to perform general diskette housekeeping operations. When executed, the message "ARE YOU SURE?" is displayed. You must key the response YES or Y with a carriage return, or the file will not be scratched.

When the SCRATCH statement is executed out of a program, no prompt messages are displayed. Temporary data files are frequently created by a program to hold transient data that will not fit in available memory. Temporary data files should be scratched before the program completes execution; otherwise a "FILE EXISTS" syntax error will be generated when the program is run next.

Files must be closed before they are scratched. If you attempt to scratch an open file, the C-128 computer may perform complex, erroneous diskette operations.

If using DOS 2.0, it is a good idea to COLLECT the diskette in immediate mode before scratching any files.

Example:

| | |
|-------------------------|--|
| SCRATCH D0, "DUMMY1" | <i>Scratch file DUMMY1 on diskette drive 0</i> |
| SCRATCH "DUMMY1" | <i>Same as above</i> |
| SCRATCH D1,"FILE1" | <i>Scratch FILE1 on diskette drive 1</i> |

SLEEP \$FE \$0B \$6BD7

Delay a program for a specified length of time.

Format:

SLEEP S

S is the number of seconds to delay (0-65535).

SLOW \$FE \$26 \$77C4

Opposite of FAST. Switches the C-128 to 1 MHz mode. 40-column text and graphics can then be displayed by the VIC-II chip.

Format:

SLOW

SOUND \$DA \$71 EC

The SOUND command directs the C-128 to output sounds and gives the user control over the SID sound chip. The SOUND statement can accept up to eight different parameters, although only the first three are required.

Format:

SOUND *v#*, *freq*, *dur* (*,stepdir*)(*,minfreq*)(*,stepval*)(*,wv*)(*,p*)

The *V#* number parameter, ranging from 1 through 3, selects one of the desired SID chip voices. The frequency parameter can range from 0 to 65535, while the duration parameter can range from 0 to 32767. The sound's duration is measured in units of one-sixtieth of a second ("jiffies").

The auxiliary parameter *stepdir* indicates the direction in which SOUND can sweep through a range of frequencies. Zero causes an upward sweep, 1 causes a downward sweep, and 2 causes an up/down oscillation. The minimum sweep frequency value, *minfreq*, is in the same format as the *freq* parameter. The parameter *stepval* contains the frequency-increment value for swept sounds; it can range between 0 and 32767. The *wv* parameter controls the waveform and can range from 0 through 3. (Zero here selects a triangle wave, 1 a sawtooth wave, 2 a variable pulse waveform, and 3 represents noise.) The *p* parameter can contain a number from 0 to 4095, and it controls the pulse width if the *wv* parameter contains a 2.

SPRCOLOR \$FE \$08 \$71 90

Defines the two multicolor sprite colors.

Format:

SPRCOLOR *N*, *C*

where *N* is the multicolor number and *C* is the color desired, both ranging 1 through 16.

SPRDEF \$FE \$1D \$7372

Starts the sprite editor.

SPRITE \$FE \$07 \$6C4F

Sets sprite parameters.

Format:

SPRITE *N, O, F, P, X, Y, M*

where *N* is the sprite number, *O* turns the sprite on (1) or off (0), *F* is the foreground color, *P* is the sprite priority, *X* and *Y* are the expand functions, and *M* is multicolor (on/off). A sprite has a priority of 1 if it is to appear behind a screen object.

SPRSAV \$FE \$16 \$76EC

The SPRSAV statement transfers the bytes forming a sprite image into a BASIC string variable and vice versa. SPRSAV can also “copy” one sprite to another. When string bytes are to be transferred into sprite storage, only the first 63 bytes of the string are used, since each sprite uses only 63 bytes. SPRSAV can be used in these three different contexts:

SPRSAV 3, QQ\$ Transfers sprite 3's image into QQ\$

SPRSAV QQ\$, 3 Transfers first 63 bytes from QQ\$ into sprite 3.

SPRSAV 3,4 Copies sprite 3 into sprite 4.

SSHAPE \$E4 \$642B

GSHAPE \$E3 \$658D

The SSHAPE and GSHAPE statements allow rectangular areas of bit-mapped graphics screens to be saved (using SSHAPE) or loaded

(using GSHAPE) to or from a string variable. Because a string can store only 255 bytes in Commodore BASIC, the size of the areas to be saved or loaded is limited.

SSHAPE Format

SSHAPE *string variable*, *x1*, *y1* (*,x2*, *y2*)

The *x2* and *y2* parameters are optional, and default to the pixel cursor-coordinate values if not included. The *x1*, *y1* coordinates determine the upper-left corner of the region to be saved to the BASIC string variable.

GSHAPE Format

GSHAPE *string variable* (*,x,y*)(*,mode*)

GSHAPE transfers the string to the graphics screen; the *x,y* coordinate pair is the coordinate of the top-left corner and defaults to the pixel cursor coordinates if not included. The *mode* parameter determines how the shape is to be placed on the graphics image. It can hold a value from 0 through 4, accomplishing one of the following functions:

- 0 Place shape as is onto bit-mapped screen (default mode).
- 1 Invert shape; pixels in the foreground color change to the background color and vice versa.
- 2 Logical OR shape with area to be overlaid. (Adds new image to image already in that area of the bit map.)
- 3 Logical AND shape with area to be overlaid.
- 4 Logical XOR (exclusive OR) shape with area to be overlaid.

Relative coordinates can be used in both SSHAPE and GSHAPE

operations; the x,y coordinates do not have to be absolute locations on the screen, but can be relative to the previous value of the pixel cursor. To select relative pixel addressing, include a + or - symbol before the desired coordinates. Absolute and relative coordinate usage can be mixed within the same statement or even the same coordinate pair! A plus sign before an x or y value indicates a rightward or downward positioning, while a minus sign indicates a leftward or upward positioning.

STASH**\$FE \$1 F****\$AA1 F**

The STASH statement transfers the contents of C-128 memory to RAM in an expansion cartridge.

STASH #bytes, intsa, expb, expsa

The #bytes parameter indicates the number of bytes to be transferred to external RAM; it can range from 1 to 65536. The parameter intsa is the starting address of the region of C-128 RAM to be transferred. The expb is the bank number (0 to 3) of the desired 64K segment within the expansion RAM cartridge; the expsa parameter is the starting address of the region in expansion RAM that will be accepting the transferred data.

STOP**\$90****\$4BCB**

The STOP statement causes the program to stop execution and return control to C-128 BASIC. A break message is displayed on the screen.

Format:

STOP

Example:

655 STOP

Will cause the message BREAK IN 655 to be displayed

SWAP **\$FE \$23** **\$AA29**

The SWAP statement exchanges the contents of a region of C-128 RAM with a region of RAM in an expansion cartridge. It has the same syntax and parameters as do FETCH and STASH. See the description of STASH for a warning about data transfers that may overlay the \$D000 I/O block.

SYS **\$9E** **\$5885**

The SYS statement is slightly different from its C-64 counterpart.

Format:

SYS address (,a)(,x)(,y)(,s)

SYS calls a machine language routine at an address in the memory configuration selected by the BANK statement. The optional parameters load (respectively) the 8502's .A, .X, .Y, and .S registers before the routine is actually executed. The address range is 0-65535. Upon execution of the routine's RTS instruction, control passes back to the BASIC interpreter. Use the RREG statement to pass the contents of 8502 registers back to BASIC variables.

TEMPO **\$FE \$05** **\$6FD7**

This changes the speed of a PLAY command.

Format:

TEMPO *S*

where *S* is a value between 0 and 255. Default value is 8. The duration of a whole note is 19.22/*S*.

TRAP **\$D7** **\$5F4D**

The TRAP statement transfers control to a specified line in a BASIC program if an error is encountered, including if the RUN-STOP key is

pressed. (It does not, however, trap DOS error messages. These must be checked using the DS\$ and DS DOS error variables.) Anytime an error occurs, control is passed to the statement whose line number is a parameter of the TRAP statement. The system variable ER contains the error number, while the string function ERR\$(ER) returns the text of the error message corresponding to the error condition.

Format:

TRAP *line#*

It should be noted that an error occurring in the routine that TRAP has passed control to cannot be TRAPped itself and will terminate program execution. If the keyword TRAP appears without a line number parameter, error TRAPPING is disabled. The RESUME statement returns control from the TRAP-handler routine to the desired portion of the program.

TROFF **\$D8** **\$58B7**

The TROFF statement disables the trace mode previously engaged by the TRON statement. It requires no parameters.

TRON **\$D9** **\$58B4**

The TRON statement turns on BASIC 7.0's trace mode. Useful during program debugging, trace mode displays the line number of the BASIC program line to be executed. TRON requires no parameters.

VERIFY **\$95** **\$9129**

The VERIFY statement compares the current program in memory with the contents of a program file. The filename can be stored in a string variable, providing the variable's name is in parentheses.

Cassette Unit Format VERIFY["*filename*"][,*dev*]

The program currently in memory is compared with the program named *filename* on the cassette in the unit specified by *dev*.

If *dev* is not present, a default of 1 is assumed and cassette unit 1 is selected. If *filename* is not present, the next file on the cassette in the selected unit is verified.

You should always verify a program immediately after saving it. The VERIFY statement is almost always executed in immediate mode.

Example:

| | |
|---------------|--|
| VERIFY | <i>Verify the next program found on the tape</i> |
| VERIFY "CLIP" | <i>Search for the program named CLIP on</i> |
| | <i>cassette unit #1 and verify it</i> |
| A\$="CLIP" | <i>Same as above</i> |
| VERIFY (A\$) | |

Diskette Drive Format VERIFY "*dr:filename*",*dev*

The program currently stored in memory is compared with the program file named *filename* on the diskette in drive *dr*. The *dev* parameter must be present and, unless, otherwise specified, it must have the value 8. If the *dev* parameter is absent, a default value of 1 is assumed and the primary cassette drive is selected.

In order to verify the program most recently saved, use the following version of the VERIFY statement:

```
VERIFY "*",8
```

You should always verify programs as soon as you have saved them. The VERIFY statement is nearly always executed in immediate mode.

Example:

| | |
|--------------------|--|
| VERIFY "*",8 | <i>Verify the program just saved</i> |
| VERIFY "0:SHELL",8 | <i>Search for the program named SHELL on</i> |
| | <i>disk drive 0 and verify it</i> |
| C\$="0:SHELL" | <i>Same as above</i> |
| VERIFY (C\$) | |

VOL **\$DB** **\$71C5**

Controls the loudness of a SOUND command.

Format:

VOL *N*

where *N* is a value from 0 (quiet) to 15 (loud).

WAIT **\$92** **\$6C2D**

The WAIT statement halts program execution until a specified memory location acquires a specified value.

Format:

WAIT *memadr,mask[,xor]*

where *mask* is a one-byte mask value and *xor* is a one-byte mask value.

The WAIT statement executes as follows:

1. The contents of the addressed memory location are fetched.
2. The value obtained in step 1 is exclusive ORed with *xor*, if present. If *xor* is not specified, it defaults to 0. When *xor* is 0, this step has no effect.
3. The value obtained in step 2 is ANDed with the specified mask value.
4. If the result is 0, WAIT returns to step 1, remaining in a loop that halts program execution at the WAIT.
5. If the result is not 0, program execution continues with the statement following the WAIT statement.

The RUN-STOP key will not interrupt WAIT statement execution.

WIDTH **\$FE \$1 C** **\$71 B6**

Changes the width of the lines drawn in DRAW. The options are: 1 (single width) and 2 (double width).

Format:

WIDTH *n*

WINDOW **\$FE \$1 A** **\$72 CC**

Establishes a functional window inside the normal screen window.

Format:

WINDOW *TLC, TLR, BRC, BRR, C*

where *TLC* is the top-left column, *TLR* is the top-left row, *BRC* is the bottom-right column, and *BRR* is the bottom-right row. *C* determines whether the new window will be cleared when it is defined.

BASIC 7.0 Functions

The C-128 can define a great number of functional operations directly from BASIC. These functions include mathematical derivations, screen-formatting instructions, and string manipulators. They are listed in alphabetical order.

ABS **\$B6** **\$8C84**

ABS returns the absolute value of a number.

Format:

ABS(*data n*)

Example:

```
A*ABS(10)           Results in A=10
A=ABS(-10)         Results in A=10
PRINT ABS(X),
ABS(Y),ABS(Z)
```

ASC **SC6** **\$B677**

ASC returns the ASCII code number for a specified character.

Format:

ASC(*data*\$)

If the string is longer than one character, ASC returns the ASCII code for the first character in the string. The returned argument is a number and may be used in arithmetic operations. If the string has zero length (is a null string), ASC returns a zero.

Example:

```
?ASC("A")           Prints 65
X=ASC("S")
?X                   Prints the ASCII value of "S," which is 83
```

ATN **SC1** **\$94B3**

ATN returns the arctangent of the argument.

Format:

ATN(*data n*)

ATN returns the value in radians in the range ± 17 .

Example:

```
A=ATN(A)
?180π*ATN(A)
```

BUMP**\$CE \$03****\$837C**

Stores a numeric value that corresponds to the objects (if any) that have collided on the video display. BUMP has a single parameter which can be either 1 or 2.

BUMP(1) examines sprite to sprite collisions. The sprites are each assigned a bit in the BUMP register. Sprite 0 is bit 0, sprite 1 is bit 1, and so on. To determine which sprites have collided, AND the value in the BUMP register with the binary value of the bit position of the sprite. To determine whether sprite 2 was in a collision, AND the value in the BUMP register with 2^2 (4). For example, BUMP(2) examines sprite to screen character collisions. The register works the same as shown earlier.

Example:

```
IF BUMP(1) AND 4 = 4 THEN PRINT "BUMP SPRITE #2!"
IF BUMP(2) AND 4 = 4 THEN PRINT "BUMP SPRITE #2!"
```

CHR\$**\$C7****\$85BF**

CHR\$ returns the string representation of the specified ASCII code.

Format:

CHR\$(byte)

CHR\$ can be used to specify characters that cannot be represented in strings. These include a carriage return and the double quotation mark.

Example:

```
IF C$=CHR$(13)
GOTO 10
?CHR$(34):
"HOHOHO":CHR$(34)
```

Branch if C\$ is a carriage return (CHR\$(13))

*Print the eight characters "HOHOHO"
(where CHR\$(34) represents a double quotation mark)*

COS **\$BE** **\$9409**

COS returns the cosine of the argument.

Format:

COS(*data n*)

DEC **\$D1** **\$8076**

Convert a hexadecimal number into a decimal number. DEC requires a string argument.

Example:

```
A=DEC("FFFF")
PRINT A
65535
```

ERR\$ **\$D3** **\$80F6**

This is a function that returns the text that describes the last error that the C-128 encountered.

Example:

```
PRINT ERR$
STRING TOO LONG
```

EXP **\$BD** **\$9033**

EXP returns the value e^{arg} . The value of e used is 2.71828183.

Format:

EXP(*arg n*)

The *arg n* must have a value in the range ± -88.029691 . A

number larger than +88.029691 will result in an overflow error message. A number smaller than -88.029691 will yield a zero result.

Example:

| | |
|-------------------|---|
| ?EXP(0) | <i>Prints 1</i> |
| ?EXP(1) | <i>Prints 2.71828183</i> |
| EV=EXP(2) | <i>Results in EV=7.3890561</i> |
| EB=EXP(50.24) | <i>Results in EB=6.59105247E+21</i> |
| ?EXP(88.0296919) | <i>Largest allowable number, yields 1.70141183E+38</i> |
| ?EXP(-88.0296919) | <i>Smallest allowable number, yields 5.87747176E-39</i> |
| ?EXP(88.029692) | <i>Out of range, overflow error message</i> |
| ?EXP(-88.029692) | <i>Out of range, returns 0</i> |

FRE **\$B8** **\$8000**

FRE is a system function that collects all unused bytes of memory into one block (called “garbage collection”) and returns the number of free bytes.

Format:

FRE(*arg*)

The *arg* can be either 0 or 1. FRE(0) indicates the amount of bank 0 RAM free for BASIC 7.0 program storage. FRE(1) returns the number of free bytes of bank 1 (variable storage) RAM.

Example:

| | |
|---------|--|
| ?FRE(1) | <i>Institute garbage collection and print the number of free bytes</i> |
|---------|--|

HEX\$ **\$D2** **\$8142**

Converts a decimal number into a hexadecimal number; it accepts a decimal (base 10) argument.

Example:

```
A=HEX$(65535)
PRINT A
FFFF
```

INSTR **\$D4** **\$99C1**

String search function. Locates a small string within a larger one and returns the smaller string's position (in characters) within the larger one.

Format:

```
INSTR (string1, string2 [, starting position])
```

Example:

```
A$=INSTR ("THE QUICK BROWN FOX", "OWN")
PRINT A$
13
```

The optional *starting position* parameter determines where the search begins.

INT **\$B5** **\$8CFB**

INT returns the integer portion of a number, rounding to the next lower signed number.

Format:

```
INT(arg n)
```

For positive numbers, INT is equivalent to dropping the fractional portion of the number without rounding. For negative numbers, INT is equivalent to dropping the fractional portion of the number and subtracting 1. Note that INT does *not* convert a floating-point number (5 bytes) to integer type (2 bytes).

Example:

| | |
|-------------|------------------------|
| A=INT(1.5) | <i>Results in A=1</i> |
| A=INT(-1.5) | <i>Results in A=-2</i> |
| X=INT(-0.1) | <i>Results in X=-1</i> |

A caution here: since floating-point numbers are only close approximations of real numbers, an argument may not yield the exact INT function value you might expect. For instance, consider the number 3.89999999. The function INT(3.89999999) would yield 3, not 4 as would be expected.

```
?INT(3.89999999)
3
```

JOY**\$CF****\$8203**

A function that returns a numeric value that corresponds to the current position of a specified joystick. The numeric positions are

| | | |
|-------------|---------------|--------------|
| | 1=UP | |
| 8=LEFT/UP | | 2=UP/RIGHT |
| 7=LEFT | | 3=RIGHT |
| 6=LEFT/DOWN | | 4=DOWN/RIGHT |
| | 5=DOWN | |
| | 0=NO MOVEMENT | |

If the number returned by JOY is greater than 128, then the FIRE button is pressed. An argument of 1 causes joystick 1 to be read, while an argument of 2 causes joystick 2 to be read.

Example:

```
PRINT JOY(1)
7
```

indicates that joystick 1 is sending *left* signals.

```
PRINT JOY(1)
135
```

indicates that joystick 2 is sending *left* signals and the FIRE button is pressed ($135 - 128 = 7$).

LEFT\$ **\$C8** **\$85D6**

LEFT\$ returns the leftmost characters of a string.

Format:

LEFT\$(arg\$,byte)

The *byte* specifies the number of leftmost characters to be extracted from the *arg\$* character string.

Example:

```
?LEFT$("ARG",2)                      Prints AR
A$=LEFT$(B$,10)                      Prints leftmost ten characters of the string B$
```

LEN **\$C3** **\$8668**

LEN returns the length of the string argument.

Format:

LEN(arg\$)

LEN returns a number that is the count of characters in the specified string. If the argument is a null string, LEN returns a zero.

Example:

```
?LEN("ABCDEF")
N=LEN(C$+D$)
```

Displays 6
Displays the sum of characters in strings C\$ and D\$

LOG **\$BC** **\$B9CA**

LOG returns the natural logarithm, or log, to the base e . The value of e used is 2.71828183.

Format:

LOG($arg\ n$)

An “ILLEGAL QUANTITY ERROR” message is returned if the argument is zero or negative.

Example:

```
?LOG(1)           Prints 0
A=LOG(10)          Results in A=2.30258509
A=LOG(1E6)         Results in A=13.8155106
A=LOG(X)/LOG(10)  Calculates log to the base 10
```

MID\$ **\$CA** **\$B61C**

MID\$ returns any specified portion of a string.

Format:

MID\$($data$,byte_1[,byte_2]$)

Some number of characters from the middle of the string identified by $data$$ is returned. The two numeric parameters $byte_1$ and $byte_2$ determine the portion of the string that is returned. String characters are numbered from the left, with the leftmost character having position 1. The value of $byte_1$ determines the first character to be extracted from the string. Beginning with this character, $byte_2$ determines the number of characters to be extracted. If $byte_2$ is absent,

then all characters up to the end of the string are extracted.

An "ILLEGAL QUANTITY ERROR" message is printed if a parameter is out of range.

Example:

```
?MID$("ABCDE",2,1)      Prints B
?MID$("ABCDE",3,2)      Prints CD
?MID$("ABCDE",3)        Prints CDE
```

PEEK \$C2 \$80C5

PEEK returns the contents of the specified memory location in the current bank. PEEK is the counterpart of the POKE statement.

Format:

PEEK(*mem adr*)

Example:

```
?PEEK(1)                      Prints contents of memory location 1
A=PEEK(20000)
```

PEN \$CE \$04 \$82AE

The PEN function returns information on the position of the light pen and can have an argument of 1 through 5.

- 1 = X position of light pen (40-column)
- 2 = Y position of light pen (40-column)
- 3 = X position of light pen (80-column)
- 4 = Y position of light pen (80-column)
- 5 = Light-pen trigger value

Example:

```
PRINT PEN(1), PEN(2)
100,100
```

indicates that the light pen is at dot position 100,100.

POINTER \$CE \$0A \$82FA

A system variable that contains the actual memory address of a specified variable (in bank 1).

```
PRINT POINTER(A$)
12349
```

POS \$B9 \$84D0

POS returns the column position of the cursor.

Format:

POS(*data*)

The *data* is a dummy function; it is not used and therefore can have any value. POS returns the current cursor position. If no cursor is displayed, the current character position within a program line or string variable is returned. Character positions begin at 0 for the leftmost character and can range up to 159.

Example:

```
?POS(1)                            At the beginning of a line, returns 0
?"ABCABC";POS(1)                 With a previous POS value of 0, displays a
                                     POS value of 6
```

POT \$CE \$02 \$824D

The POT function returns the current value(s) of the paddle controllers (1-4). The POT parameters are:

```
POT(1) = Current position of paddle 1
POT(2) = Current position of paddle 2
POT(3) = Current position of paddle 3
POT(4) = Current position of paddle 4
```


The POT values range from 0-255. If the value in POT(*N*) is greater than 255, then the FIRE button is pressed.

Example:

```
PRINT POT(1)
7
```

indicates that paddle 1 is in location 1.

```
PRINT POT(1)
355
```

indicates that paddle 1 is in screen position 100 and the FIRE button is pressed ($355 - 255 = 100$).

RCLR \$CE \$CD \$B1 9B

The RCLR function returns the color(s) of the following:

- 0 = background (40-column)
- 1 = foreground (bit map)
- 2 = multicolor #1
- 3 = multicolor #2
- 4 = border (40-column)
- 5 = character color
- 6 = background (80-column)

Example:

```
PRINT RCLR(0)
```

This indicates that the 40-column background is at the default value (12).

RDOT \$DO \$9BOC

System variable that contains the current position of the pixel cursor.
The parameters are:

- 0 = X coordinate of pixel cursor
- 1 = Y coordinate of pixel cursor
- 2 = Color of pixel cursor

RGR \$CC \$8182

System variable that contains the numeric value of the current graphics mode. The options are

- 0 = 40-column text
- 1 = 40-column bit map
- 2 = split screen
- 3 = multicolor bit map
- 4 = split screen multicolor
- 5 = 80-column text

RIGHT\$ \$C9 \$860A

RIGHT\$ returns the rightmost characters in a string.

Format:

RIGHT\$(arg\$,byte)

The *byte* identifies the number of rightmost characters that are extracted from the string specified by *arg\$*.

Example:

```
RIGHT$(ARG,2)
MMS=RIGHT$
(X$+"#",5)
```

Displays RG
MM\$ is assigned the last four characters of
X\$, plus the character#

RND \$BB \$8434

RND generates random number sequences ranging between 0 and 1.

Format:

RND(*arg n*) *Return random number*
 RND(-*arg n*) *Store new seed number*

Example:

A=RND(-1) *Store a new seed based on the value -1*
 A=RND(1) *Fetch the next random number in sequence*

An argument of zero is treated as a special case; it does not store a new seed, nor does it return a random number. RND(0) uses the current system time value TI to introduce an additional random element into play.

A pseudo-random seed is stored by the following function:

RND(-TI) *Store pseudo-random seed*

RND(0) can be used to store a new seed that is more truly random by using the following function:

RND(-RND(0)) *Store random seed*

RSPCOLOR \$CE \$07 \$8361

A function that returns the numeric value of the sprite multicolor registers. The options are:

1 = multicolor 1
 2 = multicolor 2

Example:

PRINT RSPCOLOR(1)
 12

indicates that multicolor 2 is dark gray. The color values range from 1 to 16.

RSPPOS \$CE \$05 \$8397

A function that returns the numeric value corresponding to the selected sprite position and/or speed. The options are

- 0 = X position of sprite
- 1 = Y position of sprite
- 2 = Speed of sprite (0-15)

Example:

```
PRINT RSPPOS (1,2)
12
```

indicates that sprite #1 is moving at a speed of 12.

RSPRITE \$CE \$06 \$831E

A function that returns the sprite data (assigned in a SPRITE statement) of the selected sprite. The options are

- 0 = SPRITE ON/OFF
- 1 = SPRITE COLOR
- 2 = PRIORITY
- 3 = X Expanded
- 4 = Y Expanded
- 5 = Multicolor mode

RWINDOW \$CE \$09 \$8407

A function that returns the current window information. The options are

- 0 = # lines in current window
- 1 = # rows in current window
- 2 = 40/80-column mode; returns either 40 or 80

SGN **\$B4** **\$8C65**

SGN determines whether a number is positive, negative, or zero.

Format:

SGN(*arg n*)

The SGN function returns +1 if the number is positive or nonzero, 0 if the number is zero, or -1 if the number is negative.

Example:

```
?SGN(-6)                      Displays -1
?SGN(0)                        Displays 0
?SGN(44)                       Displays 1
IF A>C THEN
SA=SGN(X)
IF SGN(M)>=0 THEN
PRINT "POSITIVE
NUMBER"
```

SIN **\$BF** **\$9410**

SIN returns the sine of the argument.

Format:

SIN(*arg n*)

Example:

```
A=SIN(AG)
?SIN(45* $\pi$ /180)              Displays the sine of 45 degrees
```

SPC **\$A6** **\$55B9**

SPC moves the cursor right a specified number of positions.

Format:

SPC(*byte*)

The SPC function is used in PRINT statements to move the cursor some number of character positions to the right. Text over which the cursor passes is not modified.

The SPC function moves the cursor right from whatever column the cursor happens to be in when the SPC function is encountered. This is in contrast to a TAB function, which moves the cursor to some fixed column measured from the leftmost column of the display. (See TAB for examples.)

The ROM code for both the SPC and TAB functions are part of the ROM routines for the PRINT statement.

SQR**\$BA****\$8FB7**

SQR returns the square root of a positive number. A negative number returns an error message.

Format:

SQR(*arg n*)

Example:

A=SQR(4)
A=SQR(4,84)
?SQR(144E30)

Results in A=2
Results in A=2.2
Displays 1.2E+16

ST**(No token)****\$7978**

ST returns the current value of the I/O status. This status is set to certain values depending on the results of the last input/output operation. ST is a system variable, not a BASIC 7.0 function.

Format:

ST

ST values are as follows:

| <i>ST Bit Position</i> | <i>ST Numeric Value</i> | <i>Cassette Tape Read</i> | <i>Cassette Tape Verify and Load</i> |
|----------------------------|-----------------------------|-------------------------------|--|
| 0 | 1 | | |
| 1 | 2 | | |
| 2 | 4 | Short block | Short block |
| 3 | 8 | Long block | Long block |
| 4 | 16 | Unrecoverable read error | Any mismatch |
| 5 | 32 | Checksum error | Checksum error |
| 6 | 64 | End of file | |
| 7 | 128 | End of tape | End of tape |

Status should be checked after execution of any statement that accesses an external device.

Example:

```

10 IF ST<>0 GOTO          Branch on any error
500
50 IF ST=4 THEN
?"SHORT BLOCK"

```

STR\$ **\$C4** **\$B5AE**

STR\$ returns the string equivalent of a numeric argument.

Format:

STR\$(arg n)

STR\$ returns the character string equivalent of the number generated by resolving *arg n*.

TI, TI\$ (No tokens) \$7978

TI and TI\$ represent two system time variables.

Format:

| | |
|------|--|
| TI | Number of jiffies since current startup (1 jiffy=1/60 sec.) |
| TI\$ | Time of day string |

Example:

```
?TI
TI$="081000"
```

USR(x) \$D7 \$1218

The USR(X) function allows the user to pass a floating-point numerical argument to a user-written machine language routine. Since the USR(x) call passes the argument — and not the address, as SYS calls do — a separate USR vector must be set up. When the C-128 system is initialized, location \$1218 (4632 decimal) contains a \$4C byte, which is the 8502 opcode for the JMP instruction. Following it is a two-byte address in standard 6502 low-byte/high-byte order. This is the USR vector, and it normally points to a routine that prints an error message. The USR vector is thus located at \$1219/\$121A (4633 and 4634 decimal) and should be directed to user-written code if it is to be used properly. The user-written machine code should return a five-byte floating-point value to the floating-point accumulator that begins at location \$63.

VAL \$C5 \$804A

VAL returns the numeric equivalent of the string argument.

BASIC Math Functions And the Jump Table

Although floating-point numbers are regularly stored in the standard five-byte format in the variable table, it's usually more useful to manipulate them in zero-page memory. Commodore BASIC 7.0 has two main floating-point accumulators that reside in zero-page RAM, along with another two temporary storage areas for floating-point numbers also in zero-page.

The two main accumulators are known as FAC and ARG. Occasionally they are also referred to as FAC1 and FAC2. Almost every BASIC math operation returns a result in FAC. Functions like LOG, SQR, ABS, or INT receive their input arguments in FAC and return results there. Denary operations, such as addition or multiplication, put one input value in FAC and one in ARG; FAC will return the result.

Table 4-5. Structure of FAC and ARG Floating-Point Accumulators

| <i>FAC</i> | | <i>ARG</i> | |
|------------|--------|------------|--|
| \$63 | FACEXP | \$6A | ARGEXP—standard floating-point exponent byte. |
| \$64 | FACHO | \$6B | ARGHO—mantissa byte 1 (with bit #7 set). |
| \$65 | FACMOH | \$6C | ARGMOH—mantissa byte 2. |
| \$66 | FACMO | \$6D | ARGMO—mantissa byte 3. |
| \$67 | FACLO | \$6E | ARGLO—mantissa byte 4. |
| \$68 | FACSGN | \$6F | ARGSGN—mantissa sign; \$00 = positive, \$FF = negative. \$6F, ARGSGN is used to indicate whether or not FAC and ARG have the same sign. A \$00 byte here indicates FAC and ARG have like signs, while \$FF indicates unlike signs. |

Both FAC and ARG use a slightly different data format than the standard five-byte floating-point number discussed previously. Bit #7 of the first mantissa bit normally reflects the mantissa sign in an ordinary BASIC floating-point variable. When the number is stored in FAC or ARG, however, this bit is always set to 1, ignoring the sign. Separate memory locations, FACSGN at \$68 and ARGSGN at \$6F, are used to record the sign of these accumulators; a zero in either FACSGN or ARGSGN indicates that the respective accumulator is positive, while a value of \$FF indicates the accumulator's contents are negative.

Routines that transfer five-byte floating-point numbers to or from FAC or ARG take this slight but important difference into account. Table 4-5 shows the format of both FAC and ARG.

Two other temporary floating-point storage areas are available in zero-page RAM. TEMPF1 (sometimes called FAC3) and TEMPF2 (sometimes called FAC4) are used to store intermediate results and can be used by programmers. Unlike FAC and ARG, these locations are only five bytes long, and bit #7 of their first mantissa bytes reflect the mantissa sign. These two temporary storage areas are presented in Table 4-6.

Two ROM routines move data from FAC to these temporary registers. The MOV2F routine, at \$8BF9, transfers the contents of the FAC to TEMPF2, while the MOV1F routine at \$8BFC transfers the

Table 4-6. Temporary Floating-Point Storage Areas

| <i>TEMPF1 (FAC3)</i> | <i>TEMPF2 (FAC4)</i> |
|----------------------|---|
| \$59 | \$5E — floating-point exponent byte. |
| \$5A | \$5F — mantissa byte #1, with implied sign. |
| \$5B | \$60 — mantissa byte #2. |
| \$5C | \$61 — mantissa byte #3. |
| \$5D | \$62 — mantissa byte #4. |

contents of FAC to TEMP1. No special setup is required before these routines are called.

The BASIC 7.0 jump table shown in Table 4-7 contains JMP entries to most of the BASIC numeric processing routines. Many of these math operations need only a number in FAC and ARG. Math operations that use a five-byte floating-point number somewhere in memory require that a pointer to the number be set up in the 8502 .A (low-byte of address) and .Y (high-byte of address). Any references to “Memory” in the math-operations section of the jump table therefore refer to a five-byte floating-point number pointed to by the 8502’s .A and .Y register (.A = low, .Y = high).

Table 4-7. BASIC 7.0 Jump Table

| <i>Hex Address</i> | <i>Code Address</i> | <i>Description</i> |
|--------------------|---------------------|--|
| AF00 | JMP \$84B4 | Convert floating-point number to integer (integer returned in FAC mantissa bytes 1 and 2). |
| AF03 | JMP \$793C | Convert integer to floating-point number (.A = integer low byte, Y = integer high byte). |
| AF06 | JMP \$8E42 | Convert floating-point number to ASCII string (string stored in FBUFFER, at \$0100). |
| AF09 | JMP \$8052 | Convert ASCII digit string to number in FAC (calls string handler LEN1 at \$866E). |
| AF0C | JMP \$8815 | Convert floating-point number to an address (0. .65535 range instead of -32768. .32767; value returned to POKER at \$16/\$17). |
| AF0F | JMP \$8C75 | Convert address to floating-point number. |
| AF12 | JMP \$882E | FAC = Memory - FAC. |

Table 4-7. BASIC 7.0 Jump Table (*continued*)

| <i>Hex Address</i> | <i>Code Address</i> | <i>Description</i> |
|--------------------|---------------------|--|
| AF15 | JMP \$8831 | FAC = ARG - FAC. |
| AF18 | JMP \$8845 | FAC = Memory + FAC. |
| AF1B | JMP \$8848 | FAC = ARG + FAC. |
| AF1E | JMP \$8A24 | FAC = Memory * FAC. |
| AF21 | JMP \$8A27 | FAC = ARG * FAC. |
| AF24 | JMP \$8B49 | FAC = Memory / FAC. |
| AF27 | JMP \$8B4C | FAC = ARG / FAC. |
| AF2A | JMP \$89CA | Calculate natural log of FAC. |
| AF2D | JMP \$8CFB | Return integer portion of FAC. |
| AF30 | JMP \$8Fb7 | Calculate square root of FAC. |
| AF33 | JMP \$8FFA | Negate FAC. |
| AF36 | JMP \$8FBE | Raise ARG to power contained in memory. |
| AF39 | JMP \$8FC1 | Raise ARG to power in FAC. |
| AF3C | JMP \$9033 | Calc EXP of floating-point accumulator. |
| AF3F | JMP \$9409 | Calculate COS of FAC. |
| AF42 | JMP \$9410 | Calculate SIN of FAC. |
| AF45 | JMP \$9459 | Calculate TAN of FAC. |
| AF48 | JMP \$94B3 | Calculate ATN of FAC. |
| AF4B | JMP \$8C47 | Round FAC. |
| AF4E | JMP \$8C84 | Find absolute value of FAC. |
| AF51 | JMP \$8C57 | Test sign of FAC. |
| AF54 | JMP \$8C87 | Compare FAC with floating-point number in memory. |
| AF57 | JMP \$8437 | Generate random floating-point number. |
| AF5A | JMP \$8AB4 | Move floating-point number to ARG. Depends on current memory configuration. |
| AF5D | JMP \$8489 | Move floating-point number in ROM to ARG. |
| AF60 | JMP \$7A85 | Move floating-point number in Bank 1 RAM to FAC. |

Table 4-7. BASIC 7.0 Jump Table (*continued*)

| <i>Hex Address</i> | <i>Code Address</i> | <i>Description</i> |
|--------------------|---------------------|---|
| AF63 | JMP \$8BD4 | Move floating-point number in ROM to FAC. |
| AF66 | JMP \$8C00 | Move number in FAC to memory. |
| AF69 | JMP \$8C28 | Move ARG to FAC. |
| AF6C | JMP \$8C38 | Move FAC to ARG. |
| AF6F | JMP \$4828 | Table of math operators at \$4828. |
| AF72 | JMP \$9B30 | Draw a line. |
| AF75 | JMP \$9BFB | Plot a point. XPOS and YPOS at \$1131-\$1134 contain X,Y coordinates. |
| AF78 | JMP \$6750 | Find next point on a circle. |
| AF7B | JMP \$5A9B | RUN a program. Accepts parameters, can load from disk, or begin at specific line number. |
| AF7E | JMP \$51F3 | Reset TXTPTR to start of BASIC, and CLR. |
| AF81 | JMP \$51F8 | CLEAR resets variable and array space, closes all files, and resets stack. |
| AF84 | JMP \$51D6 | NEW clears first BASIC line-link byte and sets TEXT-TOP equal to TXTTAB+2. Falls through to CLEAR routine. |
| AF87 | JMP \$4F4F | Relink lines of BASIC program text. |
| AF8A | JMP \$430A | "Crunch" (tokenize) ASCII input. |
| AF8D | JMP \$5064 | FNDLIN; searches program for line number contained in \$16/\$17. If .C set on return, LOWTR at \$61/\$62 contains address of link bytes in sought line. A return with .C clear means search unsuccessful. |
| AF90 | JMP \$4AF6 | NEWSTT; checks for RUN-STOP key-stroke and saves TXTPTR for possible use by CONT command. |
| AF93 | JMP \$78D7 | EVAL, vectored through \$030A. Evaluates single BASIC token. |

Table 4-7. BASIC 7.0 Jump Table (*continued*)

| <i>Hex Address</i> | <i>Code Address</i> | <i>Description</i> |
|--------------------|---------------------|---|
| AF96 | JMP \$77EF | FRMEVL routine, evaluates tokenized BASIC expression pointed to by TXTPTR (\$3D/\$3E). |
| AF99 | JMP \$5AA6 | RUN-A-PROGRM; loads program from disk, links lines, and starts program. |
| AF9C | JMP \$5A81 | SETEXC; sets program execution mode (as against direct mode), enables sprite-collision trap code, and turns off Kernal messages. |
| AF9F | JMP \$50A0 | LINGET; gets line number using TXTPTR. Stores in LINNUM at \$16/\$17. |
| AFA2 | JMP \$92EA | Call garbage collection routine to collect discarded/temporary strings. |
| AFA5 | JMP \$4DCD | Execute a BASIC program line. This routine calls CRUNCH to tokenize it. .X and .Y contain address of line, which is later loaded into TXTPTR (\$3D/\$3E). |

To understand the proper use of these floating point routines, an example program is shown in Figure 4-8. It is an extremely fast square root routine employing Newton's method instead of the formula $\text{EXP}(\text{LOG}(x)/2)$ that Commodore has used for its ROM-resident SQR routine.

Given a continuous function $f(x) = 0$, with a derivative $f'(x)$, Newton's Method finds successive approximations for the value of x iteratively by using this formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

where x_n is a trial approximation and $x_{(n+1)}$ is a closer approximation. A square root of a number (A) is the solution to the equation $x^2 - A = 0$. Since the first derivative of x^2 is $2*x$, the Newton's method solution for a square root is:

$$x_{n+1} = (x_n - \frac{A}{x_n}) / 2$$

```

$0B00    JSR $AF51    ; Call jump table entry to check sign of FAC1.
          BEQ $0B2D    ; If zero flag set, FAC = 0 so exit.
          BPL $0B0C    ; If FAC > 0, then continue processing.
          LDX #$0E    ; else exit with error #14 - "OVERFLOW ERROR"
          JMP ($0300)  ; by JMPing through the IERROR vector.
$0B0C    JSR $8BFC    ; Transfer FAC1 to "temporary FAC1"
          ; (TEMPF1, at $59 - $5D)
          JSR $0B2E    ; Call code to divide FAC1's exponent byte
          ; by 2, getting a first approximation of
          ; the square root.
          LDA #$04    ;
          STA $FA    ; Counter variable - four times through loop.
$0B16    JSR $8BF9    ; Transfer FAC1 to "temporary FAC2"
          ; (TEMPF2, at $5E - $62)
          LDY #$00    ;
          LDA #$59    ; Point to TEMPF1 (starts at $59).
          JSR $AF24    ; Call FDIV to divide TEMPF1 by FAC1.
          LDY #$00    ;
          LDA #$5E    ; Point to TEMPF2 (starts at $5E).
          JSR $AF18    ; Call FADD to add TEMPF2 to FAC1.
          DEC $63    ; Divide result in FAC by two by decrementing
          ; FAC1's exponent byte.
          DEC $FA    ; Decrement loop counter.
          BNE $0B16   ; Not done? Then loop again.
$0B2D    RTS        ; Return to BASIC with result left in FAC1.
          ;
          ; Routine to divide FAC1's exponent byte by 2.
          ;
$0B2E    LDA $63    ; Get exponent byte (FACEXP) of FAC1.
          SEC        ; .C set to prepare for subtraction.
          SBC #$81   ; Subtract 129 offset from FACEXP.
          PHP        ; Push borrow status on stack.
          LSR        ; Divide by 2 by shifting right low 6
          ; bits of FACEXP.
          CLC        ; Clear carry to prepare for add - recovering
          ; FACEXP's offset.
          ADC #$01   ;
          PLP        ; Get old borrow status.
          BCC $0B3D  ; and determine if another add necessary.
          ADC #$7F   ;
$0B3D    STA $63    ; Store to FACEXP and return.
$0B3F    RTS

```

Figure 4-8. Fast square root routine employing Newton's method

By contrast, BASIC 7.0 calculates square roots using the formula $\text{SQR}(X) = \text{EXP}(\text{LOG}(x)/2)$. This technique requires the use of two time-consuming transcendental functions, while Newton's method requires only four loops to find a square root accurate to ten decimal places. The routine shown here executes in approximately one-quarter of the time that its BASIC 7.0 counterpart uses! The routine listed in Figure 4-8 resides in the cassette buffer at \$0B00 and is called via the `USR(x)` function. Thus, the `USR` vector must be set up to point to the start of the cassette buffer. This can be accomplished by entering **POKE 4633,0: POKE 4634, 11**.

`SIN`, `COS`, and `ATN` functions require calculations involving polynomial (Taylor) series; these series-evaluation routines are contained within the C-128 BASIC ROM. They can save the user much programming effort and time when any custom function expressible with a Taylor series is being created, since the only major task necessary is the creation of a table of floating-point numbers that will be used as coefficients of the polynomial.

The routine `POLYX`, at \$9086 in the BASIC ROM, evaluates polynomials that have terms containing odd powers of x (including a zero-power, or constant, term). The `POLY` routine at \$909C, however, evaluates polynomials that have terms containing *both* odd and even powers of x . Before calling either routine, the `.A` and `.Y` registers are loaded with the low- and high-address bytes of a table of floating-point coefficients. The first byte of the coefficient table contains a number that is one less than the degree (highest power of x) of the desired polynomial, while the remaining bytes are standard five-byte floating-point constants. These constants are arranged in order of degree; the coefficient of x^0 (the constant term) comes first, followed by the coefficients of x , x^2 , x^3 , and so on.

BASIC String Handling From Machine Language

BASIC 7.0 has a very complex internal string-handling mechanism that functions quickly enough for most applications. Sometimes,

however, a machine-code routine needs access to a string created by BASIC. Of course, the BASIC program could POKE the characters of the string into a region such as the cassette buffer, but this is fairly inefficient.

A routine involved in the execution of BASIC's LEN function can provide you not only with a string's length, but also with its start address in bank 1 RAM. The routine is referred to as LEN1, and it returns with .A containing the string's length. The zero-page utility pointer, INDEX1 at \$24/\$25, will contain the string's start address.

Calling the LEN1 routine (at \$8663 in the BASIC ROM) from a program is slightly more difficult than using the arithmetic routines. LEN1 exits with bank 1 RAM selected, so a routine residing in bank 0 that calls LEN1 directly will crash. Instead, the Kernal's JSRFAR routine has to be used.

```

$1800      LDX $3D      ; Preserve TXTPTR in free zero-page RAM.
          LDY $3E      ;
          STX $FA      ;
          STY $FB      ;

          LDY #$86      ; point to ROM's string-handler routine at $866E.
          LDX #$6E      ;
          JSR $1853     ; set up and execute a "long jump" (JSRFAR) to
                    ; $866E because a direct call exits with RAM
                    ; Bank #1 selected.
          STA $FC       ; String length in .A; address is in $24/$25.
          BNE $1816     ; If len(string) not zero, continue.
          STA $63       ; else, store zero in FACEXP, leaving FAC1 = 0.
          RTS          ; and return to BASIC.

$1816      LDY #$00     ; Zero counter.
$1818      LDA #$24     ; Address of zero-page pointer containing
                    ; string's starting address
                    ; in Bank 1.
          LDX #$01     ;
          JSR $1848     ; Call routine that saves this data prior
                    ; to call to INDFET.
          LDY #$FF     ; Point to $FF74.
          LDX #$74     ;
          JSR $1853     ; Call $FF74 (INDFET) using JSRFAR. String
                    ; character returned in .A.
          STA $0200,Y   ; Store string character to BASIC text input
                    ; buffer.
                    ; Bump counter.
          INY          ;
          CPY $FC       ; Compare counter value with string length.
          BCC $1818     ; Not done yet? Loop again,
                    ; else terminate string in input buffer
          LDA $800      ; with $00 end-of-string byte.
          STA $0200,Y   ;
          JSR $1869     ; Point TXTPTR ($3D/$3E) to $0200.
          JSR $AF8A     ; Call CRUNCH routine to tokenize string in
                    ; input buffer.

```

Figure 4-9. Extended, expression-evaluation VAL function, located in free RAM at \$1800, called by USR(x) function (set up USR vector with POKE 4633,0: POKE 4634,24 before using)

```

JSR $1869 ; Reset TXTPTR to $0200, preparing for evaluation.
JSR $AF96 ; Call FRMEVL to evaluate expression. Result
           ; returned in FAC#1.
LDX $FA ; Restore TXTPTR to its original upon USR(x)s call.
LDY $FB
STX $3D
STY $3E
RTS ; Return to BASIC - result of evaluation in FAC#1.
-----
; Routine to save 8502 registers in reserved
; zero-page RAM before "cross-bank" call to
; JSRFAR routine is executed.
$1848 STA $06 ; Save .A, .X, and .Y register contents.
      STX $07
      STY $08
      PHP ; Push status on stack and
      PLA ; pull it into .A.
      STA $05 ; Save status, too.
      RTS ; Return to caller.
-----
; Set-up and call for JSRFAR routine. Calling
; address in .Y (high) and .X (low). Calls
; ROM routines in Bank #15. Routine ends
; with 8502 registers containing data
; returned by called routine.
$1853 LDA #$0F ; Select Bank #15 (RAM0, system ROMs, I/O bank.)
      STA $02 ; Save Bank number for JSRFAR call.
      STY $03 ; " high-byte address of routine to be called.
      STX $04 ; " low-byte address of routine to be called.
      JSR $02CD ; Call RAM-resident JSRFAR routine.
      LDA $05 ; Get returned status
      PHA ; and push on stack.
      LDA $06 ; Restore 8502 .A, .X, and .Y registers.
      LDX $07
      LDY $08
      PLP ; Restore processor status.
      RTS ; Return to caller with 8502 registers
           ; containing data returned by the
           ; called routine (not the JSRFAR routine.)
-----
$1869 LDX #$02 ; Reset TXTPTR to point to $0200 start of
      LDY #$00 ; BASIC's text input buffer.
      STX $3E
      STY $3D
$1871 RTS

```

Figure 4-9. Extended, expression-evaluation VAL function, located in free RAM at \$1800, called by USR(x) function (set up USR vector with POKE 4633,0: POKE 4634,24 before using) (*continued*)

Figure 4-9 lists an expression-evaluator function called by USR(x). Accepting a string as input, this routine tokenizes the string and evaluates (executes) it, returning a result in FAC. A BASIC program that uses this routine can evaluate a string containing an expression entered by a user; in other words, the user does not have to edit a program to enter or change a formula (expression). This

```

10 POKE 4633,0:POKE 4634,24
20 INPUT "FORMULA":A$
30 BANK15: A=USR(A$): PRINT A
50 GOTO20

```

Figure 4-10. Simple BASIC program to test extended VAL function listed in Figure 4-9

routine is an example of how BASIC-created strings can be manipulated by machine language. Repeated use of this extended VAL function will slow program execution because it must retokenize the expression each time it is called. Figure 4-10 lists a simple program that allows the routine to be tested.

The BASIC jump table listed in Table 4-7 contains entries for arithmetic and data-conversion functions that most machine language programmers will need. Table 4-8 contains entries to major BASIC 7.0 interpreter routines.

Table 4-8. Entry Points of Major Routines in BASIC 7.0 ROM

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|--|
| | 4000 | Jump to cold-start code at \$4023. |
| | 4003 | Jump to warm-start code at \$4009. |
| | 4006 | Jump to \$A84D IRQ handler. |
| SOFT | 4009 | Warm-start routine—clear I/O channels, restore stack, disable sprite movement, reinitialize MMU. |
| HARD | 4023 | Cold-start routine. |

Table 4-8. Entry Points of Major Routines in BASIC 7.0 ROM (*continued*)

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|--------------------|---|
| INIT-STORAGE | 4045 | Initialize BASIC pointers and vectors. |
| INIT-VOICES | 4112 | Initialize sound chip and music variables. |
| INITMMU | 417A | Set MMU preconfiguration registers for BASIC's use. PCR a, \$FF01—select RAM0, no ROM, no I/O. PCR b, \$FF02—select RAM1, no ROM, no I/O. PCR c, \$FF03—select RAM0, BASIC/Kernal ROMs, no I/O. PCR d, \$FF04—select RAM1, BASIC/Kernal ROMs, no I/O. |
| STOP-SPRITES | 418D | Stop all sprite movement. |
| SIGNON | 419B | Print "COMMODORE Basic 7.0 . . ." startup message. |
| INITVEC | 4251 | Initialize BASIC indirect vectors in \$0300 region. |
| INITAT | 4279 | CHRGET routine that is moved to \$0380. |
| INDSUBRAM0 | 4298 | ; indirect-fetch routines downloaded to RAM. |
| INDSUBRAM1 | 42A4 | |
| INDINRAM1 | 42B0 | |
| INDIN2 | 42B9 | |
| INDTXT | 42C2 | |
| | 42CE | Predefined fetch from (\$50) bank 0. |
| | 42D3 | Predefined fetch from (\$3F) bank 1. |

Table 4-8. Entry Points of Major Routines in BASIC 7.0 ROM (*continued*)

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|--|
| | 42D8 | Predefined fetch from (\$52) bank 1. |
| | 42DD | Predefined fetch from (\$5C) bank 0. |
| | 42E2 | Predefined fetch from (\$5C) bank 1. |
| | 42E7 | Predefined fetch from (\$66) bank 1. |
| | 42EC | Predefined fetch from (\$61) bank 0. |
| | 42F1 | Predefined fetch from (\$70) bank 0. |
| | 42F6 | Predefined fetch from (\$70) bank 1. |
| | 42FB | Predefined fetch from (\$50) bank 1. |
| | 4300 | Predefined fetch from (\$61) bank 1. |
| | 4301 | Predefined fetch from (\$24) bank 0. |
| CRUNCH | 430A | Crunch tokens, vectored through \$0304. |
| ESCLK | 4321 | Escape token crunch, vectored through \$030C. |
| RESER | 43E2 | Checks if keyword found. |
| RESLST | 4417 | Table of keywords not requiring escape tokens. |
| OPLIST | 46FD | Keyword vectors. |
| OPTAB | 4828 | Operator vectors. |
| ERRTAB | 484B | Table of error messages. |
| ERSTUP | 4A82 | Translates error message (# in .A) to string start address in \$24/\$25. |

Table 4-8. Entry Points of Major Routines in BASIC 7.0 ROM (*continued*)

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|--------------------|--|
| GONE | 4AA2 | Read and execute next statement, vectored through \$0308. |
| NEWSTT | 4AF6 | Set up and execute a statement. |
| CONT | 4B34 | Update CONT pointer. |
| XEQCM3 | 4B3F | Execute statement. |
| STOP | 4BCB | Code for STOP. |
| END | 4BCD | Code for END. |
| GETFMN | 4BF7 | Function handler. |
| FINGO | 4C3F | Convert a token to an index value for jump table. |
| OROP | 4C86 | Code for OR. |
| ANDOP | 4C89 | Code for AND. |
| DOREL | 4C86 | Check for matching substrings. |
| REDDY | 4D2A | Print "READY." |
| ERROR | 4D3F | Output error message, vectored through \$0300. |
| MAIN | 4DC6 | BASIC main input loop, vectored through \$0302. Gets text into input buffer, tokenizes, and executes or stores line. |
| MAIN1 | 4DE2 | Handle new line input, check line number, and so on. Make room for line, if necessary, by moving other parts of program. |
| LNKPRG | 4F4F | Relink lines after lines deleted, added, or modified. |
| INLIN | 4F93 | Routine that gets characters from keyboard and stores in input buffer at \$0200. |
| SEARCH | 4FAA | Search BASIC stack for FOR/NEXT, DEF FNx data. |
| | 4FFE | Insert data in run-time stack. |

Table 4-8. Entry Points of Major Routines in BASIC 7.0 ROM (*continued*)

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|--------------------|--|
| | 5017 | Check if room for more string storage. |
| | 5047 | Save stack pointer. |
| | 5050 | Set stack pointer. |
| | 5059 | Remove entry from stack. |
| FNDLIN | 5064 | Find BASIC line whose line number is contained in \$16/\$17. |
| LINGET | 50A0 | Line number input and handler. |
| LIST | 50E2 | Code for LIST. |
| QPLOP | 5151 | LIST subroutine, vectored through \$0306. |
| ES CPR | 51CD | Escape token print, vectored through \$030E. |
| NEW | 51D6 | Code for NEW. |
| RUNC | 51F3 | Prepare to RUN program. |
| CLEAR | 51F8 | Code for CLR. |
| LDCLR | 5238 | Free 8502 stack space. |
| STXPT | 5254 | Decrement TXTPTR. |
| RETURN | 5262 | Code for RETURN. |
| DATA | 528F | Code for DATA. |
| REM | 529D | Code for REM. |
| DATAN | 52A2 | Scan to next statement for DATA. |
| REMN | 52A5 | Scan to next line because of REM statement. |
| IF | 52C5 | Code for IF. |
| | 5320 | Search/skip BEGIN/BEND. |
| UNQUOTE | 537C | Avoid fetching bytes in string constant. |
| | 5391 | Code for ELSE. |
| ONGOTO | 53A3 | Code for ON. |

Table 4-8. Entry Points of Major Routines in BASIC 7.0 ROM (*continued*)

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| LET | 53C6 | Code for LET. |
| STRADJ | 54F6 | Tests string descriptors. |
| PRINTN | 553A | Code for PRINT#. |
| CMD | 5540 | Code for CMD. |
| PRINT | 555A | Code for PRINT. |
| OUTSPC | 5600 | Print format character. |
| GET | 5612 | Code for GET. |
| INPUTN | 5648 | Code for INPUT#. |
| INPUT | 5662 | Perform INPUT. |
| QINLIN | 569C | Print "?" and take input. |
| READ | 56A9 | Code for READ. |
| NEXT | 57F4 | Code for NEXT. |
| DIM | 587B | Code for DIM. |
| SYS | 5885 | Code for SYS. |
| | 58B4 | Code for TRON. |
| | 58B7 | Code for TROFF. |
| | 588D | Code for RREG. |
| | 5975 | Code for AUTO. |
| | 5986 | Code for HELP. |
| | 59AC | Place HELP underline marker in displayed text. |
| GOSUB | 59CF | Code for GOSUB. |
| GOTO | 59D8 | Code for GOTO. |
| | 5A1D | Push return address to stack for GOSUB. |
| | 5A3D | Code for GO. |
| CONT | 5A60 | Code for CONT. |
| RUN | 5A9B | Code for RUN. |
| NEWSTT | 5AA6 | Get next BASIC statement. |
| RESTRE | 5ACA | Code for RESTORE. |

Table 4-8. Entry Points of Major Routines in BASIC 7.0 ROM (*continued*)

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|--------------------|--|
| | 5AF8 | Code for RENUMBER. |
| | 5BFB | Scan program for embedded line numbers that need changing during RENUMBER. |
| | 5D19 | Change line number for RENUMBER. |
| | 5D68 | Find RENUMBER start. |
| | 5D75 | Line count for RENUMBER. |
| | 5D89 | RENUMBER increment subroutine. |
| | 5D99 | RENUMBER: Scan forward. |
| | 5DA7 | Block move. |
| | 5DC6 | Move block down. |
| | 5DDF | Move block up. |
| | 5DF9 | Code for FOR. |
| FOR | 5E87 | Code for DELETE. |
| | 5EFB | Get line range. |
| | 5F34 | Code for PUDEF. |
| | 5F4D | Code for TRAP. |
| | 5F62 | Code for RESUME. |
| | 5FB7 | Restore TRAP pointer. |
| | 5FD8 | Bad syntax — exit. |
| RESCNT | 5FDB | Print “CAN’T RESUME” error message. |
| | 5FE0 | Code for DO. |
| | 6039 | Code for EXIT. |
| | 608A | Code for LOOP. |
| | 60B4 | ”LOOP NOT FOUND” error message. |
| | 60B7 | ”LOOP WITHOUT DO” error message. |

Table 4-B. Entry Points of Major Routines in BASIC 7.0 ROM (*continued*)

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|--------------------------------|---|
| PKYDF | 60E1 | Define programmable key. |
| | 610A | Code for KEY. |
| | 61A8 | Code for PAINT. |
| | 62B7 | Code for BOX. |
| | 642B | Code for SSHAPE. |
| | 658D | Code for GSHAPE. |
| CIRSUB | 668E | Code for CIRCLE. |
| | 6750 | Find next point on circle. |
| | 6797 | Code for DRAW. |
| | 67D7 | Code for CHAR. |
| | 6955 | Code for LOCATE. |
| | 6960 | Code for SCALE. |
| | 69E2 | Code for COLOR. |
| | 6A5C | Store current colors. |
| | 6A79 | Code for SCNCLR. |
| | 6B06 | Fill memory page. |
| | 6B17 | Set screen color. |
| | 6B30 | High-resolution bit-map screen cleared. |
| | 685A | Code for GRAPHIC. |
| | 6BC9 | Code for BANK. |
| | 6BD7 | Code for SLEEP. |
| | 6C2D | Code for WAIT. |
| | 6C4F | Code for SPRITE. |
| | 6CC6 | Code for MOVSPR. |
| | 6DE1 | Code for PLAY. |
| | 6E02 | PLAY music subroutine. |
| 6EB2 | Set SID sound characteristics. | |
| 6EFD | PLAY error handler. | |
| 6FD7 | Code for TEMPO. | |

Table 4-8. Entry Points of Major Routines in BASIC 7.0 ROM (*continued*)

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|--|
| | 7046 | Code for FILTER. |
| | 70C1 | Code for ENVELOPE. |
| | 7164 | Code for COLLISION. |
| | 7190 | Code for SPRCOLOR. |
| | 71B6 | Code for WIDTH. |
| | 71C5 | Code for VOL. |
| | 71EC | Code for SOUND. |
| | 72CC | Code for WINDOW. |
| | 7335 | Code for BOOT. |
| | 7372 | Code for SPRDEF. |
| | 76EC | Code for SPRSAV. |
| | 77B3 | Code for FAST. |
| | 77C4 | Code for SLOW. |
| FRMNUM | 77D7 | Type match check. Checks for string or numeric format. |
| | 77E7 | "TYPE MISMATCH" error message. |
| | 77EA | "FORMULA TOO COMPLEX" error message. |
| FRMEVL | 77EF | Evaluate expression. |
| EVAL | 78D7 | Evaluate individual token, vectored through \$030A. |
| GIVAYF | 793C | Convert integer to floating-point format. |
| PARCHK | 7950 | Evaluate expression within parentheses. |
| CHKCOM | 795C | Check for presence of comma. |
| SYNERR | 796C | Syntax error output. |
| ISVAR | 7978 | Search for variable header. |
| MOVFRM | 7A85 | Unpack RAM to floating-point accumulator #1 (\$AF60). |

Table 4-8. Entry Points of Major Routines in BASIC 7.0 ROM (*continued*)

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| DOREL | | Check variable type. |
| NUMREL | | Check if numeric. |
| STREL | 7B3C | Check if string. |
| NOTFNS | 7B46 | Create new variable and open up space if necessary. |
| NOTFDD | 7CAB | DIM an array. |
| BSERR | 7D25 | "BAD SUBSCRIPT" error message. |
| | 7D28 | "ILLEGAL QUANTITY" error message. |
| UNULT | 7E3E | Calculate array size using number of elements/dimension. |
| | 7E71 | Array pointer handler subroutine. |
| FRE | 8000 | Code for FRE. |
| VAL | 804A | Code for VAL. |
| VAL1 | 8052 | Code to convert ASCII string to floating-point. |
| | 8076 | Code for DEC. |
| PEEK | 80C5 | Code for PEEK. |
| POKE | 80E5 | Code for POKE. |
| | 80F6 | Code for ERR\$. |
| | 8139 | Swap 8502 .X with .Y. |
| | 8142 | Code for HEX\$. |
| | 816B | Convert byte to hex. |
| | 8182 | Code for RGR. |
| | 818C | Get graphics mode. |
| | 819B | Code for RCLR. |
| | 8203 | Code for JOY. |
| | 824D | Code for POT. |

Table 4-8. Entry Points of Major Routines in BASIC 7.0 ROM (*continued*)

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|--|
| | 82AE | Code for PEN. |
| | 82FA | Code for POINTER. |
| | 831E | Code for RSPRITE. |
| | 837C | Code for BUMP. |
| | 8397 | Code for RSPOS. |
| | 83E1 | Code for XOR. |
| | 8407 | Code for RWINDOW. |
| | 8434 | Code for RND random number generator. |
| | 84A7 | Evaluate fixed number. |
| AYINT | 84B4 | Conversion from floating-point to signed integer format. |
| POS | 84C9 | Code for POS. |
| ERRDIR | 84D9 | Check if BASIC is in direct (non-program) mode. |
| | 84DD | "ILLEGAL DIRECT" error message. |
| | 84E0 | "UNDEF'D FUNCTION" error message. |
| | 84F5 | "DIRECT MODE ONLY" error message. |
| DEF | 84FA | Code for DEF. |
| GETFNM | 8528 | Check FN syntax. |
| FNDORE | 853B | Code for FN. |
| STRD | 85AE | Code for STR\$. |
| CHRd | 85BF | Code for CHR\$. |
| LEFD | 85D6 | Code for LEFT\$. |
| RIGHTD | 860A | Code for RIGHT\$. |
| MIDD | 861C | Code for MID\$. |
| PREAM | 864D | Set up string parameters. |
| LEN | 8668 | Code for LEN. |

Table 4-8. Entry Points of Major Routines in BASIC 7.0 ROM (*continued*)

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|--|
| LEN1 | 866E | Exit from string mode. Bank 1 RAM remains selected on exit. String length is returned in .A, and address is contained in \$24/\$25 (INDEX1). |
| ASC | 8677 | Code for ASC. |
| | 8688 | String vector determination. |
| | 869A | String setup. |
| MOVINS | 874E | Store string. |
| FRESTR | 877B | Discard unwanted string. |
| FRETMS | 87E0 | Clean three-string descriptor stack. |
| | 87F1 | Get byte value. |
| | 8803 | Get parameters for POKE and WAIT instructions. |
| GETADR | 8815 | Convert floating-point number to a 16-bit unsigned address-format integer. |
| FSUB | 882E | Code for SUBTRACT. |
| FSUBT | 8831 | ARG and FAC. |
| FADD | 8845 | Add memory to FAC. |
| FADDT | 8848 | Code for ADD. |
| NORML | 8917 | Normalize FAC. |
| | 894E | Rounding of FAC. |
| OVERR | 895D | "OVERFLOW" error message. |
| LOG | 89CA | Code for LOG. |
| | 8A0E | Add 0.5 to FAC. |
| FMULT | 8A24 | FAC = Memory * FAC. |
| FMULTT | 8A27 | Code for MULTIPLY. |
| ROMUPK | 8A89 | Transfer ROM data to ARG. |
| CONUPK | 8AB4 | Transfer RAM data to ARG. |

Table 4-8. Entry Points of Major Routines in BASIC 7.0 ROM (*continued*)

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|--|
| MUL10 | 8B17 | Multiply by 10. |
| | 8B33 | "DIVISION BY ZERO" error message. |
| DIV10 | 8B38 | Divide FAC by 10. |
| FDIV | 8B49 | Divide FAC into memory. |
| FDIVT | 8B4C | Code for DIVIDE. |
| MOVFM | 8BD4 | Transfer ROM data to FAC. |
| MOV2F | 8BF9 | Transfer FAC to TEMPF2 (FAC#4) at \$5E. |
| MOV1F | 8BFC | Transfer FAC to TEMPF1 (FAC#3) at \$59. |
| MOVFM | 8C00 | Transfer FAC to RAM. |
| MOVFA | 8C28 | Transfer ARG to FAC. |
| MOVAF | 8C38 | Transfer FAC to ARG. |
| ROUND | 8C47 | Round FAC. |
| SIGN | 8C57 | Get sign. \$00 if FAC is zero; \$FF if negative. |
| SGN | 8C65 | Code for SGN. |
| ACTOFC | | |
| INTOFC | | |
| FLOATC | 8C75 | Convert signed integer to floating-point format. |
| ABS | 8C84 | Code for ABS (\$AF4E). |
| FCOMP | 8C87 | Compare FAC to memory. \$00 result if equal. |
| QINT | 8CC7 | Convert FLP to signed integer. |
| INT | 8CFB | Code for INT. |
| FIN | 8D22 | Convert ASCII digit string to floating-point number. |
| FINLOG | 8DB0 | Get ASCII digit. |
| INPRT | 8E26 | "IN" message. |

Table 4-8. Entry Points of Major Routines in BASIC 7.0 ROM (*continued*)

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|--------------------|--|
| LINPRT | 8E32 | Print integer. |
| FOUT | 8E42 | Convert floating-point number to ASCII. |
| SQR | 8FB7 | Code for SQR (\$AF30). |
| FPWR | 8FBE | Raise to power (\$AF36). |
| FPWRT | 8FC1 | Code for exponent (\$AF39). |
| NEGOP | 8FFA | Code for negation of FAC. |
| EXP | 9033 | Code for EXP (\$AF3C). |
| POLYX | 9086 | Polynomial evaluator, Taylor series with odd-powers. |
| POLY | 909C | Polynomial evaluator, Taylor series with integer powers. |
| SAVET | 9112 | Code for SAVE. |
| VERIFYT | 9129 | Code for VERIFY. |
| LOADT | 912C | Code for LOAD. |
| OPENT | 918D | Code for OPEN. |
| CLOSET | 919A | Code for CLOSE. |
| SLPARA | 9299 | Make room for string. |
| COMBYT | | Check for presence of comma and get character. |
| CMMERR | | Check for last comma. |
| GARBA2 | 92EA | Garbage collection routine finds unused strings. |
| COS | 9409 | Code for COS (\$AF3F). |
| SIN | 9410 | Code for SIN (\$AF42). |
| TAN | 9459 | Code for TAN (\$AF45). |
| ATN | 9483 | Code for ATN (\$AF48). |
| | 9520 | PRINT USING subroutine. |
| | 99C1 | Code for INSTR. |
| | 9B0C | Code for RDOT. |

Table 4-8. Entry Points of Major Routines in BASIC 7.0 ROM (*continued*)

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|---|
| DRAWLN | 9B30 | Draw a line (\$AF72). |
| GPLOT | 1BFB | Plot a point (\$AF75). |
| | 9C70 | Set high-resolution bit-map color cell. |
| | 9E2F | Parse GRAPHIC command. |
| | A07E | Code for CATALOG/ DIRECTORY. |
| | A11D | Code for DOPEN. |
| | A134 | Code for APPEND. |
| | A157 | Find next secondary address. |
| | A16F | Code for DCLOSE. |
| | A18C | Code for DSAVE. |
| | A1A4 | Code for DVERIFY. |
| | A1A7 | Code for DLOAD. |
| | A1C8 | Code for BSAVE. |
| | A218 | Code for BLOAD. |
| | A267 | Code for HEADER. |
| | A2A1 | Code for SCRATCH. |
| | A2D7 | Code for RECORD. |
| | A322 | Code for DCLEAR. |
| | A32F | Code for COLLECT. |
| | A346 | Code for COPY. |
| | A362 | Code for CONCAT. |
| | A36E | Code for RENAME. |
| | A37C | Code for BACKUP. |
| | A3BF | Parse DOS commands. |
| | A5E7 | "MISSING FILE NAME" error message. |
| | A5EA | "ILLEGAL DEVICE NUMBER" error message. |

Table 4-8. Entry Points of Major Routines in BASIC 7.0 ROM (*continued*)

| <i>Label</i> | <i>Hex Address</i> | <i>Comments</i> |
|--------------|------------------------|--|
| | A5ED | "STRING TOO LONG" error message. |
| | A7E1 | "ARE YOU SURE?" error message. |
| | A845 | Select Bank 15—RAM0, ROMs, and I/O bank. |
| | A84D | BASIC 7.0 IRQ service routine. Loads VIC-II with contents of shadow registers, handles split screen, sprite collision, and light-pen events. |
| | AF00-AFA7 | BASIC 7.0 jump-table entries |

Chapter 5

The C-128 Video System

The C-128 system contains a powerful video system based upon two quite different video-display processor chips, the VIC-II (8564) and the 8563. The VIC-II chip allows the C-128 to have all of the sprite graphics capabilities and the 40-column text display of the famous C-64; the 8563 video chip allows a true 80-column display and an ultrahigh-resolution, 640- by 200-pixel graphics display. The 8563, in addition, has its own independent 16K of video RAM that is not directly accessible to the 8502 or Z-80 microprocessor.

The software contained in the screen editor and BASIC 7.0 ROMs allows a lot of user control over each of these processors. Special editor *escape* commands perform common text-editing func-

tions, while BASIC 7.0 provides complete control of VIC-II graphics operations. Unfortunately, the ROM system software does not take advantage of any of the graphics capabilities of the 8563 chip.

Screen Editor Escape And Control Sequences

The C-128's screen editor system is much more comprehensive than that of previous PET/CBM computers. Whereas the older screen editors provided only minimal operations (such as cursor movements and character insertions and deletions), the C-128's screen editor has over 30 special functions that make editing a program (or text, for that matter) a breeze. Through the use of the ESC key in combination with another character, a user can execute such complex screen-editing operations as line insertion or deletion, partial line or screen erasures, cursor selection, and tab settings. Most of these functions, except where noted, apply to both the 80- and 40-column text screens.

Screen editor escape sequences save programming time for BASIC 7.0 and machine language programmers, too. The escape code—CHR\$(27) in BASIC 7.0—can be included along with another character within a string; when the string is printed to the screen, the desired editing action occurs. For example, the simple string CHR\$(27) + "@" (\$1B \$40 in hexadecimal) erases the screen, starting at the current cursor position. It is much quicker—and shorter—than a BASIC loop like FOR I= S TO 2023: POKE I,0: NEXT. Such simplifications make the C-128's BASIC 7.0 a very effective programming language, because the screen editor routines do much of the programmer's work. C-128 programs therefore don't necessarily require the rigor and tedium of machine-code programming.

Programmers should be careful when using escape sequences that may inadvertently be sent to the printer. The escape sequence,

when sent to a printer, may trigger some unusual mode and result in a garbage printout. For example, the Commodore 1525, MPS-801, and MPS 803 dot-matrix printers regard a CHR\$(27) code as specifying a dot location where printing will begin. Printer interfaces that are used to connect a Commodore computer to a standard printer also may regard the escape code as a special character. Generally, diverting all screen output containing screen editor escape character codes is not a recommended practice because of inadvertent triggering of special printer features.

Besides these escape functions, the C-128 screen editors also use several control characters that engage special functions. In fact, the escape character itself is merely CTRL-[(left square bracket). All control codes are only a single byte. New control codes that have been added to the Commodore character set, or those that have modified functions, are listed in this chapter.

The following paragraphs list all C-128 escape and control functions, along with their ASCII decimal and hexadecimal equivalents. Where noted, the particular function operates only on the 80-column screen.

The descriptions of these screen editor codes often refer to *logical* and *physical* screen lines. A physical screen line is simply one actual line, 40 or 80 columns, depending upon which editor is currently being used. In contrast, a logical line refers primarily to lines displayed by BASIC's LIST command. These lines can be either one or two physical lines long in 80-column mode, or up to four physical lines long in 40-column mode. Even though the listed BASIC line may be much longer than one physical line, the line of BASIC is treated as though it were one line. Many editor escape sequences operate on logical lines and may actually affect several physical lines, especially when dealing with lines of BASIC 7.0 text. However, text printed to the screen can usually be regarded solely in terms of physical lines.

Editor Escape Sequences

ESC-A decimal 27, 65 \$1B, \$41

The ESC-A sequence enables auto-insert mode. When auto-insert is enabled, every time a character is entered in the middle of a screen line all characters are moved to the right one position. Characters can “wrap around” the end of a physical line. Auto-insert mode essentially functions as an insert character generated after every keystroke appears on the screen. This mode is disabled by ESC-C.

ESC-B decimal 27, 66 hex \$1B, \$42

The ESC-B sequence sets the bottom of the current screen window at the current cursor position. When defining a screen window using ESC sequences, an ESC-B sequence sets the current cursor position as the lower right corner of the desired window. If the cursor position is not in the proper column when an ESC-B is generated, the window will not be of the proper width.

ESC-C decimal 27, 67 hex \$1B, \$43

The ESC-C sequence merely turns off the auto-insert mode enabled by the ESC-A sequence described earlier. Although an ESC-O will disable the Commodore insert, quote, and reverse-video modes, it does not suppress auto-insert mode.

ESC-D decimal 27, 68 hex \$1B, \$44

The ESC-D sequence deletes the line on which the cursor is currently positioned. When a BASIC listing is being edited, ESC-D keystrokes will delete a complete line of BASIC text from the screen, even if it is longer than one physical line long. ESC-D will delete the whole logical line no matter what physical line of a BASIC statement the cursor is positioned upon. (The ESC-D sequence does not delete the line from the program resident in memory, but merely removes it from the current screen display. Re-LISTing the program will make the line appear again.)

ESC-E decimal 27, 69 hex \$1B, \$45

The ESC-E sequence disables cursor flashing. The cursor then appears as a solid block. (It may also, if selected, appear as an underline-style cursor in 80-column mode.)

ESC-F decimal 27, 70 hex \$1B, \$46

The ESC-F sequence enables cursor flashing.

ESC-G decimal 27, 71 hex \$1B, \$47

The ESC-G sequence allows a “bell” tone to be generated whenever a CTRL-G is encountered. The tone emanates from the monitor or television speaker, because the C-128 itself has no sound-producing device (transducer).

ESC-H decimal 27, 72 hex \$1B, \$48

The ESC-H sequence disables the production of a bell tone whenever a CTRL-G character is encountered.

ESC-I decimal 27, 73 hex \$1B, \$49

The ESC-I sequence inserts a blank line. The screen line containing the cursor is blanked, while the text of that line is copied to the next screen line. When a line of BASIC text is listed and occupies more than one physical line, an ESC-I sequence will “push down” all of the physical lines comprising that statement if the cursor is positioned on its first physical line. If the cursor is on a subsequent physical line of a long line of BASIC text, the ESC-I sequence will insert a blank line and copy the original text of that line onto the next physical line.

ESC-J decimal 27, 74 hex \$1B, \$4A

The ESC-J sequence moves the cursor to the start of the current logical screen line. Even if the cursor is on a subsequent physical line of BASIC text in a long statement, the cursor will be repositioned to the first character (that is, in the line number field) of that line of BASIC.

ESC-K decimal 27, 75 hex \$1B, \$4B

The ESC-K sequence moves the cursor to the end of the current logical screen line. Even if the cursor is on the first physical line of BASIC text in a long statement, the cursor will be repositioned to the last character of that line of BASIC.

ESC-L decimal 27, 76 hex \$1B, \$4C

The ESC-L sequence enables text to be scrolled off the screen. Text normally scrolls upward, although the ESC-W sequence scrolls text downward by one line. Both the ESC-V (upward scroll, one line) and ESC-W (downward scroll, one line) functions are not affected by the ESC-L sequence.

ESC-M decimal 27, 77 hex \$1B, \$4D

The ESC-M sequence prevents text from scrolling off the screen. When this non-scroll mode is enabled, a line that normally “pushes up” the rest of the text screen is instead displayed on the top screen line. Subsequent text will be displayed after the top screen line. Thus, long listings or disk directory displays will wrap around the screen several times and overwrite previously displayed text. Both the ESC-V (upward scroll, one line) and ESC-W (downward scroll, one line) functions are not disabled by the ESC-M sequence.

ESC-N decimal 27, 78 hex \$1B, \$4E

The ESC-N sequence applies only to 80-column screens. It returns the screen to normal, disabling the reverse-video screen display. Note that this escape function does not specifically apply to reverse characters themselves. For example, a white-on-black screen that had been reversed (using ESC-R, reverse 80-column screen function) to a black-on-white screen would be restored to a white-on-black (white text on black background) screen.

ESC-O decimal 27, 79 hex \$1B, \$4F

The ESC-O sequence cancels quote mode (where cursor movements show up as special graphics characters within strings), insert

mode, and reverse-character modes. ESC-O will not disable auto-insert mode or affect bell status, scrolling status, or other screen editor parameters.

ESC-P decimal 27, 80 hex \$1B, \$50

The ESC-P sequence erases text from the current cursor position to the beginning of the logical (not physical) screen line. The erasure will cross physical line boundaries if the cursor is positioned on a physical line other than the first.

ESC-Q decimal 27, 81 hex \$1B, \$51

The ESC-Q sequence erases text from the current cursor position to the end of the current logical line. The erasure will cross physical line boundaries if the logical line is more than one physical line long, and thus will erase subsequent physical lines of BASIC text.

ESC-R decimal 27, 82 hex \$1B, \$52

The ESC-R sequence applies only to 80-column screens. This function “flips” the screen colors, exchanging the background color for the character color. The 80-column screen can be restored to its normal color scheme by using the ESC-N sequence.

ESC-S decimal 27, 83 hex \$1B, \$53

The ESC-S sequence applies only to 80-column screens. This function causes the block cursor (full character height) to be displayed. The block cursor can be changed to an underline cursor by using the ESC-U sequence.

ESC-T decimal 27, 84 hex \$1B, \$54

The ESC-T sequence sets the top of the current screen window at the current cursor position. When a screen window is being defined using ESC sequences, an ESC-T sequence defines the current cursor position as the *upper left* corner of the desired window. If the cursor position is not in the proper column when an ESC-T is generated, the window will not be the proper size.

ESC-U decimal 27, 85 hex \$1B, \$55

The ESC-U sequence causes the underline cursor to be displayed; it applies only to the 80-column screen. The ESC-S function will switch the cursor to a block. Cursor flashing is disabled or enabled by the ESC-E and ESC-F sequences, respectively.

ESC-V decimal 27, 86 hex \$1B, \$56

The ESC-V sequence scrolls the screen up one line. Text scrolling off the top line of the screen is not recoverable by executing the scroll-down function. The ESC-V function is not affected by the ESC-L and ESC-M enable/disable scroll functions.

ESC-W decimal 27, 87 hex \$1B, \$57

The ESC-W sequence scrolls the screen down one line. Text scrolling off the bottom line of the screen is not recoverable by executing the scroll-up function. The ESC-W function is not affected by the ESC-L and ESC-M enable/disable scroll functions.

ESC-X decimal 27, 88 hex \$1B, \$58

The ESC-X sequence toggles between 40- and 80-column video output. It exchanges local screen editor variables at \$E0-\$F9 with those saved at \$0A40; the TAB stop and line-wrap bit maps at \$0354-\$0361 are exchanged with those saved at \$0A60.

ESC-Y decimal 27, 89 hex \$1B, \$59

The ESC-Y sequence sets the default TAB stops. Each TAB is eight spaces.

ESC-Z decimal 27, 90 hex \$1B, \$5A

The ESC-Z sequence clears all TAB stops. When the TAB key is pressed after an ESC-Z sequence has been generated, one TAB keystroke will move the cursor to the end of the physical screen line.

Editor Control Characters

CTRL-B decimal 2 hex \$02

The CTRL-B character applies only to the 80-column screen and turns on the 8563 underline attribute. This underlining can be disabled by a CHR\$(130) (hex \$82) character.

CTRL-G decimal 7 hex \$07

The CTRL-G character corresponds to the ASCII BEL character and produces a bell tone in the monitor's speaker when output is enabled by the ESC-G sequence. The bell tone output can be disabled using the ESC-H sequence.

CTRL-I decimal 9 hex \$09

The CTRL-I character tabs a preset number of spaces to the right. TABbing will not overwrite lines of BASIC text with spaces; each TAB stop can be set or cleared with the CTRL-X character. The ESC-Y sequence sets default TAB stops every eight spaces, while the ESC-Z sequence clears all TAB stops.

CTRL-J decimal 10 hex \$0A

The CTRL-J character corresponds to the ASCII LF line-feed character; text output occurs on the next screen line after a CTRL-J character is encountered.

CTRL-K decimal 11 hex \$0B

The CTRL-K character disables character-set switching using the SHIFT/COMMODORE key combination. This code replaces the CHR\$(9) code used on previous PET/CBM machines to disable case switching.

CTRL-L decimal 12 hex \$0C

The CTRL-L character enables character-set switching using the SHIFT/COMMODORE key combination. This code replaces the

CHR\$(8) code used on previous PET/ CBM machines to enable case switching.

CTRL-O decimal 15 hex \$0F

The CTRL-O character enables the character-flash attribute; it applies solely to the 80-column screen. This attribute can be disabled by a CHR\$(143) (hex \$8F) character.

CTRL-[decimal 27 hex \$1B

The CTRL-[character is merely the escape character. This control combination and the escape key both produce the same PetASCII character.

CHR\$(130) hex \$82

CHR\$(130) has no corresponding control-key character. It disables the underline attribute in 80-column mode that was enabled by CTRL-B.

CHR\$(143) hex \$8F

CHR\$(143) has no corresponding control-key character. It disables the character-flash attribute in 80-column mode that was enabled by CTRL-O.

The C-128's 80-Column Video Display System

The C-128's 8563 video chip generates the C-128's 80-column video output. It contains 37 internal registers. Its overall architecture and register layout is comparable to that of the industry-standard 6545E video controller. It also contains extra logic to manage an independent 16K block of video RAM. Although the C-128 only supports 8563 text manipulations, 640 × 200 bit-mapped ultrahigh-resolution graphics displays are possible. The 8563 also has the ability to display

characters from both of two 256-byte character sets simultaneously; thus, characters from the lower- and uppercase sets — as well as all graphics characters — can be on the screen at the same time.

The 8563 Independent RAM Block

The 8563's independent 16K RAM block is not directly accessible by either the 8502 or the Z-80 microprocessor; instead, commands must be sent to the 8563 chip, requesting to read or write data to the independent RAM. Table 5-1 shows a map of 8563 display RAM.

Normally, the 8563 is configured to display an 80×25 text screen. The lowest 2K of the 16K video RAM is used to store the actual characters forming the text display; data that is stored here is arranged in a line-by-line format. The data stored here can be thought of as an index value (or offset) into the character definition area. Storage is thus quite similar to that of the VIC-II video chip. The

Table 5-1. Map of 8563 Independent Display RAM Usage

| <i>Hex Address</i> | <i>Function</i> |
|--------------------|--|
| \$0000-\$07FF | Text storage for 80-column character matrix. |
| \$0800-\$0FFF | Character attribute storage; color data, reverse-video, flash, and underline attributes for every character are stored here. |
| \$2000-\$3FFF | Character definition storage. Each character has a 16-byte definition field, although only the first 8 bytes are normally used. (The remaining 8 bytes of the definition are \$00 null bytes.) |

index value is not an ASCII value, but rather a screen code. In fact, the screen codes for C-128 8563 and VIC-II characters are identical.

The next 2K segment of display RAM is allocated to character attribute storage. Each position in this region of display RAM corresponds with a character-storage position exactly 2048 bytes below it. Data stored in a character attribute location determines whether or not a character in a corresponding text-storage location is flashing, reversed, or underlined. Attribute bytes also contain RGBI (red, green, blue, and intensity) color data. Table 5-2 shows the bit assignments for a typical attribute byte.

The alternate character-set bit (bit 7) allows characters from both the uppercase/graphics and the upper/lowercase character sets to be on the screen at the same time. Since this really makes each character's screen code 9 bits long, the 8563 can display 512 different characters at once—avoiding the VIC-II's 256-character limitation. When the reverse video bit, bit 6, is set, it merely reverses the foreground and background colors for the character associated with the given attribute byte. Note that the C-128 character-set ROMs already include reversed character definitions (screen codes 128-255, or \$80-\$FF hex).

Bit 5, when high, selects the character underline attribute. The underline is the same color as the displayed character being underlined, and its thickness is determined by bits 4 through 0 of 8563 register 29. When bit 4 is high, character blinking is enabled, switch-

Table 5-2. Attribute Bit Assignments

| <i>Bit 7</i> | <i>Bit 6</i> | <i>Bit 5</i> | <i>Bit 4</i> | <i>Bit 3</i> | <i>Bit 2</i> | <i>Bit 1</i> | <i>Bit 0</i> |
|------------------------------------|------------------|----------------|--------------|--------------|--------------|--------------|--------------------|
| Alter- nate character set | Reverse video | Under- line | Blinking | Red bit | Green bit | Blue bit | Inten- sity bit |

ing between foreground and background colors. Blink rate is determined by bit 5 of 8563 register 24. Bits 3 through 1 of an attribute byte determine the color of its associated character, while bit 0 determines its intensity.

There is 4K of free RAM above the character and attribute storage that normally is unused by the C-128 system. The programmer can use this area as a “second screen” by merely setting up the appropriate pointers in the 8563 chip. And, because the region is 4K long, there’s enough storage for both a second text matrix and a corresponding attribute-storage region.

The display RAM contains long character definitions that are 16 bytes long at locations \$2000-\$3FFF. This 8K region normally contains 512 16-byte character definitions (two sets of 256 characters each). The 8563, when displaying normal 80-column text, needs only eight bytes for a character definition, so the extra eight bytes of the character definition field are occupied by \$00 “filler bytes.”

If character display is not needed, the 16K of RAM can contain a 640- by 200-pixel bit map, occupying exactly 16,000 bytes. Bit mapping, to be described later, can be selected by setting bit 7 of 8563 register #25.

The 8563 Registers

Table 5-3 is a map of the 8563 register structure. However, these registers are not directly accessible by either C-128 microprocessor; two special 8563 registers appear in the processor’s map (when the C-128 4K I/O block is in context) to allow communication between the processor and the 8563’s 37 registers and independent RAM.

Location \$D600, mnemonically referred to as VDCADR, contains six write-only address bits that select one of the 37 8563 internal registers. When read, status data is returned, indicating whether or not the output data is valid. Light-pen status is also returned; if the LP bit is high, the light pen has been triggered and has left the row/col-

Table 5-3. Internal 8563 Register Map. (These registers are not directly accessible by the processor. The CPU must use the 8563 register read/write routine in Figure 5-3. Bits flagged with an asterisk have no function.)

| | | <i>Register Bit #</i> | | | | | | | |
|-------------|----|------------------------------------|---------------|-------|----------------------------|---------------------------|---|----------------|---|
| <i>8563</i> | | | | | | | | | |
| <i>REG.</i> | # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| | 0 | Horizontal total | | | | | | | |
| | 1 | Horizontal displayed | | | | | | | |
| | 2 | Horizontal sync position | | | | | | | |
| | 3 | Vertical sync width | | | | Horizontal sync width | | | |
| | 4 | Vertical total | | | | | | | |
| | 5 | * | * | * | * | Vertical total adjust 0-4 | | | |
| | 6 | Vertical displayed | | | | | | | |
| | 7 | Vertical sync position | | | | | | | |
| | 8 | * | * | * | * | * | * | Interlace mode | |
| | 9 | * | * | * | Character total, vertical | | | | |
| | 10 | * | -Cursor mode- | | | Cursor start scan line | | | |
| | 11 | * | * | * | Cursor end scan line | | | | |
| | 12 | Display start address, high byte | | | | | | | |
| | 13 | Display start address, low byte | | | | | | | |
| | 14 | Cursor position, high byte | | | | | | | |
| | 15 | Cursor position, low byte | | | | | | | |
| | 16 | Light pen, vertical | | | | | | | |
| | 17 | Light pen, horizontal | | | | | | | |
| | 18 | Update location, high byte | | | | | | | |
| | 19 | Update location, low byte | | | | | | | |
| | 20 | Attribute start address, high byte | | | | | | | |
| | 21 | Attribute start address, low byte | | | | | | | |
| | 22 | Horiz. Character Total | | | | Horiz. chars. displayed | | | |
| | 23 | * | * | * | Chars. displayed, vertical | | | | |
| | 24 | Copy/ | Reverse | Blink | Vertical smooth scroll | | | | |
| | | fill | screen | rate | | | | | |

Table 5-3. Internal 8563 Register Map (*continued*)

| | | <i>Register Bit #</i> | | | | | | | |
|-------------|----------------------|---|---------------------------|-----------------------|---------------------|--------------------------|---------------------------|----------|----------|
| 8563 | | | | | | | | | |
| REG. | # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 25 | Text/graphics enable | Attrib. enable | Semi-graphics double | Pixel graphics double | — | Horizontal smooth scroll | — | — | — |
| 26 | — | — | Foreground color | — | — | — | Background color | — | — |
| 27 | — | — | Address increment per row | | | | — | — | — |
| 28 | — | Char. set address | — | RAM type | * | * | * | * | * |
| 29 | * | * | * | — | Underline scan line | — | — | — | — |
| 30 | — | Block fill/copy word count (value—1) | | | | — | — | — | — |
| 31 | — | CPU read/write data transfer register | | | | — | — | — | — |
| 32 | — | Block fill/copy source address, high byte | | | | — | — | — | — |
| 33 | — | Block fill/copy source address, low byte | | | | — | — | — | — |
| 34 | — | Display enable, begin | | | | — | — | — | — |
| 35 | — | Display enable, end | | | | — | — | — | — |
| 36 | * | * | * | * | * | — | # RAM refreshes/scan line | — | — |

umn scan address in two internal 8563 registers. The layout of VDCDAT and VCDADR is shown in Table 5-4.

Location \$D601 (called VDCDAT) is the actual 8563 processor-accessible I/O register. It will contain data to be sent to a particular 8563 register or to an address within the 16K RAM space. It may also contain data returned from an 8563 register or display RAM; in either case, the data is only valid if bit 7 of \$D600 is set. This bit should therefore be tested prior to beginning an 8563 read or write

Table 5-4. Register Layout of Processor-Accessible 8563 Registers

| <i>Bit 7</i> | <i>Bit 6</i> | <i>Bit 5</i> | <i>Bit 4</i> | <i>Bit 3</i> | <i>Bit 2</i> | <i>Bit 1</i> | <i>Bit 0</i> |
|------------------------|---------------------|-------------------|--------------|--------------|--------------|--------------|--------------|
| 8563 status | Light-pen triggered | Vertical blanking | * | * | * | * | * |
| VDCADR, \$D600, write. | | | | | | | |
| * | * | R5 | R4 | R3 | R2 | R1 | R0 |

VDCDAT, \$D601—Contains byte to be transferred to/from 8563.

operation. Figure 5-1 shows how to read from or write to an 8563 register.

The screen editor already contains these routines, so it's not necessary to include them in your programs. The read routine is called VDCIN and is located at \$CDDA in the screen editor region of

```

; Reads data from 8563 register (whose address
; is in .X) into .A.
;
; .X contains the desired 8563 register address.
; Store register address to VDCADR.
READ   LDX #$ regnum
loop   STX $D600
       BIT $D600
       BPL loop
       LDA $D601
       RTS
; Data in VDCDAT now valid, so it's safe to read.
; Return to caller with data in .A.
;
; Writes data from .A into 8563 register whose
; address is in .X.
;
; .X contains the desired 8563 register address.
; Store register number to VDCADR.
WRITE  LDX #$ regnum
loop   STX $D600
       BIT $D600
       BPL loop
       STA $D601
       RTS
; Now it's safe to store data.
; Return to caller.

```

Figure 5-1. Reading from or writing to an internal 8563 register

```
900 RD = 52698
910 SYS RD, 0, 26
920 RREG Q
930 RETURN
```

Figure 5-2. Reading an 8563 register

the Kernal ROM, while the write routine (sometimes called VDCOUT) is located at \$CDCC.

Anytime these routines need to be accessed from BASIC 7.0, the SYS command (with parameters) can be used. RREG is also used with the read routine to load the byte returned in the 8502 accumulator into a BASIC numeric variable. To read 8563 register 26 from BASIC, the statements in Figure 5-2 can be used.

Q will contain the contents of 8563 register 26 after RREG puts the value from .A into the variable Q. The zero in the SYS call is merely a placeholder. To write data to the 8563 from BASIC, the subroutine in Figure 5-3 can be used. This example writes a byte value from variable Q to 8563 register 26.

```
940 RD = 52684
950 SYS RD, Q, 26
960 RETURN
```

Figure 5-3. Writing to an 8563 register

Descriptions of Important 8563 Registers

Register 6—Vertical Displayed This register contains the number of screen lines displayed in one frame and determines the frame's vertical height (in characters). If you increase the number in this register, the 8563 can display more lines. A good-quality monochrome composite monitor can easily display 29 lines of text instead of the usual 25. The screen editor obviously does not take advantage of these extra screen lines, so they are untouched by editor, making them perfect for use as status lines. (The RAM devoted to display attribute storage will have to be moved up by $80*n$ bytes, where n is the number of additional lines added. Otherwise, attribute bytes will overwrite the extra text bytes; the extra text region will also overwrite the start of the attribute area, creating strange coloring, flashing, and underlining of characters at the top of the screen.)

Register 7—Vertical Sync Position This register contains one more than the number of screen lines from the start of the displayed part of the frame to the beginning of the vertical sync pulse.

Register 8—Interlace Mode Select The low two bits of this register determine the type of video interlacing used; the high five bits are currently unused.

- 00 Non-interlace mode (standard video)
- 10 Non-interlace mode (standard video)
- 01 Interlaced sync mode
- 11 Interlaced sync and video mode

The interlaced video modes provide a high-quality video signal to monitors that can accept such signals. Normal monitors do not require these signal types, and in fact will display a jittery image if fed an interlace-mode video signal.

Register 10—Cursor Start Scan Line, Bits 4-0

This nybble contains the top scan line of the cursor. Zero refers to the top line of a character pixel matrix. Increasing this value decreases the cursor's vertical size.

Register 10—Cursor Mode, Bits 5 and 6*Bits 6, 5 Cursor Mode*

| | |
|----|--------------------------|
| 00 | Solid, unblinking cursor |
| 01 | No cursor |
| 10 | Flashing cursor, fast |
| 11 | Flashing cursor, slow |

Register 11—Cursor End Scan Line This register contains the number of the last scan line of the cursor, plus 1. Increasing this value thickens the cursor.

Registers 12 and 13—Display Address, High and Low Bytes The two-byte value here determines the start address of the character-storage region in video RAM. This register pair normally points to \$0000.

Registers 14 and 15—Cursor Position, High and Low Bytes This register pair contains the current address of the cursor position. Note that it is not in row/ column format, but refers to a byte in character memory; character memory is arranged in line-by-line order.

Registers 16 and 17—Light-Pen Vertical and Horizontal These registers contain the value recorded when the light-pen input undergoes a positive-edge transition. Valid light-pen data is indicated by bit 6 of the 8563's status register at \$D600. The registers contain dot values, not character values.

Registers 18 and 19—Update Location, High and Low Bytes After an address is placed in this register pair, the processor can read from or write to 8563 video RAM. If data is placed in 8563 register 31 after the “update address” is set up, that particular location in 8563 RAM is changed. If 8563 register 31 is read instead of being written to, then the data at the update address in RAM will be sent to the VDCDAT register. Anytime data is written to registers 18 and 19, an automatic read operation is performed. The update address is also incremented automatically after every read or write operation. Using registers 18 and 19, along with register 31, is the only way the processor can access 8563 RAM. The automatic increment of the update address lets continued reads from register 31 (CPU data) fetch subsequent data in RAM.

Registers 20 and 21—Attribute Table Start Address High and Low Bytes These bytes contain the start address of the attribute table.

Register 24—Character Blink Rate, Character 5 This bit determines the character blink rate if the blinking attribute is set. If set, the character will flash quickly. If the bit is clear, the character will flash at half the previous rate.

Register 24—Reverse Screen, Bit 6 This bit exchanges foreground and background colors when set.

Register 24—Block Copy/Fill, Bit 7 This bit, when set, engages the block copy function, enabling one area of RAM to be copied to another. Register 30 contains the number of bytes to be copied, minus one, while registers 32 and 33 contain the start address of the region to be copied. The destination address for a block copy operation should be placed in registers 18 and 19 prior to setting this bit. When bit 7 is clear, a block fill operation takes place instead. The data byte used to fill a memory region should be placed in register 31

prior to setting up the address in registers 18 and 19. See the high-resolution graphics demonstration program for an example of the block fill operation.

Register 25—Attribute Disable, Bit 6 If this bit is clear, no display attributes will be used by the 8563, freeing display RAM for other uses such as alternate screens.

Register 25—Bit-Map Mode, Bit 7 If this bit is clear, the 8563 functions display normal text. If set, the 16K of RAM stores a 640×200 high-resolution bit map.

**Register 26—Background Color (R,G,B,I), Bits 3-0
—Foreground Color (R,G,B,I), Bits 7-4**

**Register 28—Character Set Address, Bits 7-5—
Underline Scan Line Count, Bits 4-0** Bits 7 through 5 contain the high three bits of the start address of the character definitions. Bits 4 through 0 contain the number of scan lines used for the underline character. Scan line #0 is at the top of a character.

**Registers 32 and 33—Block Copy Start Address
(High and Low)** This register pair contains the start address of one RAM region that will be transferred to another.

High-Resolution Bit-Mapped Graphics on the 8563

The 8563 supports a high-resolution graphics mode that allows a 640×200 pixel image to be displayed, twice the horizontal resolution of the VIC-II graphics screen. This bit map occupies 16000 bytes of the 16384-byte independent video RAM area, so storage of character definitions and text screen(s) are destroyed.

Figure 5-4 lists a program that allows bit-mapped graphic displays to be created. It uses the screen editor routines at \$CDCC and \$CDDA to read from or write to a specific 8563 register. Lines 100-210 of the listing merely take care of “housekeeping” functions such as screen scaling, error trapping, and so forth.

To invoke this bit-map mode, bit #7 of 8563 register #25 is set. Since the other bits of this register are not relevant to graphics operations, 128 (\$80 hex) can merely be stored to register #25, avoiding the cumbersome and time-consuming register-read, OR operation, and register-write sequence. Line 260 enables the graphics mode and sets the foreground and background colors by writing 113 (\$71) to 8563 register #26.

Since the data in the text screen and character definition storage areas of the 16K video RAM is now regarded as bit-map display information, it must be cleared. The 8563’s block-fill operation can be used to do this; first, line 270 sets 8563 registers 18 and 19 (update

```

100 DEF FNA(X)=SIN(X)
110 TRAP310:FAST: BANK15:WR=52684:RE=52698:FORI=0T07:PT(I)=2*(7-I):NEXT
120 INPUT "..... MAXIMUM (X)= ";MX
130 INPUT "..... MINIMUM (X)= ";MN
140 INPUT "EXPECTED MAXIMUM (Y)= ";Y2
150 INPUT "EXPECTED MINIMUM (Y) ";Y1
160 IF MN>MX THEN T=MN:MN=MX:MX=T
170 IF Y1>Y2 THEN T=Y1:Y1=Y2:Y2=T
180 VS=200/ABS(Y2-Y1)
190 VO=100-Y2-Y1
200 HS=ABS(MX-MN)/640
210 GOTO 260
220 AD=INT(X)/8+INT(Y)*80:H=AD/256:L=ADAND255
230 SYS WR,H,18:SYS WR,L,19:SYS RE,0,31:RREG BY
240 SYS WR,H,18:SYS WR,L,19:SYS WR,BY OR PT(XAND7),31
250 RETURN
260 SYS WR,128,25:SYS WR,113,26
270 SYS WR,0,18:SYS WR,0,19
280 FORI=0 TO 63: SYS WR,0,31:SYS WR,255,30:NEXT
290 FOR X=0T0639:Y=VO-VS*FNA(X*HS+MN):GOSUB220:NEXT
300 GETKEY AS
310 SYS WR,64,25:SYS WR,240,26
320 BANK15:FORI=0T08:SYSWR,32,31:SYSWR,255,30:NEXT:SYS49191
330 PRINT ERRS(ER); " ERROR IN ";EL
340 END

```

Figure 5-4. Plotter demonstration using 8563 bit-mapped graphics

location, high and low) to the start of video RAM, \$0000. Since bit #7 of register #24 is normally clear while BASIC's screen editor is in control of the 8563, a block fill operation (instead of a block copy operation) has already been selected, and no mode setup is necessary. Every time a loop in line 280 executes, 256 bytes of the display RAM are cleared; first, the desired filler byte (0) is written to 8563 register #31 (CPU read/write data). This action is followed by writing the desired number of locations to be filled to 8563 register #30 (word count-1). Since 255 has been stored to register #30, 256 bytes of display RAM are cleared every time line 280 loops. There is no need to change any data in 8563 registers 18 and 19 since they are automatically "post-incrementing"; in other words, the value of the update location register pair is increased by one after every write-to-RAM operation.

Once the screen is cleared, it is ready to receive the plotted data; the subroutine beginning at line 220 accomplishes the task of plotting a point on the bit map. First, the desired memory address is determined. Since each row of pixels is 80 characters of 8 pixels, the Y value must be multiplied by 80 and added to the x value divided by 8. (Note that all pixel locations use absolute, not relative, coordinates here. Before a functional value of the formula is plotted, it must be scaled appropriately. The beginning lines of this plotting program accomplish these scaling operations.) Once this address has been calculated, its high and low bytes are found. These address bytes are written to 8563 registers 18 and 19 (update address, high and low), and the byte contained at that address is read from 8563 register #31 (CPU read/write data). The appropriate bit to be set is determined by the expression

$$\text{NEWBYTE} = \text{OLDBYTE OR } 2^{\text{X AND } 7}$$

The array PT, after initialization, contains successive powers of two so that time-consuming exponentiation operations do not have to occur in the plotting subroutine (see line 240). The byte, with the

appropriate bit set, is then written back to the same address within 8563 of RAM, from which it was obtained.

After the plotting ends, bit #7 of 8563 register #24 is cleared in order to restore text mode display. Register 26 is loaded with the foreground and background text display colors, while line 320 fills text-screen RAM (\$0000—\$07FF) with the screen code for a space character. (Otherwise, garbage will appear on the text screen.) The program terminates after the screen editor routine INIT80 (located at \$C027 hex, or 49191 decimal) is called to restore the RAM-resident character definitions.

User-Defined Character Sets For the 80-Column Text Screen

Character sets created for the VIC-II chip can be used with the 8563 chip, providing that a special routine handles the transfer. Data for the character patterns displayed by both the VIC-II and 8563 are arranged similarly, although each space for an 8563 character definition occupies 16 bytes; the last 8 are unused and can contain \$00 “filler bytes.” The listing in Figure 5-5 contains a routine that downloads a bank 0 RAM-resident character set to video RAM.

First, pointers are set up to the user-defined character set. The routine then sets 8563 Registers 18 and 19 (update location, high and low bytes) to \$2000, the start address of character definitions in 8563 video RAM. Eight bytes are fetched (per loop) from the new character set and written to the 8563’s RAM. Note how the auto-incrementing of 8563 registers 18 and 19 lets programmers avoid having to reload these registers after each operation. After the 8 bytes are transferred, 8 \$00 “filler bytes” are written to 8563 RAM to account for the 16-byte character definition field. The pointer address is correspondingly updated by adding 8 to it, so that both the fetching and writing of character data bytes are “in step.” Appendix A contains a memory dump of a new character set called “Digifont.” It can be entered via the monitor.

```

0B00 A9 00 LDA #S00 ; select Bank 15 - ROMs, RAM0, and I/O block
0B02 8D 00 FF STA $FF00 ; store MMU data to Configuration Register.
0B05 A0 A0 LDY #SA0 ; set up zero-page pointer to character
0B07 A2 00 LDX #S00 ; definitions that begin at SA000.
0B09 84 FC STY $FC
0B0B 86 FB STX $FB
0B0D A9 20 LDA #S20 ; set 8563 Update Location register pair
0B0F A2 12 LDX #S12 ; to point to character definition region
; in 8563 independent RAM ($2000).
0B11 20 CC CD JSR $CDCC ; write to 8563 chip - Update Location, High
0B14 A9 00 LDA #S00
0B16 E8 INX
0B17 20 CC CD JSR $CDCC ; write to 8563 chip - Update Location, Low
0B1A A2 1F LDX #S1F ; Select 8563 register #31 (CPU Read/Write
; Data) for following operations.
0B1C A0 00 LDY #S00 ; Zero counter.
0B1E A9 3F LDA #S3F ; Select RAM Bank 0 and
0B20 8D 00 FF STA $FF00 ; store MMU data to Configuration Register.
0B23 B1 FB LDA ($FB),Y ; Fetch byte from character definitions
; in system RAM (bank 0),
0B25 48 PHA ; save temporarily,
0B26 A9 00 LDA #S00 ; and select System Bank (RAM0, ROM, I/O)
0B28 8D 00 FF STA $FF00 ; by storing new MMU data to config. reg.
0B2B 68 PLA ; Get back the byte.
0B2C 20 CC CD JSR $CDCC ; Write to 8563 RAM. 8563 Registers 18 & 19
; automatically incremented.
0B2F C8 INY ; Bump counter.
0B30 C0 08 CPY #S08 ; Eight times through loop? (eight bytes
; per character.)
0B32 90 EA BCC $0B1E ; No, loop again.
0B34 A9 00 LDA #S00 ; Else, fill remaining eight bytes of 8563
; definitions with S00 filler bytes.
0B36 A0 08 LDY #S08 ; Eight times through loop.
0B38 20 CC CD JSR $CDCC ; Write to 8563 & independent RAM.
0B3B 88 DEY ; Decrement counter.
0B3C D0 FA BNE $0B38 ; Loop again if not done.
0B3E A5 FB LDA $FB ; Get low-byte of zero-page pointer.
0B40 18 CLC ; Prepare for addition.
0B41 69 08 ADC #S08 ; Add 8 (to compensate for 8 "filler bytes".)
0B43 85 FB STA $FB ; Store back to low byte of pointer.
0B45 90 02 BCC $0B49 ; Carry? No, then branch.
0B47 E6 FC INC $FC ; Else also increment high byte.
0B49 A5 FC LDA $FC ; Get high byte of pointer.
0B4B C9 B0 CMP #S80 ; and see if transfer complete.
0B4D D0 CF BNE $0B1E ; No, go through loop again.
0B4F 60 RTS ; Else, return to caller.

```

Figure 5-5. Character set loader routine. The new character set is expected to reside in bank 0 RAM at \$A000-\$AFFF. This routine resides in the cassette buffer and its calling address is \$0B00 hex (2816 decimal). To restore the normal character set (and clear these new definitions), the INIT80 routine at \$C027 should be called.

The 8563 register-write routine at \$CDCC (in the screen editor ROM) is employed to handle the testing of the 8563 status bit. Its companion register-write routine (at \$CDDA) could be used if character definitions need modifications periodically. The original Com-

modore character set can be restored (that is, transferred from the character ROM at \$D000 to the 8563's RAM) by calling the INIT80 routine at \$C027 (49191 decimal).

The VIC-II Video Chip

The C-128 also contains the VIC-II chip that made the C-64 famous for its graphics capabilities. Programming the VIC-II chip in a C-128 environment is, for the most part, quite similar to programming in a C-64 system. There are a few extra details—such as “shadow registers” and IRQ interrupts—that must be taken into account when programming the VIC-II while in C-128 mode. Figure 5-6 shows a map of the VIC-II chip registers.

Banked Memory and the VIC-II

The VIC-II chip can access only 16K of RAM (within a given 64K region) due to its limited number of addressing lines. Bits 1 and 0 in location \$DD00 (the CIA #2 I/O chip) select which one of the four 16K ranges of RAM (within a 64K bank) that the VIC-II will access. Normally, the C-128 locates VIC-II memory in the lowest 16K block of RAM bank 0, at \$0000-\$03FF. Of course, the VIC-II usually needs only a maximum of 10K of RAM, if bit-mapped graphics are being displayed.

The RAM configuration register (RCR) of the MMU chip determines the 64K RAM bank from which the VIC-II will read and write data. Bits 7 and 6 of the RCR (at \$D506) select one of four 64K RAM banks for the VIC-II to access; thus, the 16K block of RAM does not necessarily have to be contained in RAM bank 0.

“Shadow” Registers

In C-128 mode, direct writes to VIC-II registers usually have only a momentary effect; during every IRQ interrupt cycle, the VIC-II registers are updated from “shadow registers” in low RAM memory.

| 53248-54271 | <i>VIC-II Chip Control Registers</i> |
|--------------------|---|
| 53248 | Sprite #0: X position |
| 53249 | Sprite #0: Y position |
| 53250 | Sprite #1: X position |
| 53251 | Sprite #1: Y position |
| 53252 | Sprite #2: X position |
| 53253 | Sprite #2: Y position |
| 53254 | Sprite #3: X position |
| 53255 | Sprite #3: Y position |
| 53256 | Sprite #4: X position |
| 53257 | Sprite #4: X position |
| 53258 | Sprite #5: X position |
| 53259 | Sprite #5: Y position |
| 53260 | Sprite #6: X position |
| 53261 | Sprite #6: X position |
| 53262 | Sprite #7: X position |
| 53263 | Sprite #7: Y position |
| 53264 | Sprites 0-7 MSB of X position |
| 53265 | VIC II Chip Control Register |
| | Bits 0-2: Smooth Scroll (Y direction) |
| | Bit 3: 24/25 Row Select (24=0) |
| | Bit 4: Screen Blanking (Blank=0) |
| | Bit 5: Enable Bit Map Mode (1=ON) |
| | Bit 6: Extended Color Text (1=ON) |
| | Bit 7: Raster Value Register (MSB) |
| 53266 | Raster Value Register (Bits 0-7) |
| 53267 | Light Pen (X Value) |
| 53268 | Light Pen (Y Value) |
| 53269 | Sprite Display Control Register |
| | Bit 0: Sprite #0 (1=ON) |
| | Bit 1: Sprite #1 (1=ON) |
| | Bit 2: Sprite #2 (1=ON) |
| | Bit 3: Sprite #3 (1=ON) |
| | Bit 4: Sprite #4 (1=ON) |

Figure 5-6. C-128 VIC-II video chip registers

| 53248-54271 | <i>VIC-II Chip Control Registers</i> |
|--------------------|---|
| | Bit 5: Sprite #5 (1=ON) |
| | Bit 6: Sprite #6 (1=ON) |
| | Bit 7: Sprite #7 (1=ON) |
| 53270 | VIC II Control Register #2 |
| | Bits 0-2: Smooth Scroll (X direction) |
| | Bit 3: 38/40 Column Select (1=40) |
| | Bit 4: Multicolor Mode (1=ON) |
| | Bit 5-7: Not used |
| 53271 | Sprite Expansion Register (Y direction) |
| | Bit 0: Sprite #0 (1=2X) |
| | Bit 1: Sprite #1 (1=2X) |
| | Bit 2: Sprite #2 (1=2X) |
| | Bit 3: Sprite #3 (1=2X) |
| | Bit 4: Sprite #4 (1=2X) |
| | Bit 5: Sprite #5 (1=2X) |
| | Bit 6: Sprite #6 (1=2X) |
| | Bit 7: Sprite #7 (1=2X) |
| 53272 | VIC II Address Control Register |
| | Bit 0: Not used |
| | Bits 1-3: Character Set Location |
| | Bits 4-7: Screen Location |
| 53273 | VIC II Interrupt Control Register |
| | Bit 0: Raster Compare |
| | Bit 1: Sprite-Background Collision |
| | Bit 2: Sprite-Sprite Collision |
| | Bit 3: Light Pen Interrupt |
| | Bit 4-6: Not used |
| 53274 | VIC II Interrupt Enable Register |
| | Bit 0: Raster Compare (1=ON) |
| | Bit 1: Sprite-Background Collision (1=ON) |
| | Bit 2: Sprite-Sprite Collision (1=ON) |
| | Bit 3: Light Pen Interrupt (1=ON) |
| | Bit 4-6: Not Used (1=ON) |

Figure 5-6. C-128 VIC-II video chip registers (*continued*)

| 53248-54271 | <i>VIC-II Chip Control Registers</i> |
|--------------------|--|
| 53275 | Sprite-Background Priority Register Bit 0: Sprite #0 (1=SPRITE) Bit 1: Sprite #1 (1=SPRITE) Bit 2: Sprite #2 (1=SPRITE) Bit 3: Sprite #3 (1=SPRITE) Bit 4: Sprite #4 (1=SPRITE) Bit 5: Sprite #5 (1=SPRITE) Bit 6: Sprite #6 (1=SPRITE) Bit 7: Sprite #7 (1=SPRITE) |
| 53276 | Sprite Multicolor Control Register Bit 0: Sprite #0 (1=Multicolor) Bit 1: Sprite #1 (1=Multicolor) Bit 2: Sprite #2 (1=Multicolor) Bit 3: Sprite #3 (1=Multicolor) Bit 4: Sprite #4 (1=Multicolor) Bit 5: Sprite #5 (1=Multicolor) Bit 6: Sprite #6 (1=Multicolor) Bit 7: Sprite #7 (1=Multicolor) |
| 53277 | Sprite Expansion Register (X direction) Bit 0: Sprite #0 (1=2X) Bit 1: Sprite #1 (1=2X) Bit 2: Sprite #2 (1=2X) Bit 3: Sprite #3 (1=2X) Bit 4: Sprite #4 (1=2X) Bit 5: Sprite #5 (1=2X) Bit 6: Sprite #6 (1=2X) Bit 7: Sprite #7 (1=2X) |
| 53278 | Sprite-Sprite Collision Register Bit 0: Sprite #0 (1=Collision Detect) Bit 1: Sprite #1 (1=Collision Detect) Bit 2: Sprite #2 (1=Collision Detect) Bit 3: Sprite #3 (1=Collision Detect) Bit 4: Sprite #4 (1=Collision Detect) |

Figure 5-6. C-128 VIC-II video chip registers (*continued*)

53248-54271

VIC-II Chip Control Registers

- Bit 5: Sprite #5 (1=Collision Detect)
- Bit 6: Sprite #6 (1=Collision Detect)
- Bit 7: Sprite #7 (1=Collision Detect)
- 53279 Sprite-Background Collision Register
 - Bit 0: Sprite #0 (1=Collision Detect)
 - Bit 1: Sprite #1 (1=Collision Detect)
 - Bit 2: Sprite #2 (1=Collision Detect)
 - Bit 3: Sprite #3 (1=Collision Detect)
 - Bit 4: Sprite #4 (1=Collision Detect)
 - Bit 5: Sprite #5 (1=Collision Detect)
 - Bit 6: Sprite #6 (1=Collision Detect)
 - Bit 7: Sprite #7 (1=Collision Detect)
- 53280 Screen Border Color
 - 0 = BLACK
 - 1 = WHITE
 - 2 = RED
 - 3 = CYAN
 - 4 = PURPLE
 - 5 = GREEN
 - 6 = BLUE
 - 7 = YELLOW
 - 8 = ORANGE
 - 9 = BROWN
 - 10 = LIGHT RED
 - 11 = DARK GREY
 - 12 = MEDIUM GREY
 - 13 = LIGHT GREEN
 - 14 = LIGHT BLUE
 - 15 = LIGHT GREY
- 53281 Background Color #1
 - 0 = BLACK
 - 1 = WHITE
 - 2 = RED

Figure 5-6. C-128 VIC-II video chip registers (*continued*)

53248-54271**VIC-II Chip Control Registers**

3 = CYAN
4 = PURPLE
5 = GREEN
6 = BLUE
7 = YELLOW
8 = ORANGE
9 = BROWN
10 = LIGHT RED
11 = DARK GREY
12 = MEDIUM GREY
13 = LIGHT GREEN
14 = LIGHT BLUE
15 = LIGHT GREY

53282**Background Color #2**

0 = BLACK
1 = WHITE
2 = RED
3 = CYAN
4 = PURPLE
5 = GREEN
6 = BLUE
7 = YELLOW
8 = ORANGE
9 = BROWN
10 = LIGHT RED
11 = DARK GREY
12 = MEDIUM GREY
13 = LIGHT GREEN
14 = LIGHT BLUE
15 = LIGHT GREY

53283**Background Color #2**

0 = BLACK
1 = WHITE

Figure 5-6. C-128 VIC-II video chip registers (*continued*)

53248-54271

VIC-II Chip Control Registers

- 2 = RED
- 3 = CYAN
- 4 = PURPLE
- 5 = GREEN
- 6 = BLUE
- 7 = YELLOW
- 8 = ORANGE
- 9 = BROWN
- 10 = LIGHT RED
- 11 = DARK GREY
- 12 = MEDIUM GREY
- 13 = LIGHT GREEN
- 14 = LIGHT BLUE
- 15 = LIGHT GREY

53284

Background Color #3

- 0 = BLACK
- 1 = WHITE
- 2 = RED
- 3 = CYAN
- 4 = PURPLE
- 5 = GREEN
- 6 = BLUE
- 7 = YELLOW
- 8 = ORANGE
- 9 = BROWN
- 10 = LIGHT RED
- 11 = DARK GREY
- 12 = MEDIUM GREY
- 13 = LIGHT GREEN
- 14 = LIGHT BLUE
- 15 = LIGHT GREY

53285

Sprite Multicolor Register #0

53286

Sprite Multicolor Register #1

Figure 5-6. C-128 VIC-II video chip registers (*continued*)

| 53248-54271 | <i>VIC-II Chip Control Registers</i> |
|--------------------|---|
| 53287-53294 | Sprite Color Registers 0-7 0 = BLACK 1 = WHITE 2 = RED 3 = CYAN 4 = PURPLE 5 = GREEN 6 = BLUE 7 = YELLOW 8 = ORANGE 9 = BROWN 10 = LIGHT RED 11 = DARK GREY 12 = MEDIUM GREY 13 = LIGHT GREEN 14 = LIGHT BLUE 15 = LIGHT GRAY |
| 53295 | Keyboard Control Register. Bits 7-3 unused. Bits 2,1, and 0 used as “extended” keyboard scan lines for the numeric keypad, etc. |
| 53296 | Clock speed register. Bits 7-2 are unused. Bit #1 engages a “test mode” used only for servicing. Bit #0 switches to 2 MHz clock speed, disabling the VIC-II display. |

Figure 5-6. C-128 VIC-II video chip registers (*continued*)

Direct writes will be useful only if the IRQ has been disabled via an SEI instruction or if GRAPHM, at \$D8, contains an \$FF. The split-screen and interrupt-driven graphics functions accessible from BASIC 7.0 will not function properly, though, but VIC-II programming becomes almost exactly the same as it was on the C-64.

| | | |
|---------|--------|--|
| GRAPHM, | \$D8 | (If GRAPHM contains \$FF, refreshing of VIC-II registers is disabled.) Bit #7—selects multicolor mode graphics. Bit #6—selects split-screen display (text with graphics). Bit #5—selects ordinary (monochrome) bit-mapped graphics. |
| CURMOD | \$0A2B | Contains current 8563 cursor mode (none, solid, slow blinking, fast blinking). |
| VMI | \$0A2C | Functions the same as the VIC-II register at \$D018. Contains the base address of the character text matrix and character base pointer. Applies to text mode only. |
| VM2 | \$0A2D | Functions the same as the VIC-II register at \$D018. Contains pointer to color matrix and 8K bitmap. Applies to graphics mode only. |

Figure 5-7. Important Shadow Registers used by VIC-II and 8563

Additionally, the 8563 uses shadow registers for pointers to the start of text screen storage, attribute storage, and the current cursor mode. When necessary, these “VDC” shadow registers should be updated, instead of writing to the actual 8563 registers.

A “well-behaved” C-128 program writes to the RAM-resident shadow registers instead of the VIC-II itself. Figure 5-7 lists these shadow registers and their functions.

Chapter 6

The C-128 Kernal: An Overview

The operating system of the Commodore 128's "native" mode is contained in 16K of ROM, located at \$E000 through \$FFFF. This "Kernal" contains standard routines used for file handling, generalized I/O, device selection, serial bus I/O, and so on. The C-128's Kernal also contains memory management routines that allow the user to select banks of memory (or I/O devices) as well as to perform cross-bank loads, stores, comparisons, or subroutine calls.

The overall structure of the C-128 Kernal is much the same as that of the C-64, VIC-20, or even the old PET. The top area of the Kernal ROM — page \$FF — contains what is known as a *jump table*.

This is merely a table of JMP \$XXXXs, one entry per Kernal function, that enforces some degree of software compatibility among various machines. If Commodore decides to revise some ROM routines in the Kernal, the programmer shouldn't have to change the code whatsoever; the Kernal's jump table presents a uniform software interface that allows programmers to isolate themselves from low-level hardware details. For example, the general character output routine can be called with JSR \$FFD2. This is true for all 6502-based Commodore machines, from the old PET to the C-128. Even if Commodore decides to modify this routine in future revisions of the C-128 ROM software, the programmer still will call the routine with JSR \$FFD2.

The Kernal jump tables of the C-64 and C-128 are, for the most part, the same. Thus, programs that exclusively use Kernal calls—and that avoid direct jumps to ROM routines—make rewriting programs for new machines much easier. In fact, Commodore doesn't recommend bypassing this jump table and directly calling Kernal ROM routines.

Commodore's Kernal corresponds roughly to the BIOS, and some parts of the BDOS, of CP/M. (See Chapter 7 for CP/M coverage.) Unlike other operating systems, however, the Kernal in your C-128 is not a DOS (Disk Operating System). It contains no routines that actually handle disk drive operations itself. Instead, it passes commands over a special serial bus to an intelligent controller within the disk drive, which handles all the low-level details of disk file handling, directory structuring, and so forth. The BASIC 7.0 disk commands do give the appearance of a real C-128 disk operating system, but these keywords merely cause commands and filenames to be sent to the 1571 drive (or the 1541, 1540, and so on), which actually processes them.

Earlier Commodore computers have their screen editor routines contained within the Kernal ROM, and these routines can, in a way, actually be considered a part of a microcomputer's operating system. The editor routines did not, however, contain respective entries in the

Kernal jump table. Because of some of the extra functions in the C-128's operating system, the C-128's Kernal has increased in size compared with its predecessor in the C-64. The screen editor routines, handling both 80- and 40-column screens, now occupy the ROM space at \$C000 to \$CFFF. Commodore has finally seen fit to give the editor routines their own jump table, which occupies \$C000 to \$C02D in the ROM bank. Useful screen editor routines are listed in Chapter 3.

Kernal routines are normally used while in the *system bank*, bank \$0F(15). Thus, all ROMs (except the character ROM) and I/O chips are enabled. Additionally, RAM bank 0 occupies the available RAM space in the system bank. Kernal routines can also be called with the JSRFAR (or sometimes JMPFAR) routine.

Vectors

Some Kernal and editor—and even BASIC 7.0—routines are *vector*ed through RAM locations. These two-byte vectors, in standard 6502 low-byte/high-byte form, point to the actual code of the Kernal routine and are located in a table beginning at \$02FC. They are initialized upon power-up with their standard values by the Kernal routine RESTOR, which can be called with a JSR \$FF8A (65418 decimal). RAM vectors are especially useful because the programmer can point them at his or her own code, resulting in either complete replacement or modification of the Kernal routine.

It should be noted that the Kernal jump table is really a table of fixed vectors, since they are in ROM. For example, the CHROUT (also called BSOUT) routine's entry in the Kernal jump table, \$FFD2, passes control to a RAM vector located at \$0326 to \$0327. This vector, called IBASOUT, can be redirected to point at the programmer's own character output routine. This routine might, for example, convert Commodore's PetASCII characters to standard ASCII before sending them to an output device.

Unfortunately, because of possible future revisions of the Kernal ROM code, Commodore may change the addresses pointed to by these vectors. If the user's routine merely pre-processes data and then actually jumps to a fixed location in ROM that contains the Kernal routine's code, the user's code may not function on later versions of the C-128. Because of this, it is advisable to preserve the original vector contents in an unused part of RAM before diverting the vector to the user's code. Once this user-written code has been executed, the routine should jump "through" the *copy* of the original vector (using the 6502's jump indirect, `JMP $(xxxx)`) so that the execution of the Kernal code is guaranteed. Of course, direct jumps are sometimes unavoidable. If the user's routine replaces the "front end" of a vectored Kernal routine, for example, the original "front end" of the Kernal routine should obviously not be executed also. Thus, a direct jump to the middle of the Kernal routine will be necessary. These direct ROM jumps are also unavoidable if the Kernal routine does not have its own RAM vector.

Kernal Dictionary

The dictionary of C-128 Kernal calls presented here lists the calling address, name, and function of each of the C-128 Kernal routines addressed by the jump table. With the exception of those routines marked by a plus sign (+) or asterisk (*), every entry in this table pertains to all past implementations of the Kernal in older machines.

The entries marked with an asterisk before their calling addresses are specific to the C-128 and do not have any C-64 equivalent. An entry marked with a plus sign prior to its calling address is similar to its C-64 counterpart but possesses some significant differences. For these instances, the section following this table details each Kernal function that is either unique to the C-128 or is different than its previous C-64 implementation. The user should refer to C-64 documentation for information on Kernal calls whose uses and functions are unchanged on the C-128.

The comments in the “Mnemonics” and “Descriptions and Comments” columns of Table 6-1 list not only the standard Commodore name for these routines but also any “dangerous” conditions or other special details that should be observed. Some mnemonics of a few Kernal routines have been changed from their original PET/CBM ones. For example, the \$FFD2 character output routine is called CHROUT in C-64 documentation but BSOUT in C-128 manuals; similarly, another C-64 Kernal routine called UNTALK has been renamed UNTLK in C-128 documents. Both incarnations perform (outwardly) the same way. Additionally, this section lists appropriate RAM vectors of Kernal routines, where necessary.

Table 6-1. Kernal Dictionary

| <i>Jump Table Address</i> | <i>Mnemonics</i> | <i>Descriptions and Comments</i> |
|---------------------------|------------------|---|
| * \$FF47 | SPIN__SPOUT. | This sets up the serial bus to support the new fast serial bus protocol for the 1571 drives. |
| * \$FF4A | CLOSE__ALL. | Closes all files on a device. Unlike CLALL, it not only deletes the file parameters from the logical file table, but also sends an “untalk” signal over the serial bus, if in use. Avoids creating a “splat” disk file by actually closing it on the drive as well as in the C-128. |
| * \$FF4D | C64MODE. | Switches from C-128 “native” mode to C-64 mode. Once in C-64 mode, there’s no way to return to C-128 mode without using the reset switch. |

Table 6-1. Kernal Dictionary (*continued*)

| <i>Jump Table Address</i> | <i>Mnemonics</i> | <i>Descriptions and Comments</i> |
|---------------------------|------------------|--|
| * \$FF50 | DMA_CALL. | Sends command to direct memory access device, such as a RAM expansion cartridge. |
| * \$FF53 | BOOT_CALL. | Reads boot sector (track 1, sector 0) from disk. Can load a series of sectors into RAM. |
| * \$FF56 | PHOENIX. | Prepares for use of external (internal) function cartridges (ROM) for special ID characters. |
| * \$FF59 | LKUPLA. | Finds device number and secondary address for a given logical address, if in use. |
| * \$FF5C | LKUPSA. | Finds logical file and device numbers for a given secondary address, if in use. |
| * \$FF5F | SWAPPER. | 80- or 40-column video output toggling. |
| * \$FF62 | DLCHR. | Downloads ROM character set to 8563's RAM. |
| * \$FF65 | PFKEY. | Programs C-128 function key's output string. |
| * \$FF68 | SETBNK. | Selects bank that contains filename for SETNAM, and the RAM bank of the data for LOAD/SAVE/VERIFY. |
| * \$FF6B | GETCFG. | Given a (logical) bank number, will look up corresponding value for MMU configuration registers. |

Table 6-1. Kernal Dictionary (*continued*)

| <i>Jump Table Address</i> | <i>Mnemonics</i> | <i>Descriptions and Comments</i> |
|---------------------------|------------------|--|
| * \$FF6E | JSRFAR. | Cross-bank JSR that calls subroutine in another bank. Upon RTS, execution resumes in code in original bank. |
| * \$FF71 | JMPFAR. | Cross-bank JMP that causes execution of code in another bank. |
| * \$FF74 | INDFET. | Cross-bank LDA (pointer), Y. The byte to be fetched may be in any bank. |
| * \$FF77 | INDSTA. | Cross-bank STA (pointer), Y. The byte in .A can be stored in any bank having RAM or I/O. |
| * \$FF7A | INDCMP. | Cross-bank CMP (pointer), Y. The byte that .A is tested against may be in any bank. |
| * \$FF7D | PRIMM. | Print immediately. This routine prints to the default output device a string less than 255 characters long. The string must have a \$00 terminator byte. |
| + \$FF81 | CINT. | Initializes screen editor, etc. |
| + \$FF84 | IOINIT. | Initializes I/O devices. |
| + \$FF87 | RAMTAS. | Initializes RAM and buffers, etc. |
| \$FF8A | RESTOR. | Initializes Kernal RAM vectors; will not affect editor or BASIC 7.0 RAM vectors. |

Table 6-1. Kernal Dictionary (*continued*)

| <i>Jump Table Address</i> | <i>Mnemonics</i> | <i>Descriptions and Comments</i> |
|---------------------------|------------------|---|
| \$FF8D | VECTOR. | Copies Kernal RAM vectors to another location, or loads Kernal RAM vectors from a user-created table. Status of 8502's .C (carry) bit determines load or copy function. |
| \$FF90 | SETMSG. | Turns Kernal's control and error messages on or off. |
| \$FF93 | SECND. | Sends secondary address (after a LISTEN command) over the serial bus. |
| \$FF96 | TKSA. | Sends secondary address over serial bus after a TALK command has been issued. |
| + \$FF99 | MEMTOP. | Sets or reads top of system RAM. (Bank 0 only) |
| + \$FF9C | MEMBOT. | Sets or reads bottom of system RAM. (Bank 0 only) |
| \$FF9F | KEY. | Scans whole C-128 keyboard. Called by IRQ interrupt. |
| \$FFA2 | SETTMO. | Normally unused in C-128, but disables I/O "time-outs" for an IEEE interface cartridge. |
| \$FFA5 | ACPTR. | Gets byte from serial port. |
| \$FFA8 | CIOUT. | Sends byte out of serial port. |
| \$FFAB | UNTLK. | Sends an UNTALK command over serial bus. |
| \$FFAE | UNLSN. | Sends an UNLISTEN command over serial bus. |
| \$FFB1 | LISTN. | Sends LISTEN command over serial bus. |

Table 6-1. Kernal Dictionary (*continued*)

| <i>Jump Table Address</i> | <i>Mnemonics</i> | <i>Descriptions and Comments</i> |
|---------------------------|------------------|--|
| \$FFB4 | TALK. | Sends TALK command over serial bus. |
| \$FFB7 | READSS. | Reads I/O status byte. Called READST in C-64 documentation. |
| \$FFBA | SETLFS. | Sets up logical file number, device number, and secondary address for an I/O channel. |
| + \$FFBD | SETNAM. | Sets up filename length and address. |
| \$FFC0 | OPEN. | Opens a logical file. RAM vector (IOPEN) at \$031A. |
| + \$FFC3 | CLOSE. | Closes a logical file. RAM vector (ICLOSE) at \$031C. |
| \$FFC6 | CHKIN. | Sets current input channel. RAM vector (ICHKIN) at \$031E. |
| \$FFC9 | CKOUT. | Sets current output channel. RAM vector (ICKOUT) at \$0320. |
| \$FFCC | CLRCH. | Restores default I/O channels. RAM vector (ICLRCH) at \$0322. |
| \$FFCF | BASIN. | Gets a character from current input channel. RAM vector (IBASIN) at \$0324. BASIN is called CHRIN in C-64 documentation. |
| \$FFD2 | BSOUT. | Outputs a character through current output channel. RAM vector (IBSOUT) at \$0326. |

Table 6-1. Kernal Dictionary (*continued*)

| <i>Jump Table Address</i> | <i>Mnemonics</i> | <i>Descriptions and Comments</i> |
|---------------------------|------------------|---|
| \$FFD5 | LOAD. | Loads RAM from file on cassette or disk. RAM vector (ILOAD) at \$0330. |
| \$FFD8 | SAVE. | Saves RAM to file on cassette or disk. RAM vector (ILOAD) at \$0332. |
| \$FFDB | SETTIM. | Sets "jiffie" (1/60 sec.) software clock. |
| \$FFDE | RDTIM. | Reads "jiffie" clock. |
| \$FFE1 | STOP. | Scans RUN-STOP key. RAM vector (ISTOP) at \$0328. |
| \$FFE4 | GETIN. | Gets data from an input buffer. RAM vector (IGETIN) at \$032A. |
| \$FFE7 | CLALL. | Closes files and channels by removing entries from logical file table. RAM vector (ICLALL) at \$032C. |
| \$FFEA | UDTIM. | Updates system "jiffie" clock every 1/60 second. |
| + \$FFED | SCRORG. | Gets current screen window size. Called SCREEN in C-64 documentation. |

Modified Kernal Routines

The Kernal routines flagged by a plus sign in the table just presented, although similar to their brethren in the standard PET/CBM/C-64 Kernal, differ slightly (but significantly) from their previous implementations. The following text describes these differences.

CINT (\$FF81)

The CINT (\$FF81) routine initializes the screen editor. In the C-128 it is also callable at \$C000, since CINT is part of the editor's jump table, too. The CINT routine, like that of the C-64, accomplishes the task of initializing the 40-column editor's variables and the VIC-II chip's registers. CINT on the C-128 also initializes the 80-column screen variables and the 8563 chip's registers. The ROM character definitions are copied to the 8563's RAM, too. CINT also pre-defines the function keys and sets up vectors. Before calling CINT, interrupts should be disabled because some editor IRQ vectors will be set up. Also, a full initialization of editor and other system variables can be avoided by setting bit #6 high of location \$0A04 (INIT__STATUS, at 2564 decimal, in RAM bank 0). When this same bit is low, a full-screen editor/video initialization is allowed. This avoidance of resetting the editor is particularly useful during a debugging session after a program crashes.

IOINIT (\$FF84)

IOINIT is the Kernal routine responsible for initializing the C-128's I/O hardware and is one of the major routines called when a system reset occurs. First it sets up registers in both 6526 CIA chips (at \$DC00 and \$DD00) so that the timers, serial port, RS-232, cassette port, and keyboard are initialized. IOINIT also sets up the 8502 port at locations \$00 and \$01 (D6510 and R6510), and determines whether the system is to generate NTSC-style (National Television Standards Committee) or PAL-style (Phase-Alternating Line) video; the system variable PALCNT, at \$0A03, is updated afterward. (PALCNT is sometimes called PALNTS.)

At this point, the SID sound chip and the VIC-II and 8563 video chips are initialized. The INIT__STATUS system variable at \$0A04 determines whether or not the character set will be downloaded to the 8563 (80-column) chip's independent 16K of RAM. If INIT__STATUS' bit #7 is set, only a partial initialization occurs, because

no character definitions will be downloaded to 8563 RAM. This is quite useful; a custom character set will not be overwritten if the system is reset.

Finally, the normal 60-Hz IRQ source is initialized and any previous interrupts are cleared. Before calling IOINIT, interrupts should be disabled so as not to disturb I/O devices being initialized; interrupts can be reenabled after calling IOINIT by using a CLI instruction. IOINIT does not require any parameters prior to calling, and it changes the contents of the .A, .X, and .Y registers upon return.

RAMTAS (\$FF87)

RAMTAS on a C-64 both tests RAM and determines the amount of free RAM available in order to set up several pointers. Unlike its C-64 counterpart, however, RAMTAS does not test memory. The C-64 and C-128 versions of RAMTAS also clear zero-page RAM and allocate the RS-232 and cassette buffers. Additionally, RAMTAS on the C-128 sets the SYSTEM-VECTOR to BASIC 7.0's cold-start location, \$4000, and sets the DEJA-VU flag (\$0A02, decimal 2562 in RAM bank 0) to indicate that RAM has been initialized. When \$0A02 contains \$A5 (165 decimal), the C-128 does not complete a full reset sequence when the reset switch is pressed; instead, RAMTAS and most other initialization routines are bypassed. Like the use of the INIT__STATUS variable, DEJA-VU makes life a little bit easier while you are debugging a crashed (or soon-to-be-crashed) program.

MEMTOP (\$FF99) and MEMBOT (\$FF9C)

MEMTOP and MEMBOT have been implemented in the C-128's Kernal merely for the sake of compatibility. They are called in the same way that the C-64's MEMTOP and MEMBOT are, but they really have little significance in the C-128 "native mode" environment because of its banked memory and fixed buffer locations. However,

MEMTOP does set the pointer MEMSIZ (at \$0A07) to \$FF00, while MEMBOT sets the pointer MEMSTR (at \$0A05) to \$1C00. Since neither the Kernal nor BASIC 7.0 refers to these locations, they are quite innocuous. When the carry bit is clear, MEMTOP and MEMBOT *set* the upper and lower bounds of system RAM. The 8502's .X and .Y registers contain the least- and most-significant byte (respectively) to be loaded into MEMSIZ or MEMSTR. If the carry bit is set, the MEMTOP and MEMBOT routines will read the current top and bottom of system RAM into .X and .Y. Both these routines only refer to bank #0.

SETNAM (\$FFBD)

SETNAM on the C-128 performs exactly the same way as SETNAM in C-64 mode does—the 8502's .X and .Y registers are loaded with the low and high bytes (respectively) of the filename's starting address, while .A should contain the string length. However, string data on the C-128 is customarily stored in RAM bank #1 by the BASIC 7.0 interpreter. Thus, SETBNK must be called to select the proper RAM bank that contains the filename string. See the next section for information on SETBNK.

CLOSE (\$FFC3)

CLOSE on the C-128 functions almost exactly the same as its counterpart in the C-64 Kernal. However, a full CLOSE is not performed when

- The 8502's carry bit is set.
- The device number indicates a disk drive (the device number is between 8 and 30, inclusive).
- The secondary address is 15 (disk drive command channel). This special case allows the disk drive's command channel to

be effectively OPENed and CLOSEd without closing any other open files on the device; it merely deletes the file's entry from the table of open files.

SCRORG (\$FFED)

SCRORG on the C-128 corresponds to the SCREEN call in the C-64's Kernal. It can also be accessed from the editor's jump table by calling \$C00F. Whereas SCREEN on the C-64 returns the 40×25 screen size in the .X and .Y registers, respectively, SCRORG on the C-128 returns the size of the current window in use. The window's width is returned in .X, while .Y contains the window's height. The maximum screen size (80 or 40) is returned to .A, depending upon which screen editor is in use. Since SCRORG is really an editor routine, its entry can also be found at \$C00F in the editor jump table.

PLOT (\$FFF0)

Like SCRORG/SCREEN, the Kernal PLOT routine retains its overall character while having been modified for a windowing environment. PLOT now does not use absolute row/column screen addressing, but instead addresses a given screen location relative to a window position. There are several zero-page memory locations that control the cursor position within a window; PLOT can be considered an editor routine in disguise, and PLOT can be called—as an editor routine—at \$C018.

New C-128 Kernal Routines

The C-128 has 19 added Kernal calls. Several of these concern memory management and cross-bank operations. Others deal with disk I/O, while a few more can be considered “utilities.” This section

will detail the operation, parameters, and other minutiae concerning these additional Kernal calls.

SPIN_SPOUT (\$FF47)

SPIN_SPOUT readies the fast serial port for I/O. By setting the 8502's carry bit before calling, SPOUT is selected. SPIN is selected by ensuring that the carry bit is clear before calling \$FF47. The FSDIR bit of the MMU's register 5 (\$D505, bit #3) is zeroed for input and set high for output. CIA #1 (6526) registers 4, 5, 13, and 14 are also initialized. Users don't really need to concern themselves with SPIN_SPOUT; it's a very low-level routine in the same category as ACPTR, UNTLK, TKSA, and so on. Only programmers dealing with the serial but "primitives" need concern themselves about SPIN_SPOUT. No register setup is required prior to calling this routine; on return, only .A will be used.

CLOSE_ALL (\$FF4A)

CLOSE_ALL closes all files on a device. Load into .A the device number of the "unit" needed to be closed. After calling CLOSE_ALL, all files or channels on that device will be closed. If the current I/O channel (such as redirected screen output) is being closed on the device, the default I/O channels will be restored. CLOSE_ALL is the machine-code equivalent of BASIC 7.0's DCLOSE command. After the CLOSE_ALL code has been executed, the previous contents of the 8502's .A, .X, and .Y registers will be destroyed.

C64MODE (\$FF4D)

C64MODE causes the C-128, in "native" mode, to switch over to C-64 mode. This routine selects the C-64 BASIC 2.0 and Kernal ROMs, sets locations 0 and 1 (D6510 and R6510) properly, and then resets the

system. There is no way to exit C-64 mode on the C-128, because the MMU chip is not present in the C-64 memory map. However, the 8563 80-column video chip is accessible, as well as 2-MHz processing and access to the full C-128 keyboard. For more information, please consult Part I of this book. The C64MODE routine does not require any parameters.

DMA _ CALL (\$FF50)

The DMA _ CALL Kernal routine is used to send a command to a direct memory access device, such as Commodore's 1700-series RAM expansion cartridges for the C-128. First, necessary information should be sent to the proper DMA chip registers that begin at \$DF00. (These registers are not present when a RAM expansion cartridge is not in place!) Prior to calling \$FF50, the 8502's .X register should contain the proper bank, #0 through #15, while the .Y register should contain the number of the DMA controller command. On return, registers .X and .Y are used. After the MMU has been properly set up, DMA _ CALL uses a RAM-based routine located at \$03F0 to actually issue the command to the DMA device. During the DMA transfer, the 8502 is held in a "wait state." After the transfer is complete, control is relinquished to the 8502 processor and system memory is configured the way it was at the time a call was issued to \$FF50. The programmer's code should check the DMA status by reading \$DF00 once the DMA activity has been completed and act appropriately. See the instruction manual for the RAM expansion cartridge for more details.

Current versions of the C-128's Kernal ROM contain a bug that can corrupt data within I/O registers, possibly crashing or locking up the system. This bug also affects the BASIC 7.0 statements FETCH, STASH, and SWAP. The \$D000-\$DFFF I/O block is active during any DMA transfer, so the programmer must either create a custom DMA routine to replace the Kernal's or to restrict all DMA transfers

to memory regions outside the 4K I/O block. Thus, before using these BASIC memory management commands, the programmer should limit the amount of memory transferred appropriately. Since essential code for DMA transfers is contained in common RAM at \$03F0, you can avoid problems by bypassing DMA —CALL like this:

```

      .
      .
      LDX #$00 ; select bank #0 (i.e., no I/O banked in).
      JSR $FF6B ; get MMU configuration data.
      TAX      ; and put in .X register.
      LDY #$84 ; DMA device's "stash" command
      JSR $03F0 ; execute a DMA command.
      .
      .
      .

```

BOOT_CALL (\$FF53)

BOOT_CALL tries to load and execute the boot sectors from a disk. Before calling this routine, .A should contain the ASCII value of the drive number (0=\$30, 1=\$31, and so on) while .X should contain the actual device number itself (8, 9, 10, and so on). On return, .A, .X, and .Y will be used. The carry bit can be tested for an error; if set, control can pass to an error-handling routine. BOOT_CALL can load a sequence of sectors, if specified in the header bytes of the boot sector itself—track 1, sector 0. Figure 6-1 shows the layout of the boot header. If byte 6 is larger than zero, a sequential read of that number of sectors into RAM will occur, beginning at the address pointed to by bytes 3 and 4, in the bank number specified by byte 5. If the filename exists, it will be loaded at its start address, while “booting <title>” is displayed on the screen. After this file load is completed, a jump (JSR) is issued to the ML code following the filename. More details on the autoboot procedure, with some autoboot examples, can be found in Chapter 7.

| | | | | | | |
|-----------|---------|---------------------|-----|-----------|-------|-----------------|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| "C" | "B" | "M" | adL | adH | bank# | #sectors |
| 7 ... n | $(n+1)$ | $(n+2) \dots (n+x)$ | | $(n+x+1)$ | | $(n+x+2) \dots$ |
| (title) | \$00 | (filename) | | \$00 | | (ML code) |

Figure 6-1. Track 1, sector 0 boot-sector layout where n =title length, and x =filename length. adH and adL are the low and high bytes of the load address; bank# is the desired RAM bank to receive the data.

PHOENIX (\$FF56)

PHOENIX is called during Kernal initialization. The internal (not available to the jump table) Kernal routine POLL searches for C-128 cartridges and stores their IDs in a *physical address table*. PHOENIX then calls the cartridges via their cold-start routines in the sequence: external low, external high, internal low, internal high. After calling all these cartridges, PHOENIX also searches for an autoboot disk in drive 0 of device #8 by calling BOOT_CALL. PHOENIX returns (if control has not passed, say, to some autobooted routine) with the 8502's .A, .X, and .Y registers changed.

LKUPLA (\$FF59) and LKUPSA (\$FF5C)

LKUPLA and LKUPSA are Kernal *utility* routines. They obtain the file parameters for a given logical or secondary address. The Kernal uses these two routines to establish unique logical addresses, while the disk drive requires unique secondary addresses (*channels*). Whenever BASIC opens a disk file, it must "work around" any currently open files. To use LKUPLA, load .A with the logical file number to be

searched for, and then call \$FF59. To use LKUPSA, load .Y with the desired secondary address to be searched for and then call \$FF5C. For either routine, the .C (carry) bit will be set if the entry in the open-file table is not found. Otherwise, the carry bit is clear if no entry is found. If an entry is found, the 8502's .A, .X, and .Y registers will contain the logical address, device number, and secondary address, respectively.

LKUPLA and LKUPSA are usually used in a loop—the program might have to run through several logical (or secondary) addresses already in use before it finds a free one.

SWAPPER (\$FF5F)

SWAPPER is also available at \$C02A through the editor's jump table at \$C02A. SWAPPER toggles between 80- and 40-column video output. It merely exchanges current local editor variables (such as TAB counts and screen link bytes) with those of the other screen editor. The high bit (bit #7) of location \$D7, MODE, indicates the current screen display mode. (Do not confuse these modes with the GRAPHIC modes of BASIC 7.0.) Bit #7 of MODE is high if the 80-column editor is active. SWAPPER does not require any input parameters; the .A, .X, and .Y registers are changed upon return from SWAPPER.

DLCHR (\$FF62)

DLCHR is also available as INIT80 at \$C027 through the editor's jump table. This routine copies the ROM-based (\$D000-\$DFFF in bank #14) character set to the 8563's independent 16K RAM. Character definitions in the 8563's own RAM are stored at \$2000 to \$3FFF, with eight \$00 bytes of "padding" between each 8563 character definition. DLCHR requires no register setup, and the .A, .X, and .Y registers are used. For details concerning character definitions and alternate character sets, please consult Chapter 5.

PFKEY (\$FF65)

PFKEY is also available as KEYSET at \$C021 within the editor's jump table. This routine determines the string to be output whenever a function key is pressed. Note that function keys F1 through F8 are assigned indexes \$01 through \$08, while the SHIFT/RUN-STOP combination is assigned \$09. HELP is assigned \$0A. To define a function key's output string, create both a string and a zero-page pointer to the string. The pointer should be three bytes long; the first two bytes point to the string address in standard 6502 low-byte/high-byte form, while the third byte indicates the bank number in which the string is contained. Load the 8502's .A with the address of this pointer. Load .Y with the output string's length, and .X with the key number to be assigned the output string. PFKEY does change the contents of the .A, .X, and .Y registers. After calling PFKEY or KEYSET, the 8502 carry bit will be clear if the definition was successful. If the carry bit is set, this means no room is left for the output string, and no change will be made to the key definition table. There are only 246 bytes available for output string storage. The table of output strings, PKYDEF, is located at \$100A to \$10FF. Current output string lengths are stored in a table called PKYBUF at \$1000 to \$1009. Figure 6-2 shows how to assign the machine language monitor's directory-list command to the F1 key.

SETBNK (\$FF68)

SETBNK selects the bank that contains the filename for the SETNAM routine. Since SETNAM is a prerequisite for opening a file, SETBNK is also required. This routine also sets the bank number (0-15) for LOAD, SAVE, and VERIFY operations. To use SETBNK, load .A with the number of the bank you wish all LOADs, SAVEs, and so on to occur. The .X register should contain the number of the bank that contains the filename for SETLFS. The 8502's register contents are unchanged upon exit from SETBNK. Remember to add calls to SETBNK when porting file-handling routines from the C-64 to the C-128.

```

>1800 44 49 52 45 43 54 4F 52 59 0D
>00FC 00 18 00 : Points to "DIRECTORY" stringat $1800 in Bank 0.

      .
      .
      .
      LDX #$00
      STX $FF00 ; Select bank #15 (ROMs banked in)
      LDA $FC ; .A contains zero-page string pointer address.
      LDX #$01 ; Key number for <F1> key.
      LDY #$0A ; String 10 characters long-including <CR>.
      JSR $FF65 ; Call PFKEY (also KEYSET at $C021).
      .
      .
      .

```

Figure 6-2. Example of programming a function key (F1 key defined with the DIRECTORY command)

GETCFG (\$FF6B)

GETCFG reads the proper MMU set-up data byte from a look-up table, using the bank number as an index. For example, given bank #15, GETCFG returns a zero. To use GETCFG, load the .X register with the logical bank number and call GETCFG. On return, the 8502's accumulator will contain the proper value to be stored to the MMU chip's configuration register or preconfiguration register. Figure 6-3 shows an example of selecting bank #15.

JSRFAR (\$FF6E) and JMPFAR (\$FF71)

JSRFAR and JMPFAR allow subroutines to be called or "jumped to" in another bank. These routines are really resident in "common" RAM; JSRFAR can be called from any bank with JSR \$02CD, while JMPFAR can be called from any bank with JSR \$02E3. To use either routine, the programmer will have to store parameters in low zero-page memory. Location \$02 should be loaded with the destination

```
LDA #$0F      ; Select bank #15.  
JSR $FF6B    ; Call GETCFG to get MMU data.  
STA $FF00    ; Store to MMU configuration reg.
```

Figure 6-3. Bank selection using GETCFG

bank number, while locations \$03 and \$04 should contain the desired destination address in nonstandard high-byte/low-byte format. Location \$05 should contain the current processor status, while locations \$06, \$07, and \$08 should contain the register values .A, .X, and .Y will assume immediately prior to execution of the subroutine.

Locations \$05 through \$08 will contain the returned contents of the .S, .A, .X, and .Y registers (respectively) upon completion of the called subroutine and return from JSRFAR. (This last condition obviously doesn't apply to JMPFAR, since there is no return.) Although very handy, these routines consume many clock cycles. Try to avoid using them if you can manipulate some MMU registers to change banks and properly call the desired routine. Also, remember that calling a routine in a bank without system ROM may result in problems if interrupts occur; it is often good practice to disable interrupts or to be able to handle them if they occur. Figure 6-4 shows how to call a routine in another bank using JSRFAR.

INDFET (\$FF74) and INDSTA (\$FF77)

INDFET and INDSTA allow cross-bank LDA and STA operations, respectively. These routines also support indirect indexing. Like JSRFAR and JMPFAR, these routines do require some setup before use. Both routines require allocating a zero-page pointer before use, also. Before calling INDFET, .A should contain the pointer to the

```

STA $06           ; Preserve .A, .X, .Y contents before call.
STX $07
STY $08
PHP              ; Push 8502 status on stack.
PLA              ; Bring processor status into .A.
STA $05           ; Save 8502 status for restoration after JSRFAR call.
                  ; We'll call $AF30 in the BASIC 7.0 ROM.
                  ; Select bank #15.
LDA #$0F         ; Point to $AF30 and set up JSRFAR pointers.
LDX #$AF
LDY #$30
STA $02
STX $03
STY $04
JSR $FF6E        ; Call JSRFAR.
LDA $05           ; Get old status.
PHA              ; Push on stack.
LDA $06           ; Restore original .A, .X, .Y contents before call.
LDX $07
LDY $08
PLP              ; Restore original processor status before call.
.
.
.
.

```

Figure 6-4. Using JSRFAR to call an out-of-bank routine

address, while .X should contain the desired bank number. The 8502's .Y should contain an index value; if none is required, set .Y to zero. INDSTA requires that .A contain the data byte to be stored. The .X register should contain the desired bank number; load .Y with the appropriate index value (zero, if no index value is needed). INDSTA also requires the pointer's address to be stored in \$02B9, STAVEC. Both routines return with the .X register unchanged. Of course, INDFET returns data in .A, unlike INDSTA. Figure 6-5 shows how to use INDFET and INDSTA.

INDCMP (\$FF7A)

INDCMP executes a cross-bank comparison (CMP). It can compare .A with a byte in any bank. Before calling, .A should contain the data

```

INDFET
    LDA #$80      ; Point fetch address at $8000.
    LDX #$00      ;
    STY $FD       ; Set up a zero-page pointer.
    STX $FC
    LDA #$FC      ; .A contains zero-page pointer address.
    LDY #$00      ; No indexing; would be non-zero if an
                  ; index or offset were needed.
    LDX #$0F      ; Fetch from bank #15.
    JSR $FF74     ; Call Kernal's INDFET routine.

    .             ;.A now loaded with contents of $8000in bank #15.
    .
    .             ; Equivalent of LDA (pointer, bank#), index.
-----
INDSTA
    LDY #$80      ; Point to store address at $8000.
    LDX #$00      ;
    STY $FD       ; Set up zero-page pointer.
    STX $FC
    LDA #$FC      ; .A contains zero-page pointer address.
    STA $02B9     ; STAVEC contains zero-page pointer address.
    LDA "byte"    ; Byte to be stored.
    LDX #$02      ; Select bank #2.
    LDY #$00      ; No indexing needed.
    JSR $FF77     ; Call INDSTA. $8000 in bank #2 now contains "byte".
    .
    .             ; Equivalent of STA (pointer, bank), index.
    .
    .

```

Figure 6-5. Examples of INDSTA and INDFET

to be compared with, .X should contain the bank number, and .Y should contain the index. Zero the .Y register if no indexing is needed. Pointer setup is also required; the address of the zero-page pointer should be stored to CMPVEC at \$02C8. INDCMP changes the value in the .X register. Figure 6-6 shows how to use INDCMP.

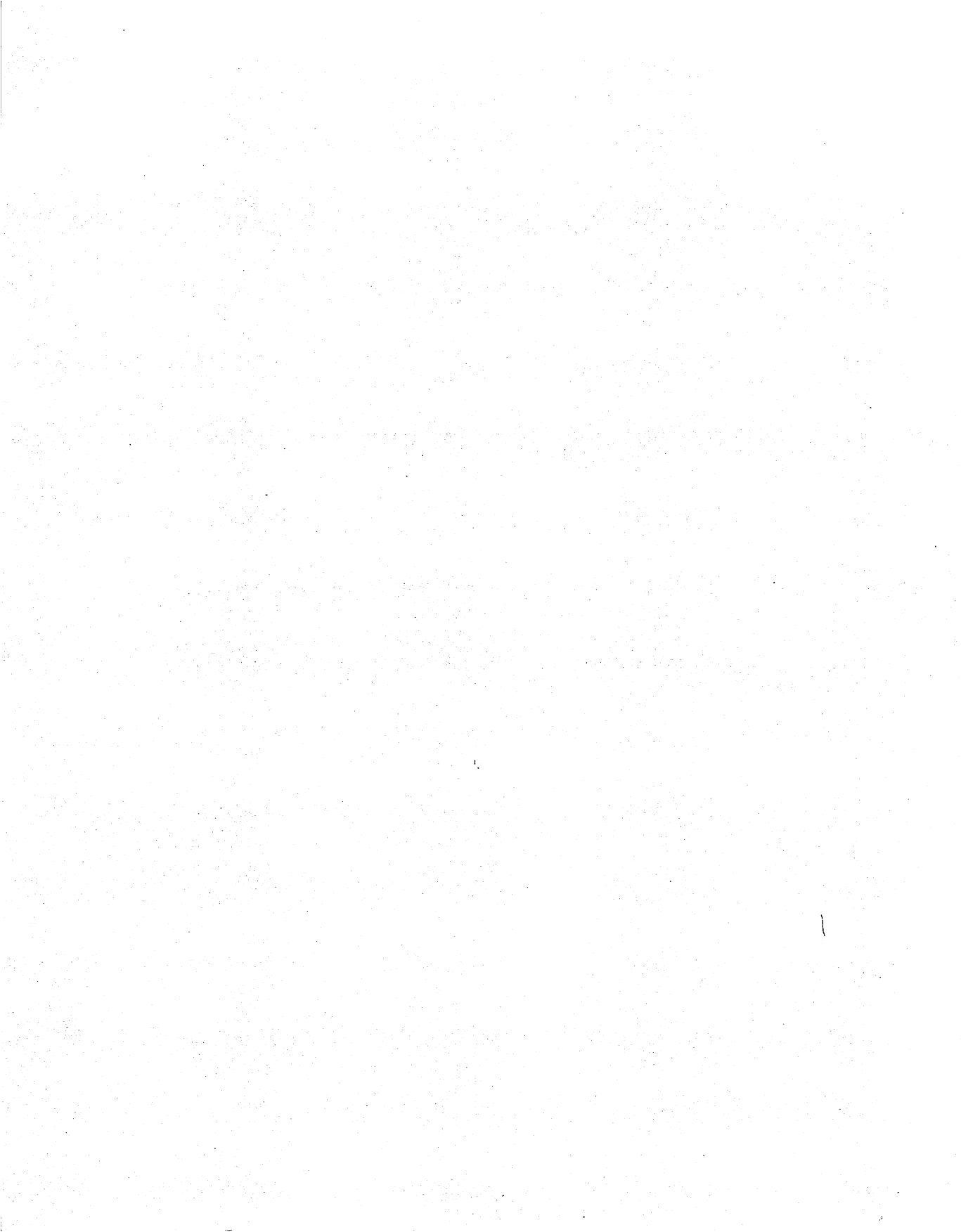
PRIMM (\$FF7D)

PRIMM stands for "Print Immediate" and is quite a useful utility routine. PRIMM prints, to the current output device, the string

```
LDY #$80      ; Point to $8000.
LDX #$00
STY $FD      ; Set up a zero-page pointer.
STX $FC      ;
LDA #$FC      ; .A contains pointer address.
STA $02C8    ; Vector address stored for INDCMP's usage.
LDA "byte"   ; Byte to be compared with.
LDX #$0F     ; Bank #15.
LDY #$00     ; No indexing necessary.
JSR $FF7A    ; Call INDCMP.
BEQ .....   ; Is .A = ($8000, bank #15)? Then branch!
              ; Equivalent of CMP (pointer, bank), index.
```

Figure 6-6. Using INDCMP

immediately following JSR \$FF7D. The program cannot jump (JMP) to \$FF7D because PRIMM has to examine the stack to find the string's start address. The string itself must be terminated with a \$00 byte, and it cannot contain \$00 bytes within the string. The string also cannot be longer than 256 bytes, including the terminator byte. PRIMM does not change the contents of the 8502 registers and does not require any input parameters.



Chapter 7

Disk and I/O Operations On the C-128

The C-128, although sharing many of the C-64's I/O devices, has a much more refined software interface to I/O devices like the disk drive. For example, programs can be autobooted upon power-up of the system. The serial interface to the disk drive has been much improved, allowing vastly increased disk I/O throughput. And, while in burst mode, data on non-Commodore (MFM) disk formats can be read.

Many of these features depend on the use of the new 1571 high-speed disk drive, which was designed as a companion device for the C-128 system. However, many functions (such as autobooting) will function on the older 1541 disk drive that usually is the companion of the C-64 computer.

1541 and 1571 Disk Compatibility

The 1541 and 1571 drives are, on the surface, completely compatible; the 1571's DOS commands are really a superset of those in 1541 DOS. Almost any C-64 program that accesses the 1571 drive will perform flawlessly. Without special programming, however, the C-64 or the C-128's C-64 mode cannot take advantage of most of the enhanced features of the 1571 drive. Thus, programs will still load, save, or execute other disk operations at the same speed as the 1541.

Problems sometimes arise when the C-128 system is switched over to C-64 mode without resetting the disk drive. The user isn't usually aware of any incompatibilities because ordinary LOAD and SAVE operations function properly. However, the 1571 drive controller can "lock up" when copy-protected software, special high-speed disk-copy programs, or the various fast-load programs available for the C-64 that increase the speed of disk access are used. Some of these programs halt all operation of the C-64 until the drive returns data—if the drive is locked up, then the C-64 itself is also locked up. Only a RUN-STOP/RESTORE combination or a cold reset is likely to return control to the user.

To avoid these symptoms, the user should observe a proper power-up sequence for C-64 mode. Starting with the C-128 and all serial bus devices off, the user should first apply power to the 1571 drive and then power up the computer either with a C-64 cartridge in the expansion port or with the COMMODORE key held down. By directly entering C-64 mode, you have avoided the C-128's disk

autoboot sequence, which would have told the 1571 that it was connected to a C-128. If, however, you find it necessary to work under C-128 mode first and later switch to C-64 mode without turning the computer off and then on again, you should type

```
OPEN15,8,15,"UC>MO": CLOSE 15
```

This command will ensure that the 1571 is actually in 1541 mode. The drive will remain in 1541 mode until it is reset. As an alternative, the user can simply turn the 1571 off and on again after typing GO64.

Software that directly accesses the disk controller of a 1541 may not function properly on a 1571 even after the command sequence just given has been entered. This is because the 1571's DOS ROM has had to be modified to support the extra features—and different hardware—in the controller. Again, these DOS ROM differences should only affect some copy-protected programs, disk-copy utilities, and fast-load programs that have to call machine-code routines in the DOS ROM. Normal “well-behaved” programs should not have any trouble whatsoever.

One minor problem occurs when a 1571-formatted double-sided disk is validated (using the DOS V command) while in 1541 mode. If the disk is more than half full (there is data on both sides of the disk), a validation done while in 1541 mode will abort. This is because DOS will have run into an illegal track-number reference; track numbers greater than 35 are not recognized by the 1541 DOS. No data will be destroyed or changed, and the disk is still usable; the disk will merely have to be validated while in 1571 mode. Thus, if the disk is not yet half full, the validation while in 1541 mode will take place properly.

A hitch occurs when a 1571-formatted double-sided disk is validated on a true 1541 disk drive (not on a 1571 in 1541 mode). During validation, if 1541 DOS encounters any references to sectors on the disk's second side, DOS will abort with an error, just like the 1571 in 1541 mode. However, if the double-sided disk is less than half full, an actual 1541 drive will complete a normal validation. At the end of this

validation, when the BAM (block allocation map) is rewritten, the fourth byte (byte #3 of track 18, sector 0) of the BAM sector will be set to zero. A 1571-formatted double-sided disk has its fourth byte of track 18, sector 0 set to \$80 (decimal 128), indicating a double-sided disk. When an actual 1541 changes this byte to zero, the disk becomes single-sided; the user can set this byte back to \$80 to restore the disk's double-sided status using a sector editor program. The 1571 in 1541 mode will not change this byte; when a double-sided disk is validated in 1541 mode but is less than half full, validation will take place completely—and the disk will still be double-sided.

Organization of a 1571 Double-Sided Disk

The file structures of 1571 and 1541 disks are the same. However, there have been some slight changes and additions to the organization of the BAM sector on a 157 double-sided diskette.

Like a 1541-formatted diskette, the first 144 bytes of the BAM sector (track 18, sector 0) contain both the BAM and other disk information. The first two bytes of this sector (bytes 0 and 1) always contain \$12 \$01, the track and sector numbers of the first sector in the disk's directory. The third byte (byte #2) should always contain a \$41 byte, representing the ASCII character "A". This byte is a flag indicating that the disk can be read from or written to by a 1541, 1571, or 4040 disk drive. Other older Commodore disk formats (for example, 2031) can be read, but any attempts to write to the disk will result in a "DOS mismatch" error and the write operation will abort. The abort occurs because this third byte is not an "A" character. In fact, a user can write-protect his or her diskettes by using a sector editor to change this byte. The disk cannot be written to until it is reformatted. (There are public domain utilities that can circumvent DOS and allow this byte to be changed back to an "A"; usually these "unprotect" features are functions of programs that write-protect the disk without the use of a sector editor.)

The fourth byte (byte #3) of the disk's BAM sector contains the single-sided/double-sided disk flag mentioned previously. Bit #7 of this byte will be set if the disk is double-sided and clear if single-sided; the remaining bits have no function. Disks validated on a true 1541 drive will reset this byte to zero, while those validated on a 1571 drive—in either 1541 or 1571 mode—will leave the byte undisturbed.

Beginning at the fifth byte, the actual block allocation map begins; it extends all the way to byte #143. Each track of the disk, from 1 through 35, has a four-byte entry in the BAM. The first byte of each BAM entry contains the number of free sectors available on that particular track. The next three bytes contain a bit map of the unused sectors on that track; a high bit indicates that a sector of that track is empty, and a clear bit flags the sector as being occupied.

The field containing the disk's name occupies bytes 144 through 159 (\$90-\$9F hex). A disk name can thus be up to 16 characters long; if a disk name is shorter than this length, the unused character positions in the field are filled with \$A0 bytes (shifted-space characters). Bytes 160 and 161 always contain \$A0 shifted space characters, while bytes 162 and 163 contain the two-character disk ID. (These ID bytes are also contained on the header that precedes every sector on the disk; the header is normally unreadable except to DOS internal routines.)

Bytes 164 and 167 through 170 again contain \$A0 shifted-space filler bytes, while bytes 165 and 166 contain the characters "2A", indicating the Commodore DOS version. Bytes 171 through 220 of the BAM sector contain \$00 (null) filler bytes.

Up to this point, there's been little difference between a 1541 and a 1571 (double-sided disk) BAM sector. However, the remainder of this BAM sector is unused in a 1541 (or 1571 single-sided) diskette and consequently is occupied by \$00 filler bytes. In contrast, the same region (bytes 221-255) on a 1571 double-sided disk contains information about sector allocation on the second side of the disk. It does not contain the BAM bit map itself, but rather a sequence of bytes representing the number of sectors available (unused) on tracks 36 through 70. (Byte 221 contains information for track #36, byte 222

contains that for track #37, and so on). Each byte in this region serves the same function for tracks on side 2 as does the first byte of each BAM entry for tracks on side 1.

Note that byte 238 of the BAM sector will always contain a \$00 byte. This location contains the number of free sectors on track 53, which is the eighteenth track on side 2 of the diskette. So that DOS can follow many of the same conventions on the second side of the disk that it follows on the first side, it will not record any file data on this track — after all, track 18 on side 1 is the directory track. Sector 0 of track #53 does contain some information (to be discussed next), but the remainder of this track is untouched by DOS file management routines. This is a good area in which to store “hidden” information on double-sided disks.

The only remaining data structure needed to complete the BAM for a 1571 double-sided diskette is the actual three-byte-per-sector BAM bit map itself. Since there is no other region on side 1 that could safely store such important information without affecting compatibility or being overwritten by actual disk data, the BAM bit map for side 2 is stored on track 53, sector 0. Because this track is unused by DOS, it’s a safe place to store the bit map.

Unlike the BAM entries for tracks on side 1 of the disk, which contain four-byte entries, the additional BAM entries are only three bytes long; each entry contains only the BAM bit map for tracks 36 through 70. The byte that indicates the number of free sectors per track has already been recorded on track 18, sector 0, bytes 221 through 255. Because of the shortened BAM entries, the BAM bit map for all of side 2 occupies only 105 bytes (bytes 1 through 105) of track 53, sector 0. The remaining bytes in this extra BAM sector contain \$00 filler bytes.

It should be remembered that the space allocated for storage of actual directory information has not been increased. (The directory begins on track 18, sector 1, and can fill all sectors on that track except for the BAM sector.) Because the last 18 sectors of track 18 store directory information, 8 entries per sector, the maximum file limit for both single- and double-sided disks is still 144. The directory

Table 7-1. Structure of a Typical Directory Sector (Track# = 18, Sector# > 0)

| <i>Byte #</i> | <i>Function</i> |
|---------------|--|
| 0, 1 | Link bytes containing track and sector numbers of next sector in directory. The track byte will always contain \$12, with the exception of the last directory sector. In this case, the track link byte will contain a \$00, and the sector link byte will contain a value indicating the number of bytes used in the last sector. |
| 2-31 | Directory entry #1. |
| 34-63 | Directory entry #2. |
| 66-95 | Directory entry #3. |
| 98-127 | Directory entry #4. |
| 130-159 | Directory entry #5. |
| 162-191 | Directory entry #6. |
| 194-223 | Directory entry #7. |
| 226-255 | Directory entry #8. |

sectors follow the patterns outlined in Tables 7-1 and 7-2, and their structure remains unchanged from the one used in the 1540/1541 disk drive.

The 1571 Drive's CHGUTL Utility Commands

The 1571 drive contains a special utility command called *CHGUTL*. Instead of sending the name CHGUTL to the disk drive, its special functions are invoked using the "U0" series of disk commands. The CHGUTL functions work (with two exceptions) in both 1541 and 1571 modes, and it can be of use to C-64 programmers as well. All

Table 7-2. Structure of a Typical Directory Entry for a File

| <i>Byte #</i> | <i>Function</i> |
|---------------|---|
| 0 | Normally contains \$80 logically ORed with file-type value. When file-type value ORed with \$C0, the file is locked from being scratched. DOS does not support file locking, so the user must change the file-type byte. A \$00 byte indicates a scratched file whose directory entry can be overwritten. The file-type values are \$00 = deleted (not normally used) \$01 = sequential (SEQ) file \$02 = program (PRG) file \$03 = user file \$04 = relative file |
| 1, 2 | Track and sector numbers of first sector in file. |
| 3-18 | Filename field, maximum of 16 characters in length. Unused positions in field contain \$A0 (shifted-space) filler bytes. |
| 19-20 | Track and sector numbers for first "side sector" of a relative file. |
| 21 | Record length value for relative file usage. |
| 22-25 | Currently unused. |
| 26-27 | Track and sector numbers for first sector of replacement file. |
| 28-29 | Number of sectors in file. Stored in standard 6502 low-byte/high-byte order. |

CHGUTL functions are accessed through the command channel (#15). The format of a typical CHGUTL command is

PRINT#15, "U0>" + *command string bytes*

This assumes, of course, that logical file #15 will access channel #15, the command channel.

Changing the Sector Interleave Value

The sector interleave factor determines the “gap” between sectors written to disk. For example, assume that some sector of a file is written to track 23, sector 2. With a sector interleave value of 3, the next sector of the file will normally be stored on track 23, sector 5, unless that particular sector is already being used by another file. Successive sectors of the file will use sectors 8, 11, 14, and 17 (with the provision that the BAM has declared that they are currently unused). DOS will then wrap around and store the next data block at, say, sector 0 of the same track. When the track eventually fills, storage will progress to the next available sector on another track.

Sector interleaves were necessary on older disk drives with slower controllers. A smaller interleave value (1, for example) where sectors are stored as closely together as possible will speed disk access considerably simply because fewer rotations of the diskette are required to access a series of sectors. The BASIC statement

```
PRINT#15, "U0>S" + CHR$(interleave value)
```

will change the sector interleave to the desired value.

Controlling the Number Of Retries

If 1571 DOS has trouble reading a sector from a disk, it will retry the read five times (its cold-start value). Only then will a disk error be signalled to the user, both through the command channel and with the flashing green disk status light. It may occasionally be necessary to increase or decrease this value.

For example, many commercial C-128 software packages are copy-protected by recording an error-laden track on the disk during production. When run, the software checks the type of errors generated by the bad track in the hope that users cannot exactly duplicate

the sequence of errors to make a working backup copy. One side-effect of this copy-protection—besides hampering the usefulness of the software—is that it frequently bangs the disk drive’s read/write head against a metal “stop” in the drive mechanism. Gradually, the drive can be thrown out of alignment, requiring a trip to the service shop. By reducing the number of retries, head banging can be kept to a minimum.

On the other hand, it may be desirable to increase this retry counter if the user is reading data from an old, damaged disk or from a diskette that has been formatted on a misaligned drive. Many, many “seek” operations (where the drive checks for the presence of data halfway between the desired track and the two surrounding tracks) may be required before the sector can be read without errors. These seek operations will not usually cause head banging and consequently will not throw the drive mechanism out of alignment.

To set the retry counter to the desired value, use the statement

```
PRINT#15, "U0>R" + CHR$(# of retries)
```

Testing for Proper ROM Function

The statement

```
PRINT#15, "U0>T"
```

sends the DOS command to initiate a ROM *signature analysis*. Basically, this analysis calculates a checksum for the code in the 1571’s ROM. If the test fails, the 1571’s green light will flash four times.

Switching Between 1541 And 1571 Modes

While in 1541 mode, the DOS command string “U0>M1” will switch the 1571 drive into 1571 mode. Use of the burst transfer routines and

double-sided operation can now take place. In 1571 mode, the DOS command string "U0>M0" will cause a switchover to 1541 operation. Only single-sided Commodore disks can be read in this mode. Files stored on a double-sided disk can also be read if the complete file resides on the first side of the diskette.

The Head Select Function (1541 Mode Only)

The 1571 can access data on the second side of a double-sided diskette through the use of the *head select* commands. These commands enable read/write operations to either side 1 or side 2 of a diskette. However, these commands use references to disk sides 0 and 1; this refers to what most users call sides 1 and 2, respectively.

The DOS command string "U0>H0" selects disk side 0—the side containing the first half of the disk, along with the directory—while the DOS command string "U0>H1" selects the second side. Thus, a program in C-64 mode can access data on a 1571 double-sided diskette while in 1541 mode. Of course, this extra data does not have the benefit of a directory structure, since 1541 mode can only deal with one diskette side at a time.

This command string makes it quite possible to edit a double-sided 1571 diskette with one of the many commercial or public domain sector editors available for the C-64 (or C-64 mode). Using the "DOS Command" function that most good sector editors support, the user can type **U0>H0** or **U0>H1** to select the desired side for editing. References to tracks 36 through 70 must be translated to tracks 1 through 35 before selecting the desired track and sector for editing, since track numbers larger than 35 will generate DOS errors.

These two commands can be used to simulate a "dual" 1541 disk drive. Each side of the 1571 diskette must be formatted separately, after the appropriate "U0>Hx" command has been given to the drive. From then on, files can be opened, closed, loaded, or saved to either side of the diskette, since each side has its own directory and BAM. Each side is independent of the other, and the appropriate

“U0>Hx” command will select the desired “virtual” drive. This imitation of a dual drive does have one major deficiency, however: the C-128 (in C-128 mode), with the 1571 (in 1571 mode), will not be able to access the files on the second side of the diskette because they have not been declared in the directory on the first side. Consequently, the 1571 must be placed in 1541 mode (using the “U0>M0” DOS command string) despite the fact that the computer is running under C-128 mode, and the inherent speed advantages of the C-128/1571 combination cannot be realized.

It should be noted that users who have “double-sided” their floppy disks for use on a 1541 (using a hole punch to create a space for a write-protect tab) must still flip the disk over to use data on the second side, due to the direction in which the data was stored. Also, the second side of a disk formatted in the 1571’s 1541 mode will not be readable on an ordinary 1541 by merely flipping the disk over because of the reversed direction of recording.

Changing the Drive’s Device Number

The 1571 drive has two small switches that allow the user to select one of four device numbers—8, 9, 10, or 11. This is quite useful for two-drive situations, where one drive is always intended to be device number 8 and the other device number 9. However, it’s often desirable to control this function by using a software technique. The CHGUTL utility provides a function that changes the device number of the drive to any number in the range 4 through 31. This DOS command string is merely

“U0>” + CHR\$(*device #*)

(The use of device number 31 isn’t recommended.) If the device number is changed to 4, the disk can receive data intended for printout. Some manipulation of filenames will be necessary, since the

printer does not normally require a command string before opening a channel. This can be a useful technique to save formatted printout data from a word processor or spreadsheet program. This method can also be used to divert graphics output, normally intended for a printer, to a disk file.

1541 and 1571 Drive Internals: The Job Queue

The 1541 and 1571 drives have an internal 6502 processor along with internal ROM software and RAM for buffer and system variable storage. This controller functions in two modes: interface and FDC (floppy disk controller). Commands are passed between these two modes in what is known as a *job queue*; this queue is located in DOS memory at \$0000-\$0005. See Table 7-3. Each of the five positions in this queue corresponds to a job assigned to one of the five 256-byte RAM buffers at \$0300, \$0400, \$0500, \$0600, and \$0700 (see Figure 6-3). The last buffer, at \$0700, is reserved exclusively for BAM (block allocation map) storage, while the other buffers are used for normal file I/O. User-written code can be stored in these buffers with care.

Every ten milliseconds, the interface mode switches over to the floppy disk controller mode and examines codes in the job queue.

Table 7-3. DOS Memory Usage and Buffer Areas

| <i>Hex Address</i> | <i>Description</i> |
|--------------------|--|
| \$0000-\$0005 | Job Queue. |
| \$0006-\$0011 | Track and sector queue. |
| \$0012-\$0013 | Master copy of disk ID. |
| \$0300-\$07FF | Buffers 0-4 (buffer #4 is reserved for the BAM). |

These codes are listed in Table 7-4. Each of these command codes refers to a corresponding pair of memory locations in the track/sector queue, indicating the track and sector of the disk to be read from, written to, verified, and so on. A status code will be returned in the job queue upon completion of the operation. Note that the command codes can be differentiated from the return codes by examining the high bit (bit #7) of the value returned in a job queue location. A \$01 value indicates the successful completion of an operation with no errors.

Depending on the operation, some intermediate data may be left in a buffer even if an error occurred. For example, assume that an \$09 error code (a header checksum error) is returned by the FDC. A complete read operation has taken place, and perhaps only one byte (or even one bit) is incorrect. The complete data read from that particular sector is still in the buffer, despite the presence of an error code in the job queue.

Always set up the appropriate track and sector parameters

Table 7-4. Internal DOS Job Codes and Functions

| <i>Hex Job Codes</i> | <i>Description</i> |
|----------------------|-----------------------------------|
| \$80 | READ |
| \$90 | WRITE |
| \$A0 | VERIFY (normally called by WRITE) |
| \$B0 | SEEK |
| \$B8 | SECTOR SEEK |
| \$C0 | BUMP |
| \$D0 | JUMP |
| \$E0 | EXECUTE |

Table 7-5. Table of Codes Returned After Job Completed

| <i>Return Codes</i> | <i>Description</i> |
|---------------------|--|
| \$01 | NO ERROR. |
| \$02 | CAN'T FIND BLOCK HEADER. |
| \$03 | NO SYNC. |
| \$04 | DATA BLOCK NOT PRESENT. |
| \$05 | CHECKSUM ERROR IN DATA. |
| \$07 | WRITE-VERIFY ERROR. |
| \$08 | WRITE PROTECT TAB PRESENT. |
| \$09 | CHECKSUM ERROR IN HEADER. |
| \$0A | DATA EXTENDS INTO NEXT BLOCK (long block). |
| \$0B | DISK ID MISMATCH. |
| \$10 | 10-bit (GCR) to 8 bit (byte) DECODE ERROR. |

before writing to the job queue. There is no protection against the head moving out to an illegal track or sector, because direct accesses to the job queue bypass all Commodore DOS error checking. In fact, the head may attempt to move past track #40 if the job queue is incorrectly used. This may even lock the head by retracting it too far, requiring a trip to the service shop.

The EXECUTE job code will execute 6502 machine language routines stored in the buffer once the drive is up to speed. EXECUTE always considers the start address of the code to be at the beginning of a buffer region (for example, \$0300). The BUMP job does not require the setup of any track or sector parameters; it merely bumps (bangs) the head against the track #0 stop. Table 7-5 gives an example of coding that demonstrates how a sector can be read from the job queue. The code resides in the third buffer (see Figure 7-1) at \$0600

and can be called using the DOS memory-execute (M-E) command as follows:

```
OPEN 15,8,15
PRINT#15,"M-E"+CHR$(addr., low byte) + CHR$(addr.,
high byte)
CLOSE 15
```

The best technique for working with drive-resident machine code is to enter it into C-128 RAM using the ML monitor. Once saved to disk, it can be accessed, byte by byte, as a sequential file and then POKEd into drive memory using the DOS memory-read command. Assuming that the command channel is open and that A\$ is a string containing the bytes of the desired machine-code program to be loaded into disk RAM, the following command will send the data to any desired RAM address in the disk drive:

```
PRINT#15,"M-W"+CHR$(addr low)+CHR$(addr hi)+
CHR$(numb)+A$;
```

The parameters *addr low* and *addr hi* contain the starting address of

```

$0600  LDA #08      ;TRACK
        STA $06   ;STORE TO TRACK TABLE
        LDA #01   ;SECTOR
        STA $07   ;STORE TO SECTOR TABLE
        LDA #$B0  ;SEEK JOB TO READ ID
        STA $00   ;STORE TO JOB QUEUE

LOOP1   LDA $00
        BMI LOOP1 ;WAIT TILL SEEK DONE
        CMP #01   ;GOOD RETURN
        BNE ERROR
        LDA #$80  ;READ JOB
        STA $0000 ;STORE TO JOB QUEUE

LOOP2   LDA $0000
        BMI LOOP2 ;WAIT TILL READ DONE
        CMP #01   ;GOOD RETURN?
        BNE ERROR
        RTS      ;DATA AT $0300-03FF
```

Figure 7-1. Example of reading track 8, sector 1 into buffer 0

the region to receive the bytes contained in A\$. The *numb* parameter contains the number of bytes to be sent and should equal LEN(A\$). The maximum number of bytes that can be sent to the drive is 34 (decimal), so longer routines may have to be sent using several M-W command strings.

Finding areas within the disk drive in which to store machine-code routines is a difficult task. If too many files are open, the RAM buffer region where the user-written code resides may be written over by data being transferred to or from the disk itself. It's usually best to have as few OPEN files (if any) as possible. The machine code should perform its task and then relinquish control back to DOS so that normal file handling can take place. Additionally, free zero-page memory is hard to find in Commodore disk drives. Table 7-6 shows a zero-page memory map for the 1500 series disk drives. Note that Commodore DOS still contains remnants of the older types of DOS for their dual disk drives. Generally, locations used by "drive 1" are not used in the 1541 or 1571 drives, so they can be used by programmers creating drive-resident code.

Table 7-6. Memory Map of Zero-Page RAM Usage by Commodore Disk Drives

| <i>Hex Location</i> | <i>Content</i> | <i>CBM Label</i> | <i>Function</i> |
|---------------------|----------------|------------------|-----------------------------|
| 00-05 | 00 00 | JOBS | Job Que Buffer #0. |
| | 01 00 | | Buffer #1. |
| | 02 00 | | Buffer #2. |
| | 03 00 | | Buffer #3. |
| | 04 00 | | Buffer #4. |
| | 05 00 | | Buffer #5. |
| 06-11 | 06 00 | HDRS | Job headers Buffer #0— Low. |
| | 07 00 | | Buffer #0— High. |
| | 08 00 | | Buffer #1— Low. |
| | 09 00 | | Buffer #1— High. |
| | 0A 00 | | Buffer #2— Low. |

Table 7-6. Memory Map of Zero-Page RAM Usage by Commodore Disk Drives (*continued*)

| <i>Hex Location</i> | <i>Content</i> | <i>CBM Label</i> | <i>Function</i> |
|---------------------|----------------|------------------|----------------------------------|
| | 0B 00 | | Buffer #2— High. |
| | 0C 00 | | Buffer #3— Low. |
| | 0D 00 | | Buffer #3— High. |
| | 0E 00 | | Buffer #4— Low. |
| | 0F 00 | | Buffer #4— High. |
| | 10 00 | | Buffer #5— Low. |
| | 11 00 | | Buffer #5— High. |
| 12-15 | 12 00 | DSKID | Master copy of disk ID: Drive 0. |
| | 13 00 | | Drive 0. |
| | 14 00 | | Not used— Drive 1. |
| | 15 00 | | Not used— Drive 1. |
| 16-1A | 16 00 | HEADER | Image of last header: ID Byte 1. |
| | 17 00 | | ID Byte 2. |
| | 18 00 | | Track. |
| | 19 00 | | Sector. |
| | 1A 00 | | Checksum. |
| 1B | 1B 00 | ACTJOB | Controllers active job. |
| 1C-1D | 1C 01 | WPSW | Write-Protect change flag: |
| | | | Drive 0. |
| | 1D 01 | | Drive 1. |
| 1E-1F | 1E 10 | LWPT | Last state of WP switch: |
| | | | Drive 0. |
| | 1F 00 | | Drive 1. |
| 20 | 20 00 | DRVST | Drives current status: Drive 0. |
| 21 | 21 00 | | Speed timing flag. |
| 22-23 | 22 00 | DRVTRK | Drive track number: Drive 0. |
| | 23 00 | | Drive 1. |
| 24-2D | 24 00 | STAB | Storage table for GCR conver- |
| | | | sion. |
| | 25 00 | | |
| | 26 00 | | |
| | 27 00 | | |
| | 28 00 | | |
| | 29 00 | | |

Table 7-6. Memory Map of Zero-Page RAM Usage by Commodore Disk Drives (*continued*)

| <i>Hex Location</i> | <i>Content</i> | <i>CBM Label</i> | <i>Function</i> |
|---------------------|----------------|------------------|-----------------------------------|
| | 2A 00 | | |
| | 2B 00 | | |
| | 2C 00 | | |
| | 2D 00 | | |
| 2E-2F | 2E 00 | SAVPNT | Temporary save pointer location. |
| | 2F 00 | | |
| 30-31 | 30 00 | BUFPNT | Active buffer pointer. |
| | 31 00 | | |
| 32-33 | 32 00 | HDRPNT | Header pointer: Track. |
| | 33 00 | | Sector. |
| 34 | 34 00 | GCRPNT | GCR pointer. |
| 35 | 35 00 | GCRERR | Indicates GCR decode error. |
| 36 | 36 00 | BYTCNT | Byte counter for GCR/binary conv. |
| 37 | 37 00 | BITCNT | Bit counter. |
| 38 | 38 00 | BID | Data block ID. |
| 39 | 39 00 | HBID | Header block ID. |
| 3A | 3A 00 | CHKSUM | Checksum. |
| 3B | 3B 00 | HINIB | .not used directly. |
| 3C | 3C 00 | BYTE | .not used directly. |
| 3D | 3D 00 | DRIVE | Drive number. |
| 3E | 3E FF | CDRIVE | Current active drive number. |
| 3F | 3F 00 | JOBN | Current job number. |
| 40 | 40 00 | TRACC | Track—internal storage location. |
| 41 | 41 00 | NXTJOB | Next job. |
| 42 | 42 00 | NXTRK | Next track. |
| 43 | 43 00 | SECTR | Sector per track for formatting. |
| 44 | 44 00 | WORK | Working storage location. |
| 45 | 45 00 | JOB | Job type. |

Table 7-6. Memory Map of Zero-Page RAM Usage by Commodore Disk Drives (*continued*)

| <i>Hex Location</i> | <i>Content</i> | <i>CBM Label</i> | <i>Function</i> |
|---------------------|----------------|------------------|---|
| 46 | 46 | 00 | CTRACK .not used directly. |
| 47 | 47 | 07 | DBID Data block ID. |
| 48 | 48 | 00 | ACLTIM Accel time delay. |
| 49 | 49 | 39 | SAVSP Save stack pointer. |
| 4A | 4A | 00 | STEPS Steps to desired track. |
| 4B | 4B | 00 | TMP Temporary storage location. |
| 4C | 4C | 00 | CSECT Current sector. |
| 4D | 4D | 00 | NEXTS Next sector. |
| 4E | 4E | 00 | NXTBF Pointer to next GCR source buffer. |
| 4F | 4F | 00 | NXTPNT Pointer to next byte location in buffer. |
| 50 | 50 | 00 | GCRFLG GCR/Binary flag in active buffer. |
| 51 | 51 | FF | FTNUM Current format track. |
| 52-55 | 52 | 00 | BTAB Binary table: GCR/Binary work area. |
| | 53 | 00 | |
| | 54 | 00 | |
| | 55 | 00 | |
| 56-5D | 56 | 00 | GTAB GCR table: GCR/Binary work area. |
| | 57 | 00 | |
| | 58 | 00 | |
| | 59 | 00 | |
| | 5A | 00 | |
| | 5B | 00 | |
| | 5C | 00 | |
| | 5D | 00 | |
| 5E | 5E | 04 | AS Number of steps to accel with head. |
| 5F | 5F | 04 | AF Acceleration factor. |
| 60 | 60 | 00 | ACLSTP Steps to go before complete. |

Table 7-6. Memory Map of Zero-Page RAM Usage by Commodore Disk Drives (*continued*)

| <i>Hex Location</i> | <i>Content</i> | <i>CBM Label</i> | <i>Function</i> |
|---------------------|----------------|------------------|----------------------------------|
| 61 | 61 00 | RSTEPS | Number of run steps. |
| 62-63 | 62 05 | NXTST | Pointer to stepping run—\$FA05. |
| | 63 FA | | |
| 64 | 64 C8 | MINSTP | Minimum steps required to accel. |
| 65-66 | 65 22 | VNMI | Indirect for NMI—\$EB22. |
| | 66 EB | | |
| 67 | 67 00 | NMIFLG | NMI in progress flag. |
| 68 | 68 00 | AUTOFG | Auto driver initialization flag. |
| 69 | 69 0A | SECINC | Sector increment for sequential. |
| 6A | 6A 05 | REVCNT | Error recovery count. |
| 6B-6C | 6B EA | USRJMP | User jump table pointer—\$FFE A. |
| | 6C FF | | |
| 6D-6E | 6D 00 | BMPNT | Bit map pointer. |
| | 6E 00 | | |
| 6F-74 | 6F 6F | TEMP: T0 | Temporary work space. |
| | 70 00 | T1 | |
| | 71 00 | T2 | |
| | 72 FF | T3 | |
| | 73 00 | T4 | |
| | 74 00 | | |
| 75-76 | 75 00 | IP | Indirect pointer variable. |
| | 76 01 | | |
| 77 | 77 28 | LSNADR | Listen address device # + \$20. |
| 78 | 78 48 | TLKADR | Talker address device # + \$40. |
| 79 | 79 00 | LSNACT | Active listener flag. |
| 7A | 7A 00 | TLKACT | Active talker flag. |

Table 7-6. Memory Map of Zero-Page RAM Usage by Commodore Disk Drives (*continued*)

| <i>Hex Location</i> | <i>Content</i> | <i>CBM Label</i> | <i>Function</i> |
|---------------------|---|------------------|------------------------------------|
| 7B | 7B 00 | ADRSED | Addressed flag. |
| 7C | 7C 00 | ATNPND | Attention pending flag. |
| 7D | 7D 00 | ATNMOD | In ATN mode. |
| 7E | 7E 00 | PRGTRK | Last program accessed. |
| 7F | 7F 00 | DRVNUM | Current driver number. |
| 80 | 80 00 | TRACK | Current track. |
| 81 | 81 00 | SECTOR | Current sector. |
| 82 | 82 04 | LINDX | Logical index. |
| 83 | 83 0F | SA | Current secondary address. |
| 84 | 84 6F | ORGSA | Original secondary address. |
| 85 | 85 3F | DATA | Temporary data byte. |
| 86 | 86 00 | R0 | Temporary work area. |
| 87 | 87 00 | R1 | Temporary work area. |
| 88 | 88 00 | R2 | Temporary work area. |
| 89 | 89 00 | R3 | Temporary work area. |
| 8A | 8A 00 | R4 | Temporary work area. |
| 8B-8E | 8B 00 8C 00 8D 00 8E 00 | RESULT | Result of multiply/divide rtns |
| 8F-93 | 8F 00 90 00 91 00 92 00 93 00 | ACCUM | Remainder of multiply/divide rtns. |
| 94-95 | 94 04 95 02 | DIRBUF | Pointer to directory buffer. |
| 96 | 96 00 | ICMD | IEEE command in: not used. |
| 97 | 97 00 | MYP A | MY PA flag: not used. |

Table 7-6. Memory Map of Zero-Page RAM Usage by Commodore Disk Drives (*continued*)

| <i>Hex Location</i> | <i>Content</i> | <i>CBM Label</i> | <i>Function</i> |
|---------------------|----------------|------------------|-------------------------------------|
| 98 | 98 00 | CONT | Serial bit counter. |
| 99-A6 | 99 00 | BUFTAB | Buffer byte ptrs |
| | 9A 03 | | : Buffer #0 Low. |
| | 9B 00 | | : Buffer #0 High. |
| | 9C 04 | | : Buffer #1 Low. |
| | 9D 00 | | : Buffer #0 High. |
| | 9E 00 | | : Buffer #2 Low. |
| | 9F 05 | | : Buffer #0 High. |
| | A0 00 | | : Buffer #3 Low. |
| | A1 06 | | : Buffer #0 High. |
| | A2 00 | | : Buffer #4 Low. |
| | A3 07 | | : Buffer #0 High. |
| | A4 00 | | : CMD Buffer Low. |
| | A5 02 | | High. |
| | A6 D6 | | : Error Buffer Low. |
| | A7 02 | | High. |
| A7-AD | A7 FF | BUF0 | Inactive flags for buffers. |
| | A8 FF | | |
| | A9 FF | | |
| | AA FF | | |
| | AB 05 | | |
| | AC 06 | | |
| | AD FF | | |
| AE-B4 | AE FF | BUF1 | Active flags for buffers. |
| | AF FF | | |
| | B0 FF | | |
| | B1 FF | | |
| | B2 FF | | |
| | B3 FF | | |
| | B4 FF | | |
| B5 | B5 00 | NBKL | Number of blocks low. |
| B5-BA | B5 00 | RECL | Low record # to find relative file. |
| | B6 00 | | |
| | B7 00 | | |

Table 7-6. Memory Map of Zero-Page RAM Usage by Commodore Disk Drives (*continued*)

| <i>Hex Location</i> | <i>Content</i> | <i>CBM Label</i> | <i>Function</i> | |
|---------------------|----------------|------------------|-----------------|---------------------------------------|
| | B8 | 00 | | |
| | B9 | 00 | | |
| | BA | 00 | | |
| BB | BB | 00 | NBKH | Number of blocks high. |
| BB-C0 | BB | 00 | RECH | High record # to find relative file. |
| | BC | 00 | | |
| | BD | 00 | | |
| | BE | 00 | | |
| | BF | 00 | | |
| | C0 | 00 | | |
| C1-C6 | C1 | 00 | NR | Next record table. |
| | C2 | 00 | | |
| | C3 | 00 | | |
| | C4 | 00 | | |
| | C5 | 00 | | |
| | C6 | 00 | | |
| C7-CC | C7 | 00 | RS | Relative record size table. |
| | C8 | 00 | | |
| | C9 | 00 | | |
| | CA | 00 | | |
| | CB | 00 | | |
| | CC | 00 | | |
| CD-D2 | CD | FF | SS | Side sector table. |
| | CE | FF | | |
| | CF | FF | | |
| | D0 | FF | | |
| | D1 | FF | | |
| | D2 | FF | | |
| D3 | D3 | 00 | F1PTR | File steam 1 pointer. |
| D4 | D4 | 00 | RECPTR | 1st byte wanted from relative record. |
| D5 | D5 | 00 | SSNUM | Side sector number of relative file. |

Table 7-6. Memory Map of Zero-Page RAM Usage by Commodore Disk Drives (*continued*)

| <i>Hex Location</i> | <i>Content</i> | <i>CBM Label</i> | <i>Function</i> |
|---------------------|--|------------------|--|
| D6 | D6 00 | SSIND | Index into side sector. |
| D7 | D7 00 | RELPTR | Pointer to first byte wanted in relative file. |
| D8-DC | D8 00 D9 00 DA 00 DB 00 DC 00 | ENTSEC | Sector of directory entries. |
| DD-E1 | DD 00 DE 00 DF 00 E0 00 E1 00 | ENTIND | Index of directory entries. |
| E2-E6 | E2 00 E3 00 E4 00 E5 00 E6 00 | FILDRV | Default flag drive number. |
| E7-EB | E7 00 E8 00 E9 00 EA 00 EB 00 | PATTYP | Pattern, replace, closed-flags, type. |
| EC-F1 | EC 00 ED 00 EE 00 EF 00 F0 00 F1 00 | FILTYP | Channel file type. |
| F2-F7 | F2 00 F3 00 F4 00 F5 00 | CHNRDY | Channel status. |

Table 7-6. Memory Map of Zero-Page RAM Usage by Commodore Disk Drives (*continued*)

| <i>Hex Location</i> | <i>Content</i> | <i>CBM Label</i> | <i>Function</i> |
|---------------------|----------------|------------------|--|
| | F6 | 01 | |
| | F7 | 88 | |
| F8 | F8 | 80 | EOIFLG Temporary EOI. |
| F9 | F9 | 00 | JOBNUM Current job number. |
| FA-FE | FA | 00 | LRUTBL Least recently used buffer table. |
| | FB | 01 | |
| | FC | 02 | |
| | FD | 03 | |
| | FE | 06 | |
| FF-100 | FF | 00 | NODRV No drive flag: Drive 0 |
| | 100 | B8 | Drive 1 not used. |

Autobooting Programs With the C-128

Whenever a C-128 system is powered up, it first checks the disk in the drive (either a 1541 or 1571) for the presence of autoboot code on track 1, sector 0. The Kernal routine that accomplishes this is called `BOOT_CALL`, and its jump-table entry is located at `$FF53` in ROM. (See the Kernal chapter for `BOOT_CALL` parameters and error returns.)

`BOOT_CALL` is normally called after BASIC 7.0 has been initialized but before control has passed to the user. It will also be called if the `BOOT` command is encountered after the C-128 is running BASIC. It first closes any open files on the device and then loads the 256 bytes on track 1, sector 0 into the cassette-tape buffer. An autoboot sector is identified by the ASCII characters "CBM" appearing in the first three bytes of the sector. Table 7-7 shows the

Table 7-7. Track 1/Sector 0 Boot-Sector Layout

| | | | | | | | |
|-----------|-----------|-------|---------------|---------|-------------|-------|--------|
| Byte #: | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Contents: | "C" | "B" | "M" | adL | adH | bank# | #sctrs |
| Byte #: | 7 . . . n | (n+1) | (n+2)...(n+x) | (n+x+1) | (n+x+2).... | | |
| Contents: | (title) | \$00 | (filename) | \$00 | (ML Code) | | |

arrangement of a typical boot sector.

If the parameter #sctrs (number of sequential sectors to be loaded) is greater than zero, sectors 1, 2, 3, . . . will be read into RAM at the address pointed to by the bytes contained in the adL and adH parameters. The bank# parameter (the sixth byte of the sector) indicates which RAM bank will receive the data from the sequentially loaded sectors.

If, on the other hand, the length of the filename is greater than zero, the BOOT__CALL routine will attempt to load the file whose name is contained in the second variable-length field. The adL, adH, and bank# parameters still indicate the address and RAM bank where the file is to be loaded.

Any disk errors encountered during either the sequential sector load or the file load cause the BOOT__CALL routine to abort and send a "UI" disk command string that will reinitialize the disk drive. The title string is optional; if it is included, the message "Booting <filename>" will appear on the screen.

After any necessary disk operations take place, the BOOT__CALL routine calls the machine language code autoloaded from track 0, sector 1, using a JSR instruction.

The routine listed in Figure 7-2 is an example of a boot-sector routine that automatically switches the 1571 drive to 1541 mode. It clears the screen and then selects the underline cursor (available only to 80-column mode). Before finishing, the code engages the C-128's fast (2-MHz clock) mode. It is intended for use with a 1571 disk drive, since it uses functions of the CHGUTL command.

1. Setup of the "CBM" key string and title.

```
>00B00 43 42 4D 00 00 00 00 49 4E 49 54 49 41 4C 49 5A:CBM....INITIALIZ
>00B10 49 4E 47
```

2. Enter code using the monitor's "A" command. Code begins at \$0B14.

```
A 0B14 LDA #S00 ; Load MMU configuration register with
STA $FF00 ; data to select system ROMs & I/O block.
LDA #S05 ; Five-byte DOS command string: "UD>M0"
LDX #S53 ; whose address is at $0B53
LDY #S08 ;
JSR $FFBD ; Call SETNAM to setup filename parameters.
LDX #S00 ; Filename in RAM Bank #0.
JSR $FF68 ; Call SETNAM.
LDA #S0F ; Logical file #15.
LDX #S08 ; Device #8.
LDY #S0F ; Secondary address 15 (DOS command channel).
JSR $FFBA ; Call SETLFS.
JSR $FFC0 ; Call OPEN, DOS command string sent to drive,
; and 1571 is now in 1541 mode.
LDA #S0F ; Close file #15.
JSR $FFC3
LDA #S93 ; Clear screen.
JSR $FFD2
LDA #S1B ; Print ESCape character and ...
JSR $FFD2
LDA #S55 ; U - <ESC><U> selects underline cursor
; in 80-column mode.
JSR $FFD2 ;
LDA #01 ; Turn on 2 MHz clock bit on VIC-II chip.
STA $D030 ;
RTS ; Return to BOOT-CALL.
```

3. Create any other necessary strings. Observe ending address.

```
>00B53 55 30 3E 4D 30 00 00 00 00 00 00 00 00 00 00:UD>M0.....
```

4. Save the code and data just entered to disk as a normal file (in case of an accident). Reenter BASIC, and use the following program to transfer data stored between \$0B00 and \$0B56 to track 1, sector 0. The disk will now contain an autoboot section.

```
10 INPUT "HEX END ADDRESS ="; A$
20 EA = DEC(A$)
30 FOR I = 2816 TO EA: T$ = T$ + CHR$( PEEK(I)) : NEXT
40 OPEN 15, 8, 15: OPEN 2, 8, 2
50 PRINT#2,T$: PRINT#15, "U2";2;0;1;0
60 CLOSE 2: CLOSE 15
70 END
```

Figure 7-2. Example of creation of an autobooting configuration routine

Burst Mode Data Transfer With the 1571 Drive

The C-128, when combined with the 1571 disk drive, supports *burst mode* data transfer. Although the serial-bus I/O routines within the Kernal have much greater throughput than their counterparts on previous PET/CBM computers because of the new, fast serial protocol, the ROM routines still do not support burst mode. For example, the data transfer rate on a C-64/1541 system is approximately 360 bps (bytes/second), while the standard fast-transfer protocol on a C-128/1571 system is approximately 1600 bps. In contrast, the C-128/1571 burst mode transfer rate can exceed 3500 bytes per second as a theoretical maximum. In practice, both of these values should be devalued by about 20% to account for time used by the track-to-track movement of the head, time used to lock onto a particular track, and the time consumed while a sector “seek” is in progress. Table 7-8 lists all the burst mode commands available to 1571 DOS. Note, however, that user-written routines are needed to take advantage of this high-speed data transfer.

Table 7-8. 1571 Burst Mode Commands

| <i>Function</i> | <i>Byte Sequence (Prefixed by “U0” String)</i> | <i>Burst Output</i> |
|---|--|--|
| READ SECTORS | | |
| MFM disk side 0 or GCR disk (either side) | \$40, trk#, sect#, # of sectors | 1 burst status byte followed by data bytes (per sector). |

Table 7-8. 1571 Burst Mode Commands (*continued*)

| <i>Function</i> | <i>Byte Sequence (Prefixed by "U0" String)</i> | <i>Burst Output</i> |
|---|--|--|
| MFM disk side 1 | \$80, trk#, sect#, # of sectors. | 1 burst status byte followed by data bytes (per sector). |
| WRITE SECTORS | | |
| MFM disk side 0 or GCR disk (either side) | \$42, trk#, sect#, # of sectors. | 1 burst status byte after each sector transferred. |
| MFM disk side 1 | \$52, trk#, sect#, # of sectors. | 1 burst status byte after each sector transferred. |
| INQUIRE DISK (Log in an MFM or GCR disk before a read or write operation.) | | |
| MFM disk side 0 or GCR disk | \$04 | 1 burst status byte. |
| MFM disk side 1 | \$14 | 1 burst status byte. |
| FORMAT DISK | | |
| MFM, single-sided | \$46, \$81, \$00, sector size, last track #, #sector/track, starting track #, track offset, filler byte. | None. |
| MFM double-sided | \$66, \$81, \$00, sector size, last track #, #sector/track, starting track#, track offset, filler byte. | None. |
| | Sector size: 0 = 128 bytes/ sector 1 = 256 bytes/ sector 2 = 512 bytes/ sector | |

Table 7-8. 1571 Burst Mode Commands (*continued*)

| <i>Function</i> | <i>Byte Sequence (Prefixed by "U0" String)</i> | <i>Burst Output</i> |
|--|--|---|
| | 3 = 1024 bytes/ sector Default last track # = 39 (\$27 hex) Default starting track # = 0 Default track offset = 0 Default filler byte = \$E5 (229 decimal). | |
| GCR, double-sided, no directory or BAM | \$06,\$00, ID byte 1, ID byte#2 | None. |
| SECTOR INTERLEAVE SELECTION (multi-sector read and write only) | | |
| Set interleave value | \$08, interleave value | None. |
| Read previous value | \$88 | Returns previous value. |
| QUERY DISK FORMAT (determine GCR or MFM sector size, sectors/track) | | |
| Side 0, track 0 | \$0A | 1 burst status byte, then second burst status byte, #sectors/track, logical track#, min. sector#, max sector#, hard sector interleave. (If format is non-readable or GCR, only first [status] byte is returned.) |
| Side 1, track 0 | \$1A | (If format is non- |

Table 7-8. 1571 Burst Mode Commands (*continued*)

| <i>Function</i> | <i>Byte Sequence (Prefixed by "U0" String)</i> | <i>Burst Output</i> |
|--|--|---|
| | | readable or GCR, only first [status] byte is returned.) |
| Side 0, track n | \$8A, n | (If format is non- readable or GCR, only first [status] byte is returned.) |
| Side 1, track n | \$9A, n | (If format is non- readable or GCR, only first [status] byte is returned.) |
| INQUIRE STATUS (check drive status or load status register) | | |
| Log in disk, update status | \$4C, new status | None. |
| Check previous status | \$8C | Last I/O status. |
| Check if disk logged | \$CC | Returns old status. Error flagged if disk not logged. |
| FASTLOAD (Quick loading of entire GCR file. Wildcards required for reliable operation.) | | |
| SEQ file | \$9F, "filename", "*" | 1 status byte followed by 254 data bytes per sec- tor. (Last sector: status=\$1F, next byte=# of bytes remaining. Last data bytes of the file follow. |
| PRG file | \$1F, "filename", "*" | 1 status byte followed by 254 data bytes per sec- tor. (Last sector: status=\$1F, next |

Table 7-8. 1571 Burst Mode Commands (*continued*)

| <i>Function</i> | <i>Byte Sequence (Prefixed by "U0" String)</i> | <i>Burst Output</i> byte=# of bytes remaining. Last data bytes of the file follow. |
|---|--|--|
| CHGUTL Utilities (CHGUTL functions not actually burst mode operations) | | |
| Sector interleave | \$3E, \$53, interleave value | None returned. |
| # of retries if error occurs | \$3E, \$52, # of retries | None returned. |
| ROM checksum | \$3E, \$54 | Green lamp blinks four times if checksum wrong. |
| Mode select | \$3E, \$4D, \$30 (1541 mode) \$3E, \$4D, \$31 (1571 mode) | None returned. |
| Side select | \$3E, \$48, \$30 (side = 0) \$3E, \$48, \$31 (side = 1) | None returned. |
| Device # change | \$3E, device# (device number range: 4-30) | None returned. |

The 1571's burst mode commands include several commands that do not require any burst data transfer. These routines either format disks or change various 1571 disk drive parameters (that is, device number, interleave values, and so on). Other burst mode commands require data to be transferred between the C-128 and the 1571 using special routines.

Before a burst mode command is executed, the appropriate DOS burst command string (always prefaced by the two-character string "U0") is sent over the 1571's command channel. The CIA#2 chip (at \$DD00) is set up, and then the data is read in block form. In other words, data transfers are accomplished by sending 128-, 256-, 512-, or 1024-byte data blocks instead of sending individual characters. After the data has been transferred, normal processing resumes. Note that

interrupts must be disabled while a burst transfer routine is being executed.

Software intended for commercial use should test whether or not a 1571 fast disk drive is part of the system, or whether it is in 1541 (slow) or 1571 (fast) modes. Before any burst mode initialization occurs, a simple

```
OPEN 15,8,15, "IO": CLOSE 15
```

statement — or a machine language equivalent — will set or clear bit #6 of location \$0A1C, depending on the current mode of operation. If bit #6 is set after an OPEN call (or after any other ordinary disk activity, such as a LOAD or SAVE operation), the device will support burst data communications.

Once the burst command (read sector, write sector, and so on) has been sent through the disk's command channel (channel #15), interrupts are disabled with an SEI instruction and CIA #1's interrupt control register (often called the ICR) is cleared simply by reading it. Finally, bit #4 of the CIA #2 I/O port at \$DD00 is toggled by EORing with a set bit.

Receiving data in burst mode is only slightly more complicated. Bit #3 of CIA #1's ICR is tested to determine when a byte has been assembled from its serially received bits and is ready to be read. The receive subroutine will loop until this bit is set; afterwards, the handshake line is toggled again by EORing bit #4 of the CIA #2 I/O port (\$DD00). Finally, the byte can be read by fetching the contents of the serial data port located in CIA #1 at \$DC0C.

Example assembly code for this routine can be seen in the "RDBYTT" section of Figure 7-3. This code is usually included with some type of indexed loop so that a loading-type operation takes place. It may only be necessary to read a few bytes — usually no more than five — if drive status or disk parameters are being fetched, using the QUERY DISK FORMAT or INQUIRE STATUS commands.

```

INIT   SEI           ; disable interrupts
       LDA $DCDD    ; clear CIA#1 Interrupt Control Register by use of
                   ; read operation. Result discarded.
TOGGL  LDA $DD00    ; read CIA#2 I/O port.
       EOR #$10     ; toggle bit #4 (serial clock/handshake line.)
       STA $DD00
       RTS          ; Return to caller.

RDBYT  LDA #$08     ; select bit #3
RLOOP  BIT $DCDD    ; of ICR and wait for it to go high. Routine
                   ; is waiting for indication of byte ready to read.
       BEQ RLOOP    ; Loop if waiting.
       JSR TOGGL    ; Toggle clock/handshake line.
       LDA $DCDC    ; Get byte sent from serial port.
       RTS          ; Return to caller.

STOBYT STA ($FC),X ; Store byte into current memory bank. $FC is
                   ; zero-page pointer preset to desired "Load address".
       INX          ; Increment counter.
       RTS          ; Return to caller.

CKFMT  LDY #$13     ; QUERY DISK output parameter buffer at $1300.
       LDX #$00
       STY $FD
       STX $FC
       SEI          ; disable interrupts.
       LDX #$00     ; zero counter register.
       LDA $DCDD    ; Clear CIA#2 ICR.
       JSR TOGGL    ; Toggle clock line.

       JSR RDBYT    ; Read byte from serial bus.
       JSR STOBYT   ; Store byte to load area.
       AND #$0E     ; Mask out high nybble & low bit of status.
       BNE OUT      ; Not zero, error occurred; so exit loop.
       JSR RDBYT    ; Get MFM disk status #2.
       JSR STOBYT   ; and store to parameter buffer area.
       AND #$0E
       BNE OUT      ; Exit because of MFM error.

LOOP2  LDY #$04     ; Fetch 5 parameters in loop.
       JSR RDBYT    ; Read parameters from 1571.
       JSR STOBYT   ; Store to parameter buffer.
       DEY          ; Decrease counter.
       BPL LOOP2    ; Done five loops yet?

OUT    CLI          ; Clean up, & re-enable interrupts.
       JSR $FFCC    ; Call Kernal CLRCHN.
       RTS          ; Return to caller. Five MFM disk status
                   ; bytes at $1300.

```

Figure 7-3. Library of burst transfer routines

The CKFMT routine in Figure 7-3 reads five parameters from the drive, indicating the MFM (non-Commodore) disk's number of sectors per track, logical track number, minimum and maximum sector

numbers, and MFM interleave values. These values are stored to a disk "parameter buffer" at \$1300 and can be used for the MFM read and write routines.

It should be noted that the disk must be logged in using the INQUIRE DISK command (see Table 7-8 for its description and necessary parameters) before any sector can be read. Like the burst read or write commands themselves, the INQUIRE DISK command can be sent from a BASIC open statement. The returned burst status byte, however, must be read from a machine language routine.

Ordinary files that exist on normal Commodore diskettes can take advantage of the FASTLOAD mode. FASTLOAD can load either sequential or program files, and it operates at full burst mode speed. Again, bytes are fetched using the RDBYT routine shown in Figure 7-2. The OPEN statement used to enable the FASTLOAD routine is

```
OPEN15,8,15,"UO"+CHR$(Q)+"filename string"+"*".
```

The proper value of Q depends upon whether a PRG or SEQ file is to be loaded. Q should be 159 (\$9F hex) if a sequential file is to be loaded and should be set to 31 if an ordinary PRG (program) file will be loaded. FASTLOAD ignores the ordinary load address of a PRG file and passes its load address characters just like the data bytes of a file. A routine that FASTLOADs a file reads only 254 data bytes from the serial port for every sector transferred; this is because the first two bytes of every sector of a normal Commodore disk file contain track and sector link bytes that point to the next sector. Consequently, only 255 bytes need to be fetched from the serial port for every sector FASTLOADed—the first byte is the normal burst status byte, followed by the 254 data bytes.

The burst status byte should be tested every time a sector is transferred. A value of 0 or 1 in the status byte indicates that it is safe to transfer the 254 data bytes, while a value of 3 indicates that the desired file has not been found and the FASTLOAD routine should be aborted. A value of 31 indicates that the subsequent data bytes will

be returned from the last sector of the file. Thus, the byte returned after the status byte indicates how many bytes need to be transferred. This byte is followed by the necessary number of data bytes read from the last sector.

The arrangement of the burst status byte is shown in Table 7-9.

Various disk formats can be created using the burst FORMAT DISK commands. They can create either GCR or MFM disk formats. The GCR burst format command is merely an ordinary Commodore 1571 double-sided disk format that has no directory or BAM information written to the disk. The MFM formats let the user have greater control over the various format parameters. For example, the statement

```
OPEN 15,8,15, "UD"+CHR$(102)+CHR$(129)+CHR$(0)+CHR$(512)+
CHR$(39)+CHR$(8)
```

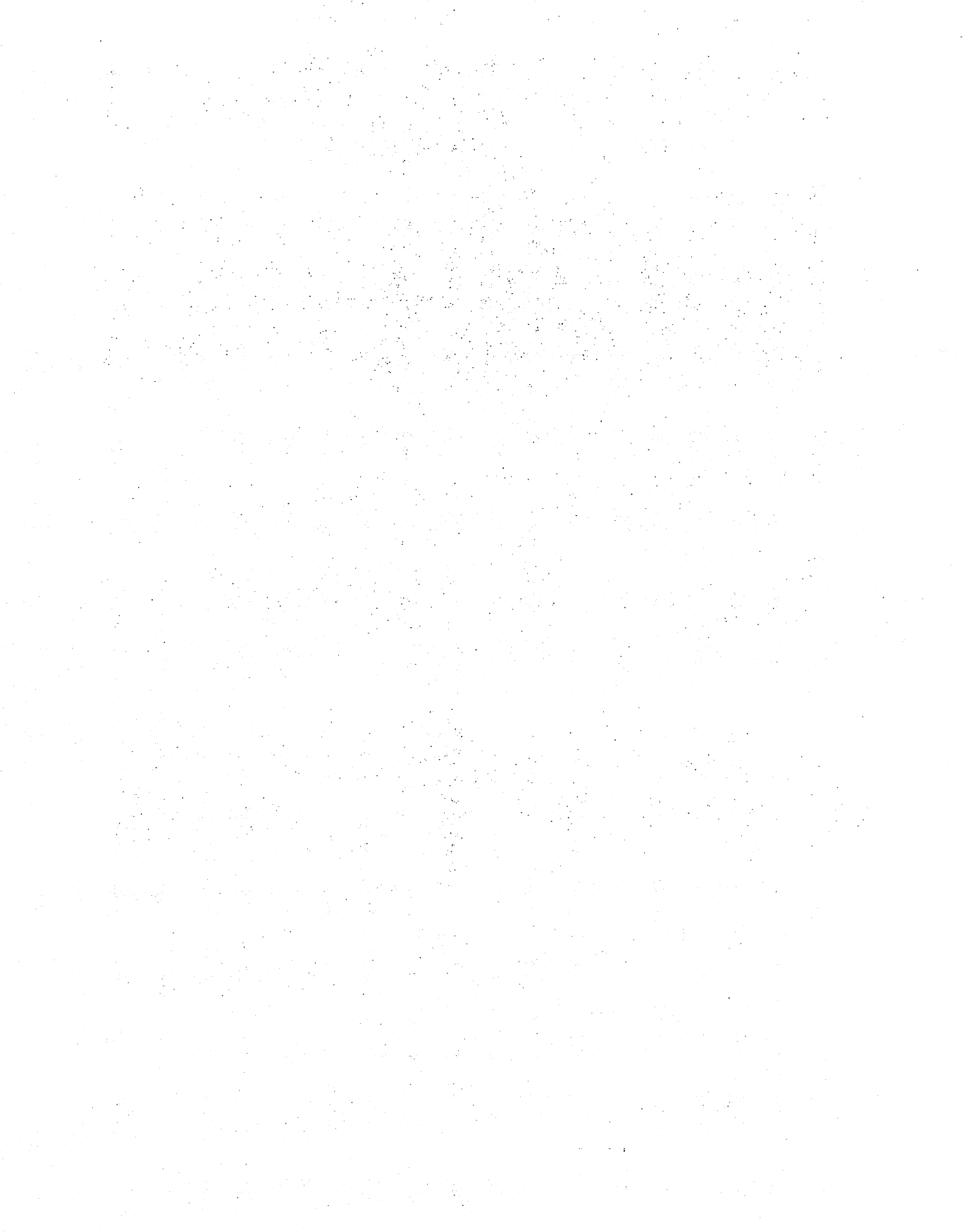
formats an IBM CP/M-86 or MS-DOS diskette, although it writes no directory information to it. Refer to Table 7-8 for descriptions of the various GCR and MFM disk format parameters.

Table 7-9. Burst Status Byte

| | | | | | | | | |
|-----------|------|----|-----|-------------|-----|-------------------|-------|---|
| Bit#: | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Function: | Mode | DN | ___ | sector size | ___ | controller status | _____ | |

Mode bit: 1 = MFM disk, 0 = Commodore GCR disk.
 DN = device number bit. DN = 0 for device #0 (default for the 1571).
 Sector size: 0 = 128 bytes/sector
 1 = 256 bytes/sector
 0 = 512 bytes/sector
 0 = 1024 bytes/sector.

Controller status nybble: \$00 or \$01—no error.
 \$01-\$0F—various read, write, or I/O errors.



Part III

The CP/M Operating System

The Commodore 128 can use a third operating system, called CP/M 3.0 or CP/M Plus. This system is an improved version of the earlier CP/M version 2.2, which has itself been implemented on a wide variety of microcomputers—Osbornes, Kaypros, Morrows, Apples, and many others. This section describes some of the unique features of CP/M 3.0 as implemented on the C-128.

Chapter 8

CP/M on The Commodore 128 System

CP/M Plus on the C-128 helps you take advantage of the system's unique features. For example, when the 1571 disk drive is being used, CP/M can read four industry-standard disk formats, in addition to the three Commodore CP/M disk formats. And because the system contains 128K of RAM memory, the CCP (console command processor, the program that acts as CP/M's "user interface" by interpreting commands, drive specifiers, wildcard characters, and so on) can always be stored in this extra RAM. You can therefore avoid having to

reload the CCP program whenever a utility, application, or transient command program's execution terminates.

Because of the uniqueness of the C-128's architecture, a dual-processor scheme is utilized. CP/M normally runs under an 8080, an 8085, or a Z-80 microprocessor. C-128 CP/M is no exception to this rule, as it utilizes a Z-80 processor. What makes the C-128's implementation unique is that certain I/O (BDOS/BIOS) functions actually run under the 8502 processor. While not really affecting compatibility with standard CP/M software, programmers should understand the activities carried out by each processor. Although this chapter will deal specifically with the C-128 implementation of CP/M Plus, programmers should obtain the Digital Research CP/M Plus User's Guide/Programmer's Guide/System Guide trilogy that covers "generic" CP/M Plus in detail.

The C-128 implementation of CP/M Plus does not require the use of any additional cartridge or other attachments. In fact, the old CP/M cartridge designed for the C-64 should *not* be used on the C-128.

Booting CP/M Plus

When a C-128-formatted CP/M disk is inserted into the disk drive and the system is reset (while in C-128 or C-64 mode), the system activates the Z-80 processor. In turn, the CP/M operating system is loaded. The CP/M disk must contain the files CP/M+.SYS to boot the system.

You learned in Chapter 1 that the Z-80 really has initial control of the C-128 system upon power-up. Besides searching for the presence of C-64 cartridges — or detecting whether or not the COMMODORE key has been held down to switch to C-64 mode — the Z-80 also downloads a small switchover routine to RAM bank 0. This code, which runs under control of the 8502 processor, configures memory properly and enables the Z-80 again for CP/M usage. Figure 8-1 lists this routine; it is called by the autoboot routine contained on C-128

```
$FFD0 SEI      ; disable IRQs before switch to Z-80 processor
      LDA #$3E ; select RAM Bank 0, no ROM, no I/O block.
      STA $FFD0 ; and store MMU data to Configuration Register.
      LDA #$80 ; Select Z-80 processor, hold EXROM and GAME high.
      STA $D505 ; Store data to Mode Configuration Register.
```

Figure 8-1. RAM-resident CP/M start-up code

CP/M disks. Initially, the 8502-based CP/M BIOS is loaded into RAM. Later, after memory has been configured for CP/M Plus, control will be passed to the Z-80, and execution begins at location \$0000 in bank 0.

Memory Usage By the CP/M Plus Operating System

Because the C-128 uses two processors — both the Z-80 itself and the 8502 — to run CP/M, there are really two memory maps for the CP/M Plus operating system.

Most low-level I/O is actually handled by the 8502 processor in the BIOS 8502 module, with the exception of 80- and 40-column video display updating. However, the ordinary CP/M BIOS tasks numbered 0 through 29 (decimal) are callable while using the Z-80 processor. For a complete list of these BIOS functions, please consult the *C-128 Programmer's Reference Guide*.

To call 8502-dependent BIOS routines, C-128 programmers should call CP/M BIOS routine #30, which invokes CP/M's "user functions." User function #4 allows access to the hardware-dependent C-128 8502 BIOS routines. One of the main functions of the 8502 BIOS is 1541/1571 disk I/O handling. Listings and descriptions of these system-dependent routines can be found on pages 695 through

701 of the *C-128 Programmer's Reference Guide*.

Figure 8-2 shows the C-128's CP/M Plus memory map that is visible to the Z-80 processor, while Figure 8-3 shows the memory that the 8502 can access.

A unique feature of CP/M Plus is the storage of the CCP (console command processor) in RAM. Whenever a transient command or application program terminates execution and returns to the

| <i>Addr.</i> | <i>Bank 0</i> | <i>Bank 1</i> | <i>Bank 2</i> | <i>Bank 3</i> |
|---------------|---|--|--|---------------|
| \$FFFF | Reserved for 8502 and MMU registers | | | |
| \$FF00 | Common BIOS (Basic Input/Output System), BDOS (Basic Disk Operating System), and SCB (System Control Block) | | | |
| \$E000 | Banked BIOS and BDOS | 59K of TPA (Transient program area) | 127K RAM disk (available in future 256K systems) | |
| \$2800 | CCP block #2 | | | |
| \$1800 | VIC-II text screen (40-column) | | | |
| \$1000 | CP/M banked ROM code | | | |
| \$0400 | | CP/M's page zero | | |
| \$0100 | | | | |
| \$0000 | | | | |

Figure 8-2. Z-80 memory map for CP/M Plus on the C-128

| <i>Addr.</i> | <i>Bank 0</i> | <i>Bank 1</i> | <i>Bank 2</i> | <i>Bank 3</i> | |
|---------------|---|-------------------------------------|--|---------------|--|
| \$FFFF | MMU Registers (CR, PCRA PCRD) | | | | |
| \$FF00 | 8502 Kernal ROM | CCP block #1 temporary storage | 127K of RAM disk for future, expanded systems. | | |
| \$E000 | 4K Block, I/O devices (VIC, SID, 8563, etc.) | 59K of TPA (transient program area) | | | |
| \$D000 | More 8502 Kernal ROM | | | | |
| \$C000 | Banked BDOS and BIOS | | | | |
| \$2800 | CCP, block #2 | | | | |
| \$1800 | VIC-II 40-column text screen | | | | |
| \$1000 | Disk buffer | | | | |
| \$0F00 | 8502 BIOS routines | | | | |
| \$0400 | Kernal RAM (common RAM, for stack, vectors, etc.) | | | | |
| \$0100 | 8502 zero-page RAM | | | | |
| \$0000 | | | | | |

Figure 8-3. 8502 memory map for CP/M Plus on the C-128

operating system, an ordinary CP/M operating system must reload a "CCP.COM" file from disk, consuming time. The C-128's CCP is stored in two parts in RAM and is copied by the Z-80 to the bottom of the TPA (transient program area) whenever a CP/M warm boot occurs (upon termination of a program or when the operating system receives a CTRL-C keystroke from the console).

Current C-128 systems are supplied with 128K of RAM. Future systems that may have 256K or more of RAM can use the extra 128K as a high-speed *RAM disk*. Note that this extra RAM in a future C-128 system would actually be part of the system's memory map; it is not accessed using DMA-controller commands, as are the RAM expansion cartridges available for the current C-128 system.

Disks and C-128 CP/M Plus

The C-128 can use several different floppy disk formats when connected to the 1571 disk drive. They can be divided into two distinct groups: GCR (group code recorded) and MFM (modified frequency modulated). All ordinary Commodore diskettes are of the GCR type. The C-128 CP/M with a 1571 drive is thus able to read the three types of Commodore CP/M disks. They are as follows:

- *Commodore 64 CP/M 2.2 Format.* C-64 CP/M 2.2 format is a single-sided disk format that was used with the C-64 CP/M cartridge. This format has been included solely for the sake of compatibility, since the other two GCR formats are better suited to CP/M usage. A C-64 CP/M 2.2 disk can store 139,264 bytes of data, excluding boot information. (This format can be read on the old 1541 disk drive.)
- *Commodore 128 CP/M 3.0 Single-Sided Format.* This disk, like that for C-64 CP/M 2.2, can be read on the old 1541 disk

drive. However, it also stores data on sectors 17 through 20 on every track—sectors the old C-64 CP/M 2.2 did not use. Thus, a C-128 single-sided CP/M 3.0 diskette can store an extra 22,016 bytes of data, independent of boot information.

- *Commodore C-128 CP/M 3.0 Double-Sided Format.* This diskette format can only be used with the new 1571 disk drive. (The user may be able to read a file written on a double-sided CP/M disk with the old 1541 drive if the file was one of the first ones to be created on the disk, and doesn't occupy any sectors on the other side of the disk.) This format can store 348,160 bytes of data.

All three diskette formats can be read by C-128 CP/M Plus without any special effort—the operating system recognizes these disk formats whenever it logs in the diskette. The **FORMAT** utility, included with the CP/M system when a C-128 is purchased, allows you to format a disk in any of these three formats. Commercial software developed especially for the C-128's CP/M Plus operating mode is usually distributed in C-128 single-sided format, although many Commodore user groups distribute Commodore CP/M software in the older C-64 format simply for the sake of universality.

What makes the C-128 a unique machine (when coupled with its companion 1571 disk drive) is that it can read many industry-standard diskette formats. The current implementations of CP/M 3.0 on the C-128 allow four standard MFM disk formats to be used, in addition to the three GCR formats just mentioned. Table 8-1 lists these disk types and their associated parameters.

When using an MFM diskette with one of these formats, a small window will be displayed in the lower left of the C-128 video display. You are prompted to scroll through the various choices of diskette formats using the right and left cursor keys; pressing **RETURN** selects that particular format, while striking **CTRL-RETURN** “locks” that particular format. When locked, all diskettes inserted into the 1571 are assumed to have been created in that selected format.

Table 8-1. MFM Disk Formats Currently Supported by C-128 CP/M Plus

| <i>Manufacturer</i> | <i>Number of Sides</i> | <i>Comments</i> |
|-----------------------------|------------------------|---|
| IBM | 1 | 512 bytes/sector, 8 sectors/track (CP/M-86 on the PC) |
| IBM | 2 | 512 bytes/sector, 8 sectors/track (CP/M-86 on the PC) |
| Kaypro II | 1 | 512 bytes/sector, 10 sectors/track |
| Kaypro IV | 2 | 512 bytes/sector, 10 sectors/track |
| Osborne (double-density) | 1 | 1024 bytes/sector, 5 sectors/track |
| Epson QX-10 | 2 | 512 bytes/sector, 10 sectors/track |

Transferring Files Between CP/M And Other Modes

The structure of a C-128 CP/M diskette is quite different from that of an ordinary Commodore disk. File transfers between C-64/C-128 modes and CP/M Plus can be quite cumbersome. Some public domain utilities can read a file from a C-64/C-128 diskette and write it, sector by sector, to a CP/M disk. Unfortunately, this method is quite slow for large files.

A better way is to BLOAD the file into bank 1 RAM (past \$1C00). As long as the file is under 205 blocks long (approximately 55K), it will fit in RAM. Since booting the CP/M operating system

does not destroy much of RAM bank 1's contents, you can switch to CP/M and later use the SAVE command to put your file onto a CP/M disk. The following steps describe how to transfer an ordinary Commodore file (which much be of PRG type) to a CP/M file.

1. Load the file into RAM bank 1 at \$1C00 using the BLOAD command (BLOAD "*filename*", B1, P7168).
2. Use the ML monitor to look through bank #1's contents to find the end of your file. Record the ending address (in hex).
3. Boot the CP/M Plus operating system (insert a CP/M Plus disk that has the CPM+.SYS, CCP.COM, and SAVE.COM programs stored on it).
4. Once CP/M has been booted, type SAVE. This loads the SAVE program into memory. Type SAVE again, and enter the desired CP/M filename at the prompt. When prompted for the starting address of the range of memory to be saved to a CP/M diskette, enter 1C00. SAVE will prompt for the ending address. Enter the hex address that you recorded in step 3.

This technique works only with program (PRG) files that can actually be loaded into RAM. Sequential (SEQ) files will have to be converted to PRG files; this task can be accomplished with any standard Commodore disk editor. (See Chapter 7, "Disk and I/O Operations," for more details.) Additionally, the first two bytes of a sequential file will be lost in the conversion because they are regarded as PRG file "load address" bytes. For this reason, it's much easier to create the file with two extra bytes at its beginning. This file transfer is quite easy to accomplish once the user becomes familiar with the method.

Revisions to the CP/M Plus System

Commodore has released two versions of the CP/M Plus operating system for the C-128. The first of these, dated August 1985, does not support the RS-232 modem port. It is also quite slow in some operations.

For this reason, a second version was released in December 1985, along with two utility programs, CONF.COM and C1571.COM, and a help file called CONF.HLP. The newer CP/M operating system can support modem port usage, letting the user use one of the many CP/M-based telecommunications programs available. The DEVICE command is used to configure the modem port. DEVICE allows the redefinition and redirection of logical and physical devices. In this case, the AUXIN: and AUXOUT: logical devices should be redefined as RS-232 channels instead of "null" devices. The following sequence accomplishes this task:

```
DEVICE AUX:=RS-232(XON,300)
```

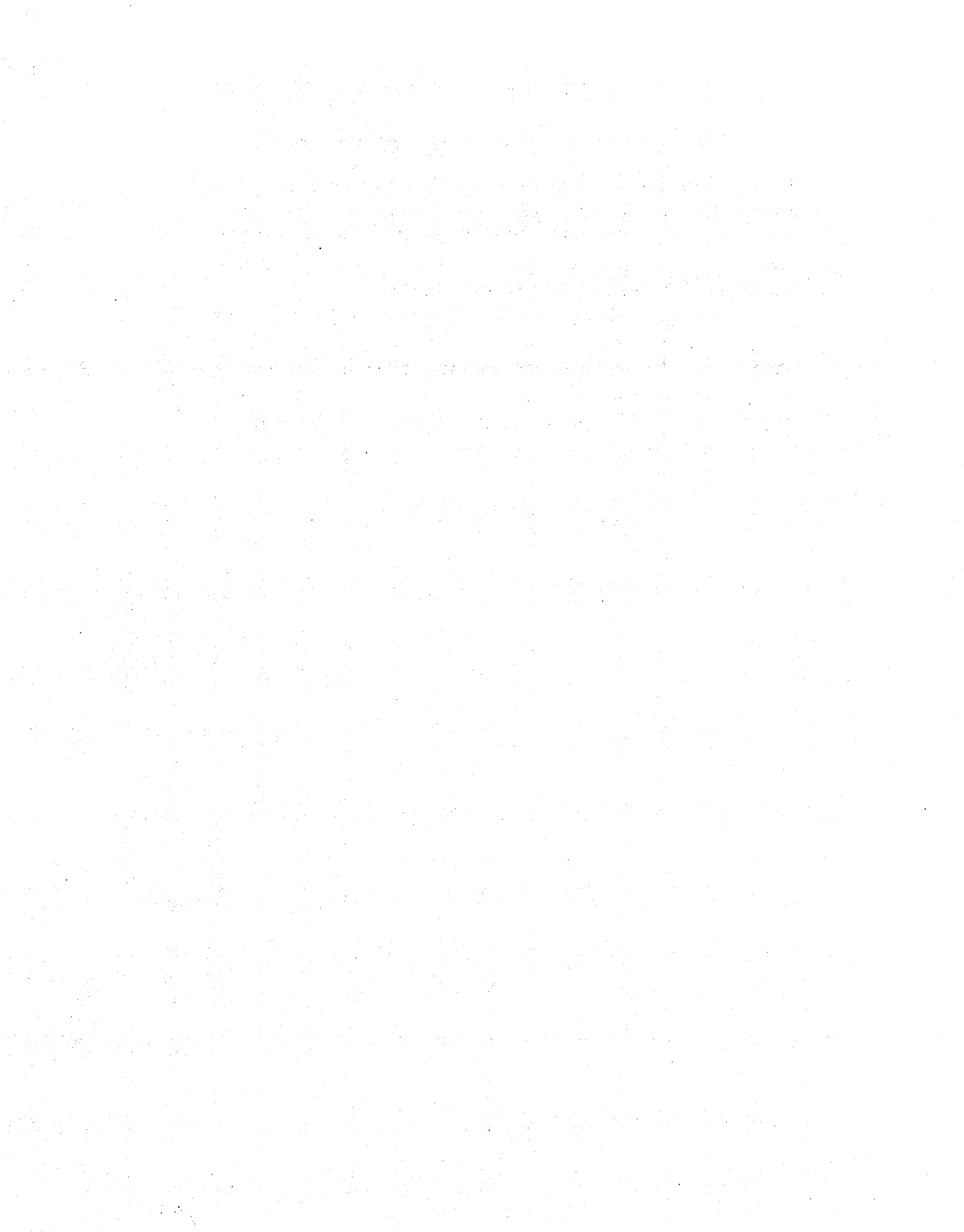
Both AUXIN: and AUXOUT: can be redefined at the same time this way. Additionally, the baud rate of the modem port (300) can be changed to suit the user's needs; it can function reliably up to 1200 bits per second, although speedy baud rates slow down the keyboard scanning rate, decreasing a fast typist's efficiency.

The CONF.COM utility program configures various C-128 CP/M system parameters, as described in the CONF.HLP file. For example, the RS-232 baud rate can be set by typing **CONF BAUD=1200**. **CONF VOL=15** turns up the sound generator's volume to its maximum level, while **CONF.MAP** displays a small ASCII keyboard table. A **POKE** command is also available from CONF.COM.

The C1571 utility program disables the 1571's verify-after-write function. Every time a sector is written to a disk, the 1571 verifies that

the data was correctly written. C1571.COM bypasses this safety measure for much faster disk I/O— with the slight risk of losing some data. It is activated by typing **C1571** (*drivelist*). For example, C1571 (A,B) disables the verify-after-write function on 1571 disk drives A: and B:. C1571, of course, only works under CP/M.

Disabling the 1571's verify-after-write can be risky, though. As a compromise between speed and data integrity, the [V] (verify) option should be used after all file transfers accomplished using the PIP utility program. This optional parameter for PIP is not really necessary when the 1571 is in its normal (i.e., write-after-verify) mode.



Appendix A

DIGIFONT: A New C-128 Character Set

This memory dump contains the character definitions for a C-128 character set called DIGIFONT. This data can be entered and saved using the C-128's built-in monitor program. The definitions must be loaded into the 8563's 16K of video RAM using the special loader program described in Figure 5-5. This character set can also be used "as-is" for the VIC-II video chip, providing pointers are set up properly.

```

>0A000 00 7E 66 6E EE EO E6 FE 00 7E 66 66 FE E6 E6 E6
>0A010 00 FC CC CC FE C6 C6 FE 00 7E 66 60 EO EO E6 FE
>0A020 00 FC CC CC CE CE CE FE 00 7E 60 60 F8 EO EO FE
>0A030 00 7E 60 60 F8 EO EO EO 00 7E 60 60 EE E6 E6 FE
>0A040 00 66 66 66 FE E6 E6 E6 00 18 18 18 38 38 38 38
>0A050 00 3E 0C 0C 0E CE CE FE 00 66 6C 78 F0 F8 EC E6
>0A060 00 60 60 60 EO EO EO FE 00 FE DA DA DA DA DA DA
>0A070 00 7C 66 66 E6 E6 E6 00 7E 66 66 E6 E6 E6 FE
>0A080 00 FC CC CC FC EO EO EO 00 7E 66 66 E6 E6 FE 07
>0A090 00 FC CC CC FC CE CE CE 00 FE C0 C0 FE OE OE FE
>0A0A0 00 7E 18 18 38 38 38 38 00 66 66 66 E6 E6 FE
>0A0B0 00 66 66 66 66 66 3C 18 00 C6 C6 C6 D6 FE EE
>0A0C0 00 CC CC 78 30 78 CC CC 00 66 66 3C 18 18 18 18
>0A0D0 00 FC 0C 18 30 60 C0 FC 00 3C 30 30 30 30 3C
>0A0E0 00 00 C0 60 30 18 0C 06 00 3C 0C 0C 0C 0C 3C
>0A0F0 00 18 3C 66 00 00 00 00 00 00 00 00 00 00 FF
>0A100 00 00 00 00 00 00 00 00 00 18 3C 3C 18 00 00 18
>0A110 00 66 66 66 00 00 00 00 00 66 66 FF 66 FF 66 66
>0A120 00 18 3E 60 3C 06 7C 18 00 62 66 0C 18 30 66 46
>0A130 00 3C 66 3C 38 67 66 3F 00 06 0C 18 00 00 00 00
>0A140 00 0C 18 30 30 30 18 0C 00 30 18 0C 0C 0C 18 30
>0A150 00 00 66 3C FF 3C 66 00 00 00 18 18 7E 18 18 00
>0A160 00 00 00 00 00 18 18 30 00 00 00 00 7E 00 00 00
>0A170 00 00 00 00 00 00 18 18 00 00 06 0C 18 30 60 C0
>0A180 00 7E 66 66 E6 E6 E6 FE 00 18 38 18 1C 1C 1C 7E
>0A190 00 FE 06 06 FE EO EO FE 00 FC 0C 0C 7E OE OE FE
>0A1A0 00 CC CC CC FE OE OE OE 00 FE C0 C0 FE OE CE FE
>0A1B0 00 7E 66 60 FE E6 E6 FE 00 FE C6 06 OE OE OE OE
>0A1C0 00 7E 66 66 FE C6 C6 FE 00 FC CC CC FE OE OE OE
>0A1D0 00 00 00 18 00 00 18 00 00 00 00 18 30
>0A1E0 00 0E 18 30 60 30 18 0E 00 00 00 7E 00 7E 00 00
>0A1F0 00 70 18 0C 06 0C 18 70 00 3C 66 06 0C 18 00 18
>0A200 00 60 30 18 00 00 00 00 00 08 1C 3E 7F 7F 1C 3E
>0A210 18 18 18 18 18 18 18 18 00 00 00 FF 00 00 00 00
>0A220 00 00 FF 00 00 00 00 00 00 FF 00 00 00 00 00 00
>0A230 00 00 00 00 00 FF 00 00 30 30 30 30 30 30 30
>0A240 0C 0C 0C 0C 0C 0C 0C 0C 00 00 00 EO 30 18 18 18
>0A250 18 18 0C 07 00 00 00 00 18 18 30 EO 00 00 00 00
>0A260 C0 C0 C0 C0 C0 C0 C0 FF 80 C0 60 30 18 0C 06 03
>0A270 03 06 0C 18 30 60 C0 80 FF C0 C0 C0 C0 C0 C0
>0A280 FF 03 03 03 03 03 03 03 00 3C 7E 7E 7E 7E 3C 00
>0A290 00 00 00 00 00 00 00 FF 00 00 36 7F 7F 7F 3E 1C 08
>0A2A0 60 60 60 60 60 60 60 60 00 00 00 07 0C 18 18 18
>0A2B0 81 42 24 18 18 24 42 81 00 3C 66 42 42 66 3C 00
>0A2C0 00 18 18 66 66 18 18 3C 06 06 06 06 06 06 06
>0A2D0 00 08 1C 3E 7F 3E 1C 08 18 18 18 FF 18 18 18 18
>0A2E0 C0 C0 30 30 C0 C0 30 30 18 18 18 18 18 18 18 18
>0A2F0 00 00 00 73 CE 00 00 00 FF 7F 3F 1F 0F 07 03 01
>0A300 00 00 00 00 00 00 00 00 F0 F0 F0 F0 F0 F0 F0
>0A310 00 00 00 00 FF FF FF FF FF 00 00 00 00 00 00
>0A320 00 00 00 00 00 00 00 FF C0 C0 C0 C0 C0 C0 C0
>0A330 CC CC 33 33 CC CC 33 33 03 03 03 03 03 03 03
>0A340 00 00 00 00 CC CC 33 33 FF FE FC F8 F0 EO C0 80
>0A350 03 03 03 03 03 03 03 03 18 18 18 1F 18 18 18 18

```

```

>0A360 00 00 00 00 0F 0F 0F 0F 18 18 18 1F 00 00 00 00
>0A370 00 00 00 F8 18 18 18 18 00 00 00 00 00 00 FF FF
>0A380 00 00 00 1F 18 18 18 18 18 18 18 FF 00 00 00 00
>0A390 00 00 00 FF 18 18 18 18 18 18 18 18 18 18 18 18
>0A3A0 C0 C0 C0 C0 C0 C0 C0 C0 E0 E0 E0 E0 E0 E0 E0 E0
>0A3B0 07 07 07 07 07 07 07 07 FF FF 00 00 00 00 00 00
>0A3C0 FF FF FF 00 00 00 00 00 00 00 00 00 00 FF FF FF
>0A3D0 03 03 03 03 03 03 03 FF 00 00 00 00 F0 F0 F0 F0
>0A3E0 0F 0F 0F 0F 00 00 00 00 18 18 18 F8 00 00 00 00
>0A3F0 F0 F0 F0 F0 00 00 00 00 F0 F0 F0 F0 0F 0F 0F 0F
>0A400 00 78 CF CE C0 CE CF 78 FF 81 99 99 01 19 19 19
>0A410 FF 03 33 33 01 39 39 01 FF 81 99 9F 1F 1F 19 01
>0A420 FF 03 33 33 31 31 31 01 FF 81 9F 9F 07 1F 1F 01
>0A430 FF 81 9F 9F 07 1F 1F 1F FF 81 9F 9F 11 19 19 01
>0A440 FF 99 99 99 01 19 19 19 FF E7 E7 E7 C7 C7 C7 C7
>0A450 FF C1 F3 F3 F1 31 31 01 FF 99 93 87 0F 07 13 19
>0A460 FF 9F 9F 9F 1F 1F 1F 01 FF 01 25 25 25 25 25 25
>0A470 FF 83 99 99 19 19 19 19 FF 81 99 99 19 19 19 01
>0A480 FF 03 33 33 03 1F 1F 1F FF 81 99 99 19 01 F8
>0A490 FF 03 33 33 03 31 31 31 FF 01 3F 3F 01 F1 F1 01
>0A4A0 FF 81 E7 E7 C7 C7 C7 C7 FF 99 99 99 19 19 19 01
>0A4B0 FF 99 99 99 99 99 C3 E7 FF 39 39 39 29 01 11
>0A4C0 FF 33 33 87 CF 87 33 33 FF 99 99 C3 E7 E7 E7 E7
>0A4D0 FF 03 F3 E7 CF 9F 3F 03 FF C3 CF CF CF CF C3
>0A4E0 FF FF 3F 9F CF E7 F3 F9 FF C3 F3 F3 F3 F3 C3
>0A4F0 FF E7 C3 99 FF FF FF FF FF FF FF FF FF FF 00
>0A500 FF FF FF FF FF FF FF FF FF E7 C3 C3 E7 FF FF E7
>0A510 FF 99 99 99 FF FF FF FF FF 99 99 00 99 00 99 99
>0A520 FF E7 C1 9F C3 F9 83 E7 FF 9D 99 F3 E7 CF 99 B9
>0A530 FF C3 99 C3 C7 98 99 C0 FF F9 F3 E7 FF FF FF FF
>0A540 FF F3 E7 CF CF CF E7 F3 FF CF E7 F3 F3 F3 E7 GF
>0A550 FF FF 99 C3 00 C3 99 FF FF FF E7 E7 81 E7 E7 FF
>0A560 FF FF FF FF E7 E7 CF FF FF FF FF 81 FF FF FF
>0A570 FF FF FF FF FF FF E7 E7 FF FF F9 F3 E7 CF 9F 3F
>0A580 FF 81 99 99 19 19 19 01 FF E7 C7 E7 E3 E3 E3 81
>0A590 FF 01 F9 F9 01 1F 1F 01 FF 03 F3 F3 81 F1 F1 01
>0A5A0 FF 33 33 33 01 F1 F1 F1 FF 01 3F 3F 01 F1 31 01
>0A5B0 FF 81 99 9F 01 19 19 01 FF 01 39 F9 F1 F1 F1 F1
>0A5C0 FF 81 99 99 01 39 39 01 FF 03 33 33 01 F1 F1 F1
>0A5D0 FF FF FF E7 FF FF E7 FF FF FF FF E7 FF FF E7 CF
>0A5E0 FF F1 E7 CF 9F CF E7 F1 FF FF FF 81 FF 81 FF FF
>0A5F0 FF 8F E7 F3 F9 F3 E7 8F FF C3 99 F9 F3 E7 FF E7
>0A600 FF 9F CF E7 FF FF FF FF FF F7 E3 C1 80 80 E3 C1
>0A610 E7 E7 E7 E7 E7 E7 E7 FF FF 00 FF FF FF FF FF
>0A620 FF FF 00 FF FF FF FF FF 00 FF FF FF FF FF FF
>0A630 FF FF FF FF 00 FF FF CF CF CF CF CF CF CF CF
>0A640 F3 F3 F3 F3 F3 F3 F3 FF FF 1F CF E7 E7 E7
>0A650 E7 E7 F3 F8 FF FF FF FF E7 E7 CF 1F FF FF FF FF
>0A660 3F 3F 3F 3F 3F 3F 3F 00 7F 3F 9F CF E7 F3 F9 FC
>0A670 FC F9 F3 E7 CF 9F 3F 7F 00 3F 3F 3F 3F 3F 3F 3F
>0A680 00 FC FC FC FC FC FC FF C3 81 81 81 81 C3 FF
>0A690 FF FF FF FF FF 00 FF FF C9 80 80 80 C1 E3 F7
>0A6A0 9F 9F 9F 9F 9F 9F 9F FF FF FF F8 F3 E7 E7 E7
>0A6B0 7E BD DB E7 E7 DB BD 7E FF C3 99 BD BD 99 C3 FF

```



```
>0A6C0 FF E7 E7 99 99 E7 E7 C3 F9 F9 F9 F9 F9 F9 F9
>0A6D0 FF F7 E3 C1 80 C1 E3 F7 FF F3 E7 E7 CF E7 E7 F3
>0A6E0 E7 E7 E7 E7 E7 E7 E7 FF CF E7 E7 F3 E7 E7 CF
>0A6F0 FF FF FF 8C 31 FF FF FF 00 80 C0 E0 F0 F8 FC FE
>0A700 FF FF FF FF FF FF FF FF 0F 0F 0F 0F 0F 0F 0F
>0A710 FF FF FF FF 00 00 00 00 00 FF FF FF FF FF FF
>0A720 FF FF FF FF FF FF FF FF 00 3F 3F 3F 3F 3F 3F
>0A730 33 33 CC CC 33 33 CC CC FC FC FC FC FC FC FC
>0A740 FF FF FF FF 33 33 CC CC 00 01 03 07 0F 1F 3F 7F
>0A750 FC FC FC FC FC FC FC FC E7 E7 E7 E7 E7 E7 E7
>0A760 FF FF FF FF F0 F0 F0 F0 E7 E7 E7 E7 E7 E7 E7
>0A770 FF FF FF 07 E7 E7 E7 E7 FF FF FF FF FF FF 00 00
>0A780 FF FF FF E0 E7 E7 E7 E7 E7 E7 E7 00 FF FF FF FF
>0A790 FF FF FF 00 E7 E7 E7 E7 E7 E7 E7 07 E7 E7 E7
>0A7A0 3F 3F 3F 3F 3F 3F 3F 1F 1F 1F 1F 1F 1F 1F 1F
>0A7B0 F8 F8 F8 F8 F8 F8 F8 F8 00 00 FF FF FF FF FF FF
>0A7C0 00 00 00 FF FF FF FF FF FF FF FF FF 00 00 00
>0A7D0 FC FC FC FC FC FC FC FC 00 FF FF FF FF 0F 0F
>0A7E0 F0 F0 F0 F0 FF FF FF FF E7 E7 E7 07 FF FF FF FF
>0A7F0 0F 0F 0F 0F FF FF FF FF 0F 0F 0F 0F F0 F0 F0
>0A800 00 7E 66 6E EE E0 E6 FE 00 00 18 3C 66 7E 66
>0A810 00 00 00 7C 66 7C 66 7C 00 00 00 3C 66 66 3C
>0A820 00 00 00 7C 66 66 66 7C 00 00 00 7E 60 7C 60 7E
>0A830 00 00 00 7E 60 78 60 60 00 00 00 7C 0C CE C6 7C
>0A840 00 00 00 66 66 7E 66 66 00 00 00 3C 18 18 3C
>0A850 00 00 00 1E 0C 0C 6C 38 00 00 00 66 6C 78 6C 66
>0A860 00 00 00 60 60 60 60 7C 00 00 00 C6 EE FE D6 C6
>0A870 00 00 00 C6 E6 F6 DE CE 00 00 00 3C 66 66 66 3C
>0A880 00 00 00 7C 66 7C 66 7C 60 00 00 00 3C 66 66 6C 3E
>0A890 00 00 00 7C 66 7C 66 66 00 00 00 7E 60 7E 06 7E
>0A8A0 00 00 00 7E 18 18 18 18 00 00 00 66 66 66 66 3C
>0A8B0 00 00 00 66 66 66 3C 18 00 00 00 C6 D6 FE 7C 6C
>0A8C0 00 00 00 66 3C 18 3C 66 00 00 00 66 66 3C 18 18
>0A8D0 00 00 00 7E 0C 18 30 7E 00 3C 30 30 30 30 30 3C
>0A8E0 00 00 C0 60 30 18 0C 06 00 3C 0C 0C 0C 0C 0C 3C
>0A8F0 00 18 3C 66 00 00 00 00 00 00 00 00 00 00 00 FF
>0A900 00 00 00 00 00 00 00 00 00 18 3C 3C 18 00 00 18
>0A910 00 66 66 66 00 00 00 00 00 66 66 FF 66 FF 66 66
>0A920 00 18 3E 60 3C 06 7C 18 00 62 66 0C 18 30 66 46
>0A930 00 3C 66 3C 38 67 66 3F 00 06 0C 18 00 00 00 00
>0A940 00 0C 18 30 30 30 18 0C 00 30 18 0C 0C 0C 18 30
>0A950 00 00 66 3C FF 3C 66 00 00 00 18 18 7E 18 18 00
>0A960 00 00 00 00 00 18 18 30 00 00 00 07 00 00 00
>0A970 00 00 00 00 00 00 18 18 00 00 06 0C 18 30 60 C0
>0A980 00 7E 66 66 E6 E6 E6 FE 00 18 38 18 1C 1C 1C 7E
>0A990 00 FE 06 06 FE E0 E0 FE 00 FC 0C 0C 7E 0E 0E FE
>0A9A0 00 CC CC CC FE 0E 0E 0E 00 FE C0 C0 FE 0E CE FE
>0A9B0 00 7E 66 60 FE E6 E6 FE 00 FE C6 06 0E 0E 0E 0E
>0A9C0 00 7E 66 66 FE C6 C6 FE 00 FC CC CC FE 0E 0E 0E
>0A9D0 00 00 00 18 00 00 18 00 00 00 00 18 00 00 18 30
>0A9E0 00 0E 18 30 60 30 18 0E 00 00 00 7E 00 7E 00 00
>0A9F0 00 70 18 0C 06 0C 18 70 00 3C 66 06 0C 18 00 18
>0AA00 00 60 30 18 00 00 00 00 00 7E 66 66 FE E6 E6 E6
>0AA10 00 FC CC CC FE C6 C6 FE 00 7E 66 60 E0 E0 E6 FE
```

```
>0AA20 00 FC CC CC CE CE CE FE 00 7E 60 60 F8 E0 E0 FE
>0AA30 00 7E 60 60 F8 E0 E0 E0 00 7E 60 60 EE E6 E6 FE
>0AA40 00 66 66 66 FE E6 E6 E6 00 18 18 18 38 38 38 38
>0AA50 00 3E 0C 0C 0E CE CE FE 00 66 6C 78 F0 F8 EC E6
>0AA60 00 60 60 60 E0 E0 E0 FE 00 FE DA DA DA DA DA DA
>0AA70 00 7C 66 66 E6 E6 E6 E6 00 7E 66 66 E6 E6 E6 FE
>0AA80 00 FC CC CC FC E0 E0 E0 00 7E 66 66 E6 E6 FE 07
>0AA90 00 FC CC CC FC CE CE CE 00 FE C0 C0 FE 0E 0E FE
>0AAA0 00 7E 18 18 38 38 38 38 00 66 66 66 E6 E6 E6 FE
>0AAB0 00 66 66 66 66 66 3C 18 00 C6 C6 C6 C6 D6 FE EE
>0AAC0 00 CC CC 78 30 78 CC CC 00 66 66 3C 18 18 18 18
>0AAD0 00 FC 0C 18 30 60 C0 FC 00 0C 18 18 30 18 18 0C
>0AAE0 18 18 18 18 18 18 18 18 00 30 18 18 0C 18 18 30
>0AAF0 00 00 00 73 CE 00 00 00 33 99 CC 66 33 99 CC 66
>0AB00 00 00 00 00 00 00 00 00 F0 F0 F0 F0 F0 F0 F0
>0AB10 00 00 00 00 FF FF FF FF FF 00 00 00 00 00 00
>0AB20 00 00 00 00 00 00 00 FF C0 C0 C0 C0 C0 C0 C0
>0AB30 CC CC 33 33 CC CC 33 33 03 03 03 03 03 03 03
>0AB40 00 00 00 CC CC CC 33 33 CC 99 33 66 CC 99 33 66
>0AB50 03 03 03 03 03 03 03 18 18 18 1F 18 18 18 18
>0AB60 00 00 00 00 0F 0F 0F 0F 18 18 18 1F 00 00 00 00
>0AB70 00 00 00 F8 18 18 18 18 00 00 00 00 00 00 FF FF
>0AB80 00 00 00 1F 18 18 18 18 18 18 18 18 18 18 18 18
>0AB90 00 00 00 FF 18 18 18 18 18 18 18 F8 18 18 18 18
>0ABA0 C0 C0 C0 C0 C0 C0 C0 E0 E0 E0 E0 E0 E0 E0 E0
>0ABB0 07 07 07 07 07 07 07 FF FF 00 00 00 00 00 00
>0ABC0 FF FF FF 00 00 00 00 00 00 00 00 00 00 00 FF FF
>0ABD0 01 03 06 6C 78 70 60 00 00 00 00 00 00 F0 F0 F0
>0ABE0 0F 0F 0F 0F 00 00 00 00 18 18 18 F8 00 00 00 00
>0ABF0 F0 F0 F0 F0 00 00 00 00 F0 F0 F0 0F 0F 0F 0F
>0AC00 00 78 CF CE C0 CE CF 78 FF FF FF E7 C3 99 81 99
>0AC10 FF FF FF 83 99 83 99 83 FF FF FF C3 99 9F 99 C3
>0AC20 FF FF FF 83 99 99 99 83 FF FF FF 81 9F 83 9F 81
>0AC30 FF FF FF 81 9F 87 9F 9F FF FF FF 83 3F 31 39 83
>0AC40 FF FF FF 99 99 81 99 99 FF FF FF C3 E7 E7 E7 C3
>0AC50 FF FF FF E1 F3 F3 93 C7 FF FF FF 99 93 87 93 99
>0AC60 FF FF FF 9F 9F 9F 9F 83 FF FF FF 39 11 01 29 39
>0AC70 FF FF FF 39 19 09 21 31 FF FF FF C3 99 99 99 C3
>0AC80 FF FF FF 83 99 83 9F 9F FF FF FF C3 99 99 93 C1
>0AC90 FF FF FF 83 99 83 99 99 FF FF FF 81 9F 81 F9 81
>0ACA0 FF FF FF 81 E7 E7 E7 E7 FF FF FF 99 99 99 99 C3
>0ACB0 FF FF FF 99 99 99 C3 E7 FF FF FF 39 29 01 83 93
>0ACCO FF FF FF 99 C3 E7 C3 99 FF FF FF 99 99 C3 E7 E7
>0ACD0 FF FF FF 81 F3 E7 CF 81 FF C3 CF CF CF CF CF C3
>0ACE0 FF FF 3F 9F CF E7 F3 F9 FF C3 F3 F3 F3 F3 F3 C3
>0ACF0 FF E7 C3 9F FF FF FF FF FF FF FF FF FF FF 00
>0AD00 FF FF FF FF FF FF FF FF FF E7 C3 C3 E7 FF FF E7
>0AD10 FF 99 99 99 FF FF FF FF FF 99 99 00 99 00 99 99
>0AD20 FF E7 C1 9F C3 F9 83 E7 FF 9D 99 F3 E7 CF 99 B9
>0AD30 FF C3 99 C3 C7 98 99 C0 FF F9 F3 E7 FF FF FF FF
>0AD40 FF F3 E7 CF CF E7 F3 FF CF E7 F3 F3 F3 E7 CF
>0AD50 FF FF 99 C3 00 C3 99 FF FF FF E7 E7 81 E7 E7 FF
>0AD60 FF FF FF FF FF E7 E7 CF FF FF FF FF 81 FF FF FF
>0AD70 FF FF FF FF FF FF E7 E7 FF FF F9 F3 E7 CF 9F 3F
```

```

>0AD80 FF 81 99 99 19 19 19 01 FF E7 C7 E7 E3 E3 E3 81
>0AD90 FF 01 F9 F9 01 1F 1F 01 FF 03 F3 F3 81 F1 F1 01
>0ADA0 FF 33 33 33 01 F1 F1 F1 FF 01 3F 3F 01 F1 31 01
>0ADB0 FF 81 99 9F 01 19 19 01 FF 01 39 F9 F1 F1 F1 F1
>0ADC0 FF 81 99 99 01 39 39 01 FF 03 33 33 01 F1 F1 F1
>0ADD0 FF FF FF E7 FF FF E7 FF FF FF E7 FF FF E7 CF
>0ADE0 FF F1 E7 CF 9F CF E7 F1 FF FF FF 81 FF 81 FF FF
>0ADF0 FF 8F E7 F3 F9 F3 E7 8F FF C3 99 F9 F3 E7 FF E7
>0AE00 FF 9F CF E7 FF FF FF FF FF FF 81 99 99 01 19 19 19
>0AE10 FF 03 33 33 01 39 39 01 FF 81 99 9F 1F 1F 19 01
>0AE20 FF 03 33 33 31 31 31 01 FF 81 9F 9F 07 1F 1F 01
>0AE30 FF 81 9F 9F 07 1F 1F 1F FF 81 9F 9F 11 19 19 01
>0AE40 FF 99 99 99 01 19 19 19 FF E7 E7 E7 C7 C7 C7 C7
>0AE50 FF C1 F3 F3 F1 31 31 01 FF 99 93 87 0F 07 13 19
>0AE60 FF 9F 9F 9F 1F 1F 1F 01 FF 01 25 25 25 25 25 25
>0AE70 FF 83 99 99 19 19 19 19 FF 81 99 99 19 19 19 01
>0AE80 FF 03 33 33 03 1F 1F 1F FF 81 99 99 19 19 01 F8
>0AE90 FF 03 33 33 03 31 31 31 FF 01 3F 3F 01 F1 F1 01
>0AEA0 FF 81 E7 E7 C7 C7 C7 C7 FF 99 99 99 19 19 19 01
>0AEB0 FF 99 99 99 99 99 C3 E7 FF 39 39 39 39 29 01 11
>0AEC0 FF 33 33 87 CF 87 33 33 FF 99 99 C3 E7 E7 E7 E7
>0AED0 FF 03 F3 E7 CF 9F 3F 03 FF F3 E7 E7 CF E7 E7 F3
>0AEE0 E7 E7 E7 E7 E7 E7 E7 FF CF E7 E7 F3 E7 E7 CF
>0AEF0 FF FF FF 8C 31 FF FF FF CC 66 33 99 CC 66 33 99
>0AF00 FF FF FF FF FF FF FF FF 0F 0F 0F 0F 0F 0F 0F
>0AF10 FF FF FF FF 00 00 00 00 FF FF FF FF FF FF FF
>0AF20 FF FF FF FF FF FF FF 00 3F 3F 3F 3F 3F 3F 3F
>0AF30 33 33 CC CC 33 33 CC CC FC FC FC FC FC FC FC
>0AF40 FF FF FF FF 33 33 CC CC 33 66 CC 99 33 66 CC 99
>0AF50 FC FC FC FC FC FC FC FC E7 E7 E7 E0 E7 E7 E7 E7
>0AF60 FF FF FF FF F0 F0 F0 F0 E7 E7 E7 E0 FF FF FF FF
>0AF70 FF FF FF 07 E7 E7 E7 E7 FF FF FF FF FF FF 00 00
>0AF80 FF FF FF E0 E7 E7 E7 E7 E7 E7 00 FF FF FF FF
>0AF90 FF FF FF 00 E7 E7 E7 E7 E7 E7 07 E7 E7 E7 E7
>0AFA0 3F 3F 3F 3F 3F 3F 3F 1F 1F 1F 1F 1F 1F 1F 1F
>0AFB0 F8 F8 F8 F8 F8 F8 F8 00 00 FF FF FF FF FF FF
>0AFC0 00 00 00 FF FF FF FF FF FF FF FF FF 00 00 00
>0AFD0 FE FC F9 93 87 8F 9F FF FF FF FF 0F 0F 0F 0F
>0AFE0 F0 F0 F0 FF FF FF FF FF E7 E7 E7 07 FF FF FF
>0AFF0 0F 0F 0F 0F FF FF FF FF 0F 0F 0F 0F F0 F0 F0

```

Appendix B

C-128 I/O Pinouts

This section contains the pinouts of all the I/O connectors on the C-128. Using this information, you can design your own interfaces for devices that do not hook up directly to the C-128.

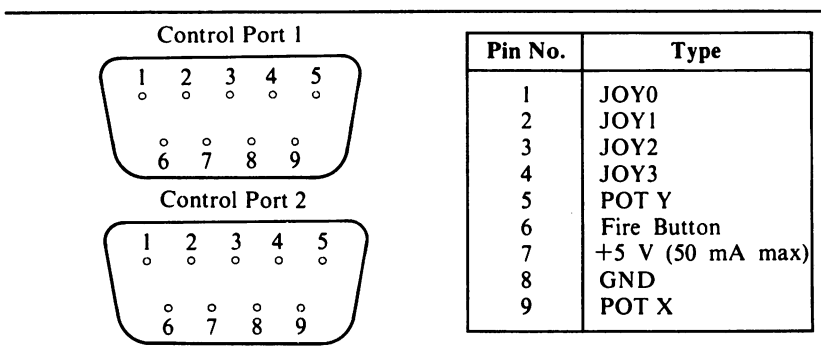
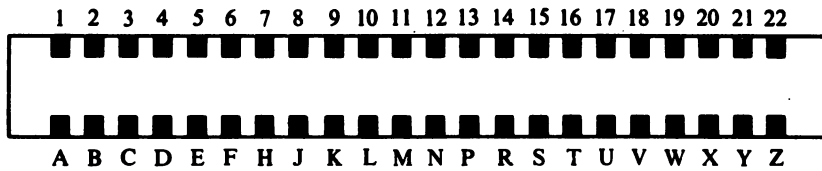


Figure B-1. Game port pinout



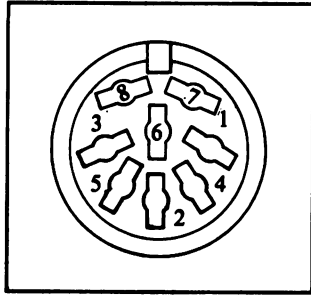
| Pin No. | Type |
|---------|-----------|
| 1 | GND |
| 2 | +5 V |
| 3 | +5 V |
| 4 | IRQ |
| 5 | R/W |
| 6 | DOT CLOCK |
| 7 | I/O 1 |
| 8 | GAME |
| 9 | EX ROM |
| 10 | I/O 2 |
| 11 | ROM L |

| Pin No. | Type |
|---------|------|
| 12 | BA |
| 13 | DMA |
| 14 | D7 |
| 15 | D6 |
| 16 | D5 |
| 17 | D4 |
| 18 | D3 |
| 19 | D2 |
| 20 | D1 |
| 21 | D0 |
| 22 | GND |

| Pin No. | Type |
|---------|-------|
| A | GND |
| B | ROM H |
| C | RESET |
| D | NMI |
| E | S 02 |
| F | A 15 |
| H | A 14 |
| J | A 13 |
| K | A 12 |
| L | A 11 |
| M | A 10 |

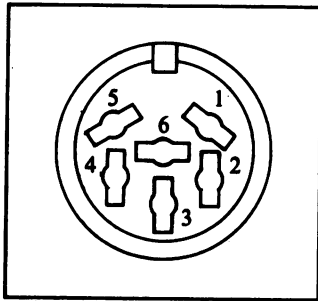
| Pin No. | Type |
|---------|------|
| N | A9 |
| P | A8 |
| R | A7 |
| S | A6 |
| T | A5 |
| U | A4 |
| V | A3 |
| W | A2 |
| X | A1 |
| Y | A0 |
| Z | GND |

Figure B-2. Expansion port pinout



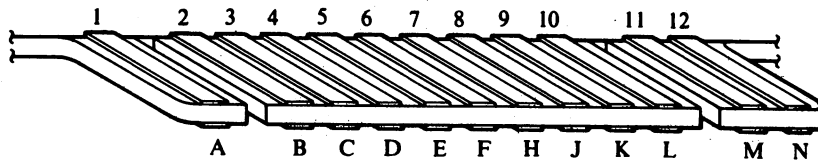
| Pin No. | Type |
|---------|-----------|
| 1 | LUMINANCE |
| 2 | GND |
| 3 | AUDIO OUT |
| 4 | VIDEO OUT |
| 5 | AUDIO IN |
| 6 | COLOR OUT |
| 7 | N/C |
| 8 | N/C |

Figure B-3. Audio/video port pinout



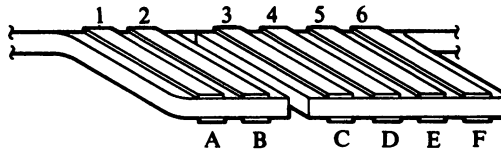
| Pin No. | Type |
|---------|--------------------|
| 1 | SERIAL SRQ IN |
| 2 | GND |
| 3 | SERIAL ATN IN/OUT |
| 4 | SERIAL CLK IN/OUT |
| 5 | SERIAL DATA IN/OUT |
| 6 | RESET |

Figure B-4. Serial I/O port pinout



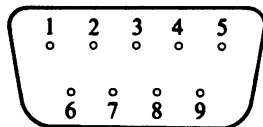
| Pin No. | Type |
|---------|-------------------|
| 1 | GND |
| 2 | +5 V |
| 3 | RESET |
| 4 | CONTROL 1 |
| 5 | SP 1 |
| 6 | CONTROL 2 |
| 7 | SP 2 |
| 8 | PC 2 |
| 9 | SERIAL ATN IN |
| 10 | +9 V (100 mA max) |
| 11 | +9 V (100 mA max) |
| 12 | GND |
| A | GND |
| B | FLAG 2 |
| C | PB0 |
| D | PB1 |
| E | PB2 |
| F | PB3 |
| H | PB4 |
| J | PB5 |
| K | PB6 |
| L | PB7 |
| M | PA2 |
| N | GND |

Figure B-5. User port pinout



| Pin No. | Type |
|---------|----------------|
| A and 1 | GND |
| B and 2 | +5 V |
| C and 3 | CASSETTE MOTOR |
| D and 4 | CASSETTE READ |
| E and 5 | CASSETTE WRITE |
| F and 6 | CASSETTE |

Figure B-6. Cassette interface pinout



| Pin No. | Type |
|---------|-------------|
| 1 | GND |
| 2 | GND |
| 3 | RED |
| 4 | GREEN |
| 5 | BLUE |
| 6 | INTENSITY |
| 7 | MONOCHROME |
| 8 | HORIZ. SYNC |
| 9 | VERT. SYNC |

Figure B-7. RGB 1 Connection

Appendix C

Conversion Tables: Trigonometric Functions

The tables in this section are intended as an aid to mathematical programming.

Hexadecimal-Decimal Integer Conversion

Table C-1 provides for direct conversions between hexadecimal integers in the range 0-FFF and decimal integers in the range 0-4095. For conversion of larger integers, the table values may be added to the figures in Table C-2.

Table C-1. Hexadecimal-Decimal Integer Conversion

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|----|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| 00 | 0000 | 0001 | 0002 | 0003 | 0004 | 0005 | 0006 | 0007 | 0008 | 0009 | 0010 | 0011 | 0012 | 0013 | 0014 | 0015 |
| 01 | 0016 | 0017 | 0018 | 0019 | 0020 | 0021 | 0022 | 0023 | 0024 | 0025 | 0026 | 0027 | 0028 | 0029 | 0030 | 0031 |
| 02 | 0032 | 0033 | 0034 | 0035 | 0036 | 0037 | 0038 | 0039 | 0040 | 0041 | 0042 | 0043 | 0044 | 0045 | 0046 | 0047 |
| 03 | 0048 | 0049 | 0050 | 0051 | 0052 | 0053 | 0054 | 0055 | 0056 | 0057 | 0058 | 0059 | 0060 | 0061 | 0062 | 0063 |
| 04 | 0064 | 0065 | 0066 | 0067 | 0068 | 0069 | 0070 | 0071 | 0072 | 0073 | 0074 | 0075 | 0076 | 0077 | 0078 | 0079 |
| 05 | 0080 | 0081 | 0082 | 0083 | 0084 | 0085 | 0086 | 0087 | 0088 | 0089 | 0090 | 0091 | 0092 | 0093 | 0094 | 0095 |
| 06 | 0096 | 0097 | 0098 | 0099 | 0100 | 0101 | 0102 | 0103 | 0104 | 0105 | 0106 | 0107 | 0108 | 0109 | 0110 | 0111 |
| 07 | 0112 | 0113 | 0114 | 0115 | 0116 | 0117 | 0118 | 0119 | 0120 | 0121 | 0122 | 0123 | 0124 | 0125 | 0126 | 0127 |
| 08 | 0128 | 0129 | 0130 | 0131 | 0132 | 0133 | 0134 | 0135 | 0136 | 0137 | 0138 | 0139 | 0140 | 0141 | 0142 | 0143 |
| 09 | 0144 | 0145 | 0146 | 0147 | 0148 | 0149 | 0150 | 0151 | 0152 | 0153 | 0154 | 0155 | 0156 | 0157 | 0158 | 0159 |
| 0A | 0160 | 0161 | 0162 | 0163 | 0164 | 0165 | 0166 | 0167 | 0168 | 0169 | 0170 | 0171 | 0172 | 0173 | 0174 | 0175 |
| 0B | 0176 | 0177 | 0178 | 0179 | 0180 | 0181 | 0182 | 0183 | 0184 | 0185 | 0186 | 0187 | 0188 | 0189 | 0190 | 0191 |
| 0C | 0192 | 0193 | 0194 | 0195 | 0196 | 0197 | 0198 | 0199 | 0200 | 0201 | 0202 | 0203 | 0204 | 0205 | 0206 | 0207 |
| 0D | 0208 | 0209 | 0210 | 0211 | 0212 | 0213 | 0214 | 0215 | 0216 | 0217 | 0218 | 0219 | 0220 | 0221 | 0222 | 0223 |
| 0E | 0224 | 0225 | 0226 | 0227 | 0228 | 0229 | 0230 | 0231 | 0232 | 0233 | 0234 | 0235 | 0236 | 0237 | 0238 | 0239 |
| 0F | 0240 | 0241 | 0242 | 0243 | 0244 | 0245 | 0246 | 0247 | 0248 | 0249 | 0250 | 0251 | 0252 | 0253 | 0254 | 0255 |
| 10 | 0256 | 0257 | 0258 | 0259 | 0260 | 0261 | 0262 | 0263 | 0264 | 0265 | 0266 | 0267 | 0268 | 0269 | 0270 | 0271 |
| 11 | 0272 | 0273 | 0274 | 0275 | 0276 | 0277 | 0278 | 0279 | 0280 | 0281 | 0282 | 0283 | 0284 | 0285 | 0286 | 0287 |
| 12 | 0288 | 0289 | 0290 | 0291 | 0292 | 0293 | 0294 | 0295 | 0296 | 0297 | 0298 | 0299 | 0300 | 0301 | 0302 | 0303 |
| 13 | 0304 | 0305 | 0306 | 0307 | 0308 | 0309 | 0310 | 0311 | 0312 | 0313 | 0314 | 0315 | 0316 | 0317 | 0318 | 0319 |
| 14 | 0320 | 0321 | 0322 | 0323 | 0324 | 0325 | 0326 | 0327 | 0328 | 0329 | 0330 | 0331 | 0332 | 0333 | 0334 | 0335 |
| 15 | 0336 | 0337 | 0338 | 0339 | 0340 | 0341 | 0342 | 0343 | 0344 | 0345 | 0346 | 0347 | 0348 | 0349 | 0350 | 0351 |
| 16 | 0352 | 0353 | 0354 | 0355 | 0356 | 0357 | 0358 | 0359 | 0360 | 0361 | 0362 | 0363 | 0364 | 0365 | 0366 | 0367 |
| 17 | 0368 | 0369 | 0370 | 0371 | 0372 | 0373 | 0374 | 0375 | 0376 | 0377 | 0378 | 0379 | 0380 | 0381 | 0382 | 0383 |
| 18 | 0384 | 0385 | 0386 | 0387 | 0388 | 0389 | 0390 | 0391 | 0392 | 0393 | 0394 | 0395 | 0396 | 0397 | 0398 | 0399 |
| 19 | 0400 | 0401 | 0402 | 0403 | 0404 | 0405 | 0406 | 0407 | 0408 | 0409 | 0410 | 0411 | 0412 | 0413 | 0414 | 0415 |
| 1A | 0416 | 0417 | 0418 | 0419 | 0420 | 0421 | 0422 | 0423 | 0424 | 0425 | 0426 | 0427 | 0428 | 0429 | 0430 | 0431 |
| 1B | 0432 | 0433 | 0434 | 0435 | 0436 | 0437 | 0438 | 0439 | 0440 | 0441 | 0442 | 0443 | 0444 | 0445 | 0446 | 0447 |
| 1C | 0448 | 0449 | 0450 | 0451 | 0452 | 0453 | 0454 | 0455 | 0456 | 0457 | 0458 | 0459 | 0460 | 0461 | 0462 | 0463 |
| 1D | 0464 | 0465 | 0466 | 0467 | 0468 | 0469 | 0470 | 0471 | 0472 | 0473 | 0474 | 0475 | 0476 | 0477 | 0478 | 0479 |
| 1E | 0480 | 0481 | 0482 | 0483 | 0484 | 0485 | 0486 | 0487 | 0488 | 0489 | 0490 | 0491 | 0492 | 0493 | 0494 | 0495 |
| 1F | 0496 | 0497 | 0498 | 0499 | 0500 | 0501 | 0502 | 0503 | 0504 | 0505 | 0506 | 0507 | 0508 | 0509 | 0510 | 0511 |
| 20 | 0512 | 0513 | 0514 | 0515 | 0516 | 0517 | 0518 | 0519 | 0520 | 0521 | 0522 | 0523 | 0524 | 0525 | 0526 | 0527 |
| 21 | 0528 | 0529 | 0530 | 0531 | 0532 | 0533 | 0534 | 0535 | 0536 | 0537 | 0538 | 0539 | 0540 | 0541 | 0542 | 0543 |
| 22 | 0544 | 0545 | 0546 | 0547 | 0548 | 0549 | 0550 | 0551 | 0552 | 0553 | 0554 | 0555 | 0556 | 0557 | 0558 | 0559 |
| 23 | 0560 | 0561 | 0562 | 0563 | 0564 | 0565 | 0566 | 0567 | 0568 | 0569 | 0570 | 0571 | 0572 | 0573 | 0574 | 0575 |
| 24 | 0576 | 0577 | 0578 | 0579 | 0580 | 0581 | 0582 | 0583 | 0584 | 0585 | 0586 | 0587 | 0588 | 0589 | 0590 | 0591 |
| 25 | 0592 | 0593 | 0594 | 0595 | 0596 | 0597 | 0598 | 0599 | 0600 | 0601 | 0602 | 0603 | 0604 | 0605 | 0606 | 0607 |
| 26 | 0608 | 0609 | 0610 | 0611 | 0612 | 0613 | 0614 | 0615 | 0616 | 0617 | 0618 | 0619 | 0620 | 0621 | 0622 | 0623 |
| 27 | 0624 | 0625 | 0626 | 0627 | 0628 | 0629 | 0630 | 0631 | 0632 | 0633 | 0634 | 0635 | 0636 | 0637 | 0638 | 0639 |
| 28 | 0640 | 0641 | 0642 | 0643 | 0644 | 0645 | 0646 | 0647 | 0648 | 0649 | 0650 | 0651 | 0652 | 0653 | 0654 | 0655 |
| 29 | 0656 | 0657 | 0658 | 0659 | 0660 | 0661 | 0662 | 0663 | 0664 | 0665 | 0666 | 0667 | 0668 | 0669 | 0670 | 0671 |
| 2A | 0672 | 0673 | 0674 | 0675 | 0676 | 0677 | 0678 | 0679 | 0680 | 0681 | 0682 | 0683 | 0684 | 0685 | 0686 | 0687 |
| 2B | 0688 | 0689 | 0690 | 0691 | 0692 | 0693 | 0694 | 0695 | 0696 | 0697 | 0698 | 0699 | 0700 | 0701 | 0702 | 0703 |
| 2C | 0704 | 0705 | 0706 | 0707 | 0708 | 0709 | 0710 | 0711 | 0712 | 0713 | 0714 | 0715 | 0716 | 0717 | 0718 | 0719 |
| 2D | 0720 | 0721 | 0722 | 0723 | 0724 | 0725 | 0726 | 0727 | 0728 | 0729 | 0730 | 0731 | 0732 | 0733 | 0734 | 0735 |
| 2E | 0736 | 0737 | 0738 | 0739 | 0740 | 0741 | 0742 | 0743 | 0744 | 0745 | 0746 | 0747 | 0748 | 0749 | 0750 | 0751 |
| 2F | 0752 | 0753 | 0754 | 0755 | 0756 | 0757 | 0758 | 0759 | 0760 | 0761 | 0762 | 0763 | 0764 | 0765 | 0766 | 0767 |

Table C-1. Hexadecimal-Decimal Integer Conversion (*continued*)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|----|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| 30 | 0768 | 0769 | 0770 | 0771 | 0772 | 0773 | 0774 | 0775 | 0776 | 0777 | 0778 | 0779 | 0780 | 0781 | 0782 | 0783 |
| 31 | 0784 | 0785 | 0786 | 0787 | 0788 | 0789 | 0790 | 0791 | 0792 | 0793 | 0794 | 0795 | 0796 | 0797 | 0798 | 0799 |
| 32 | 0800 | 0801 | 0802 | 0803 | 0804 | 0805 | 0806 | 0807 | 0808 | 0809 | 0810 | 0811 | 0812 | 0813 | 0814 | 0815 |
| 33 | 0816 | 0817 | 0818 | 0819 | 0820 | 0821 | 0822 | 0823 | 0824 | 0825 | 0826 | 0827 | 0828 | 0829 | 0830 | 0831 |
| 34 | 0832 | 0833 | 0834 | 0835 | 0836 | 0837 | 0838 | 0839 | 0840 | 0841 | 0842 | 0843 | 0844 | 0845 | 0846 | 0847 |
| 35 | 0848 | 0849 | 0850 | 0851 | 0852 | 0853 | 0854 | 0855 | 0856 | 0857 | 0858 | 0859 | 0860 | 0861 | 0862 | 0863 |
| 36 | 0864 | 0865 | 0866 | 0867 | 0868 | 0869 | 0870 | 0871 | 0872 | 0873 | 0874 | 0875 | 0876 | 0877 | 0878 | 0879 |
| 37 | 0880 | 0881 | 0882 | 0883 | 0884 | 0885 | 0886 | 0887 | 0888 | 0889 | 0890 | 0891 | 0892 | 0893 | 0894 | 0895 |
| 38 | 0896 | 0897 | 0898 | 0899 | 0900 | 0901 | 0902 | 0903 | 0904 | 0905 | 0906 | 0907 | 0908 | 0909 | 0910 | 0911 |
| 39 | 0912 | 0913 | 0914 | 0915 | 0916 | 0917 | 0918 | 0919 | 0920 | 0921 | 0922 | 0923 | 0924 | 0925 | 0926 | 0927 |
| 3A | 0928 | 0929 | 0930 | 0931 | 0932 | 0933 | 0934 | 0935 | 0936 | 0937 | 0938 | 0939 | 0940 | 0941 | 0942 | 0943 |
| 3B | 0944 | 0945 | 0946 | 0947 | 0948 | 0949 | 0950 | 0951 | 0952 | 0953 | 0954 | 0955 | 0956 | 0957 | 0958 | 0959 |
| 3C | 0960 | 0961 | 0962 | 0963 | 0964 | 0965 | 0966 | 0967 | 0968 | 0969 | 0970 | 0971 | 0972 | 0973 | 0974 | 0975 |
| 3D | 0976 | 0977 | 0978 | 0979 | 0980 | 0981 | 0982 | 0983 | 0984 | 0985 | 0986 | 0987 | 0988 | 0989 | 0990 | 0991 |
| 3E | 0992 | 0993 | 0994 | 0995 | 0996 | 0997 | 0998 | 0999 | 1000 | 1001 | 1002 | 1003 | 1004 | 1005 | 1006 | 1007 |
| 3F | 1008 | 1009 | 1010 | 1011 | 1012 | 1013 | 1014 | 1015 | 1016 | 1017 | 1018 | 1019 | 1020 | 1021 | 1022 | 1023 |
| 40 | 1024 | 1025 | 1026 | 1027 | 1028 | 1029 | 1030 | 1031 | 1032 | 1033 | 1034 | 1035 | 1036 | 1037 | 1038 | 1039 |
| 41 | 1040 | 1041 | 1042 | 1043 | 1044 | 1045 | 1046 | 1047 | 1048 | 1049 | 1050 | 1051 | 1052 | 1053 | 1054 | 1055 |
| 42 | 1056 | 1057 | 1058 | 1059 | 1060 | 1061 | 1062 | 1063 | 1064 | 1065 | 1066 | 1067 | 1068 | 1069 | 1070 | 1071 |
| 43 | 1072 | 1073 | 1074 | 1075 | 1076 | 1077 | 1078 | 1079 | 1080 | 1081 | 1082 | 1083 | 1084 | 1085 | 1086 | 1087 |
| 44 | 1088 | 1089 | 1090 | 1091 | 1092 | 1093 | 1094 | 1095 | 1096 | 1097 | 1098 | 1099 | 1100 | 1101 | 1102 | 1103 |
| 45 | 1104 | 1105 | 1106 | 1107 | 1108 | 1109 | 1110 | 1111 | 1112 | 1113 | 1114 | 1115 | 1116 | 1117 | 1118 | 1119 |
| 46 | 1120 | 1121 | 1122 | 1123 | 1124 | 1125 | 1126 | 1127 | 1128 | 1129 | 1130 | 1131 | 1132 | 1133 | 1134 | 1135 |
| 47 | 1136 | 1137 | 1138 | 1139 | 1140 | 1141 | 1142 | 1143 | 1144 | 1145 | 1146 | 1147 | 1148 | 1149 | 1150 | 1151 |
| 48 | 1152 | 1153 | 1154 | 1155 | 1156 | 1157 | 1158 | 1159 | 1160 | 1161 | 1162 | 1163 | 1164 | 1165 | 1166 | 1167 |
| 49 | 1168 | 1169 | 1170 | 1171 | 1172 | 1173 | 1174 | 1175 | 1176 | 1177 | 1178 | 1179 | 1180 | 1181 | 1182 | 1183 |
| 4A | 1184 | 1185 | 1186 | 1187 | 1188 | 1189 | 1190 | 1191 | 1192 | 1193 | 1194 | 1195 | 1196 | 1197 | 1198 | 1199 |
| 4B | 1200 | 1201 | 1202 | 1203 | 1204 | 1205 | 1206 | 1207 | 1208 | 1209 | 1210 | 1211 | 1212 | 1213 | 1214 | 1215 |
| 4C | 1216 | 1217 | 1218 | 1219 | 1220 | 1221 | 1222 | 1223 | 1224 | 1225 | 1226 | 1227 | 1228 | 1229 | 1230 | 1231 |
| 4D | 1232 | 1233 | 1234 | 1235 | 1236 | 1237 | 1238 | 1239 | 1240 | 1241 | 1242 | 1243 | 1244 | 1245 | 1246 | 1247 |
| 4E | 1248 | 1249 | 1250 | 1251 | 1252 | 1253 | 1254 | 1255 | 1256 | 1257 | 1258 | 1259 | 1260 | 1261 | 1262 | 1263 |
| 4F | 1264 | 1265 | 1266 | 1267 | 1268 | 1269 | 1270 | 1271 | 1272 | 1273 | 1274 | 1275 | 1276 | 1277 | 1278 | 1279 |
| 50 | 1280 | 1281 | 1282 | 1283 | 1284 | 1285 | 1286 | 1287 | 1288 | 1289 | 1290 | 1291 | 1292 | 1293 | 1294 | 1295 |
| 51 | 1296 | 1297 | 1298 | 1299 | 1300 | 1301 | 1302 | 1303 | 1304 | 1305 | 1306 | 1307 | 1308 | 1309 | 1310 | 1311 |
| 52 | 1312 | 1313 | 1314 | 1315 | 1316 | 1317 | 1318 | 1319 | 1320 | 1321 | 1322 | 1323 | 1324 | 1325 | 1326 | 1327 |
| 53 | 1328 | 1329 | 1330 | 1331 | 1332 | 1333 | 1334 | 1335 | 1336 | 1337 | 1338 | 1339 | 1340 | 1341 | 1342 | 1343 |
| 54 | 1344 | 1345 | 1346 | 1347 | 1348 | 1349 | 1350 | 1351 | 1352 | 1353 | 1354 | 1355 | 1356 | 1357 | 1358 | 1359 |
| 55 | 1360 | 1361 | 1362 | 1363 | 1364 | 1365 | 1366 | 1367 | 1368 | 1369 | 1370 | 1371 | 1372 | 1373 | 1374 | 1375 |
| 56 | 1376 | 1377 | 1378 | 1379 | 1380 | 1381 | 1382 | 1383 | 1384 | 1385 | 1386 | 1387 | 1388 | 1389 | 1390 | 1391 |
| 57 | 1392 | 1393 | 1394 | 1395 | 1396 | 1397 | 1398 | 1399 | 1400 | 1401 | 1402 | 1403 | 1404 | 1405 | 1406 | 1407 |
| 58 | 1408 | 1409 | 1410 | 1411 | 1412 | 1413 | 1414 | 1415 | 1416 | 1417 | 1418 | 1419 | 1420 | 1421 | 1422 | 1423 |
| 59 | 1424 | 1425 | 1426 | 1427 | 1428 | 1429 | 1430 | 1431 | 1432 | 1433 | 1434 | 1435 | 1436 | 1437 | 1438 | 1439 |
| 5A | 1440 | 1441 | 1442 | 1443 | 1444 | 1445 | 1446 | 1447 | 1448 | 1449 | 1450 | 1451 | 1452 | 1453 | 1454 | 1455 |
| 5B | 1456 | 1457 | 1458 | 1459 | 1460 | 1461 | 1462 | 1463 | 1464 | 1465 | 1466 | 1467 | 1468 | 1469 | 1470 | 1471 |
| 5C | 1472 | 1473 | 1474 | 1475 | 1476 | 1477 | 1478 | 1479 | 1480 | 1481 | 1482 | 1483 | 1484 | 1485 | 1486 | 1487 |
| 5D | 1488 | 1489 | 1490 | 1491 | 1492 | 1493 | 1494 | 1495 | 1496 | 1497 | 1498 | 1499 | 1500 | 1501 | 1502 | 1503 |
| 5E | 1504 | 1505 | 1506 | 1507 | 1508 | 1509 | 1510 | 1511 | 1512 | 1513 | 1514 | 1515 | 1516 | 1517 | 1518 | 1519 |
| 5F | 1520 | 1521 | 1522 | 1523 | 1524 | 1525 | 1526 | 1527 | 1528 | 1529 | 1530 | 1531 | 1532 | 1533 | 1534 | 1535 |

Table C-1. Hexadecimal-Decimal Integer Conversion (*continued*)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|----|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| 60 | 1536 | 1537 | 1538 | 1539 | 1540 | 1541 | 1542 | 1543 | 1544 | 1545 | 1546 | 1547 | 1548 | 1549 | 1550 | 1551 |
| 61 | 1552 | 1553 | 1554 | 1555 | 1556 | 1557 | 1558 | 1559 | 1560 | 1561 | 1562 | 1563 | 1564 | 1565 | 1566 | 1567 |
| 62 | 1568 | 1569 | 1570 | 1571 | 1572 | 1573 | 1574 | 1575 | 1576 | 1577 | 1578 | 1579 | 1580 | 1581 | 1582 | 1583 |
| 63 | 1584 | 1585 | 1586 | 1587 | 1588 | 1589 | 1590 | 1591 | 1592 | 1593 | 1594 | 1595 | 1596 | 1597 | 1598 | 1599 |
| 64 | 1600 | 1601 | 1602 | 1603 | 1604 | 1605 | 1606 | 1607 | 1608 | 1609 | 1610 | 1611 | 1612 | 1613 | 1614 | 1615 |
| 65 | 1616 | 1617 | 1618 | 1619 | 1620 | 1621 | 1622 | 1623 | 1624 | 1625 | 1626 | 1627 | 1628 | 1629 | 1630 | 1631 |
| 66 | 1632 | 1633 | 1634 | 1635 | 1636 | 1637 | 1638 | 1639 | 1640 | 1641 | 1642 | 1643 | 1644 | 1645 | 1646 | 1647 |
| 67 | 1648 | 1649 | 1650 | 1651 | 1652 | 1653 | 1654 | 1655 | 1656 | 1657 | 1658 | 1659 | 1660 | 1661 | 1662 | 1663 |
| 68 | 1664 | 1665 | 1666 | 1667 | 1668 | 1669 | 1670 | 1671 | 1672 | 1673 | 1674 | 1675 | 1676 | 1677 | 1678 | 1679 |
| 69 | 1680 | 1681 | 1682 | 1683 | 1684 | 1685 | 1686 | 1687 | 1688 | 1689 | 1690 | 1691 | 1692 | 1693 | 1694 | 1695 |
| 6A | 1696 | 1697 | 1698 | 1699 | 1700 | 1701 | 1702 | 1703 | 1704 | 1705 | 1706 | 1707 | 1708 | 1709 | 1710 | 1711 |
| 6B | 1712 | 1713 | 1714 | 1715 | 1716 | 1717 | 1718 | 1719 | 1720 | 1721 | 1722 | 1723 | 1724 | 1725 | 1726 | 1727 |
| 6C | 1728 | 1729 | 1730 | 1731 | 1732 | 1733 | 1734 | 1735 | 1736 | 1737 | 1738 | 1739 | 1740 | 1741 | 1742 | 1743 |
| 6D | 1744 | 1745 | 1746 | 1747 | 1748 | 1749 | 1750 | 1751 | 1752 | 1753 | 1754 | 1755 | 1756 | 1757 | 1758 | 1759 |
| 6E | 1760 | 1761 | 1762 | 1763 | 1764 | 1765 | 1766 | 1767 | 1768 | 1769 | 1770 | 1771 | 1772 | 1773 | 1774 | 1775 |
| 6F | 1776 | 1777 | 1778 | 1779 | 1780 | 1781 | 1782 | 1783 | 1784 | 1785 | 1786 | 1787 | 1788 | 1789 | 1790 | 1791 |
| 70 | 1792 | 1793 | 1794 | 1795 | 1796 | 1797 | 1798 | 1799 | 1800 | 1801 | 1802 | 1803 | 1804 | 1805 | 1806 | 1807 |
| 71 | 1808 | 1809 | 1810 | 1811 | 1812 | 1813 | 1814 | 1815 | 1816 | 1817 | 1818 | 1819 | 1820 | 1821 | 1822 | 1823 |
| 72 | 1824 | 1825 | 1826 | 1827 | 1828 | 1829 | 1830 | 1831 | 1832 | 1833 | 1834 | 1835 | 1836 | 1837 | 1838 | 1839 |
| 73 | 1840 | 1841 | 1842 | 1843 | 1844 | 1845 | 1846 | 1847 | 1848 | 1849 | 1850 | 1851 | 1852 | 1853 | 1854 | 1855 |
| 74 | 1856 | 1857 | 1858 | 1859 | 1860 | 1861 | 1862 | 1863 | 1864 | 1865 | 1866 | 1867 | 1868 | 1869 | 1870 | 1871 |
| 75 | 1872 | 1873 | 1874 | 1875 | 1876 | 1877 | 1878 | 1879 | 1880 | 1881 | 1882 | 1883 | 1884 | 1885 | 1886 | 1887 |
| 76 | 1888 | 1889 | 1890 | 1891 | 1892 | 1893 | 1894 | 1895 | 1896 | 1897 | 1898 | 1899 | 1900 | 1901 | 1902 | 1903 |
| 77 | 1904 | 1905 | 1906 | 1907 | 1908 | 1909 | 1910 | 1911 | 1912 | 1913 | 1914 | 1915 | 1916 | 1917 | 1918 | 1919 |
| 78 | 1920 | 1921 | 1922 | 1923 | 1924 | 1925 | 1926 | 1927 | 1928 | 1929 | 1930 | 1931 | 1932 | 1933 | 1934 | 1935 |
| 79 | 1936 | 1937 | 1938 | 1939 | 1940 | 1941 | 1942 | 1943 | 1944 | 1945 | 1946 | 1947 | 1948 | 1949 | 1950 | 1951 |
| 7A | 1952 | 1953 | 1954 | 1955 | 1956 | 1957 | 1958 | 1959 | 1960 | 1961 | 1962 | 1963 | 1964 | 1965 | 1966 | 1967 |
| 7B | 1968 | 1969 | 1970 | 1971 | 1972 | 1973 | 1974 | 1975 | 1976 | 1977 | 1978 | 1979 | 1980 | 1981 | 1982 | 1983 |
| 7C | 1984 | 1985 | 1986 | 1987 | 1988 | 1989 | 1990 | 1991 | 1992 | 1993 | 1994 | 1995 | 1996 | 1997 | 1998 | 1999 |
| 7D | 2000 | 2001 | 2002 | 2003 | 2004 | 2005 | 2006 | 2007 | 2008 | 2009 | 2010 | 2011 | 2012 | 2013 | 2014 | 2015 |
| 7E | 2016 | 2017 | 2018 | 2019 | 2020 | 2021 | 2022 | 2023 | 2024 | 2025 | 2026 | 2027 | 2028 | 2029 | 2030 | 2031 |
| 7F | 2032 | 2033 | 2034 | 2035 | 2036 | 2037 | 2038 | 2039 | 2040 | 2041 | 2042 | 2043 | 2044 | 2045 | 2046 | 2047 |
| 80 | 2048 | 2049 | 2050 | 2051 | 2052 | 2053 | 2054 | 2055 | 2056 | 2057 | 2058 | 2059 | 2060 | 2061 | 2062 | 2063 |
| 81 | 2064 | 2065 | 2066 | 2067 | 2068 | 2069 | 2070 | 2071 | 2072 | 2073 | 2074 | 2075 | 2076 | 2077 | 2078 | 2079 |
| 82 | 2080 | 2081 | 2082 | 2083 | 2084 | 2085 | 2086 | 2087 | 2088 | 2089 | 2090 | 2091 | 2092 | 2093 | 2094 | 2095 |
| 83 | 2096 | 2097 | 2098 | 2099 | 2100 | 2101 | 2102 | 2103 | 2104 | 2105 | 2106 | 2107 | 2108 | 2109 | 2110 | 2111 |
| 84 | 2112 | 2113 | 2114 | 2115 | 2116 | 2117 | 2118 | 2119 | 2120 | 2121 | 2122 | 2123 | 2124 | 2125 | 2126 | 2127 |
| 85 | 2128 | 2129 | 2130 | 2131 | 2132 | 2133 | 2134 | 2135 | 2136 | 2137 | 2138 | 2139 | 2140 | 2141 | 2142 | 2143 |
| 86 | 2144 | 2145 | 2146 | 2147 | 2148 | 2149 | 2150 | 2151 | 2152 | 2153 | 2154 | 2155 | 2156 | 2157 | 2158 | 2159 |
| 87 | 2160 | 2161 | 2162 | 2163 | 2164 | 2165 | 2166 | 2167 | 2168 | 2169 | 2170 | 2171 | 2172 | 2173 | 2174 | 2175 |
| 88 | 2176 | 2177 | 2178 | 2179 | 2180 | 2181 | 2182 | 2183 | 2184 | 2185 | 2186 | 2187 | 2188 | 2189 | 2190 | 2191 |
| 89 | 2192 | 2193 | 2194 | 2195 | 2196 | 2197 | 2198 | 2199 | 2200 | 2201 | 2202 | 2203 | 2204 | 2205 | 2206 | 2207 |
| 8A | 2208 | 2209 | 2210 | 2211 | 2212 | 2213 | 2214 | 2215 | 2216 | 2217 | 2218 | 2219 | 2220 | 2221 | 2222 | 2223 |
| 8B | 2224 | 2225 | 2226 | 2227 | 2228 | 2229 | 2230 | 2231 | 2232 | 2233 | 2234 | 2235 | 2236 | 2237 | 2238 | 2239 |
| 8C | 2240 | 2241 | 2242 | 2243 | 2244 | 2245 | 2246 | 2247 | 2248 | 2249 | 2250 | 2251 | 2252 | 2253 | 2254 | 2255 |
| 8D | 2256 | 2257 | 2258 | 2259 | 2260 | 2261 | 2262 | 2263 | 2264 | 2265 | 2266 | 2267 | 2268 | 2269 | 2270 | 2271 |
| 8E | 2272 | 2273 | 2274 | 2275 | 2276 | 2277 | 2278 | 2279 | 2280 | 2281 | 2282 | 2283 | 2284 | 2285 | 2286 | 2287 |
| 8F | 2288 | 2289 | 2290 | 2291 | 2292 | 2293 | 2294 | 2295 | 2296 | 2297 | 2298 | 2299 | 2300 | 2301 | 2302 | 2303 |

Table C-1. Hexadecimal-Decimal Integer Conversion (*continued*)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | |
|----|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| 90 | 2304 | 2305 | 2306 | 2307 | 2308 | 2309 | 2310 | 2311 | 2312 | 2313 | 2314 | 2315 | 2316 | 2317 | 2318 | 2319 |
| 91 | 2320 | 2321 | 2322 | 2323 | 2324 | 2325 | 2326 | 2327 | 2328 | 2329 | 2330 | 2331 | 2332 | 2333 | 2334 | 2335 |
| 92 | 2336 | 2337 | 2338 | 2339 | 2340 | 2341 | 2342 | 2343 | 2344 | 2345 | 2346 | 2347 | 2348 | 2349 | 2350 | 2351 |
| 93 | 2352 | 2353 | 2354 | 2355 | 2356 | 2357 | 2358 | 2359 | 2360 | 2361 | 2362 | 2363 | 2364 | 2365 | 2366 | 2367 |
| 94 | 2368 | 2369 | 2370 | 2371 | 2372 | 2373 | 2374 | 2375 | 2376 | 2377 | 2378 | 2379 | 2380 | 2381 | 2382 | 2383 |
| 95 | 2384 | 2385 | 2386 | 2387 | 2388 | 2389 | 2390 | 2391 | 2392 | 2393 | 2394 | 2395 | 2396 | 2397 | 2398 | 2399 |
| 96 | 2400 | 2401 | 2402 | 2403 | 2404 | 2405 | 2406 | 2407 | 2408 | 2409 | 2410 | 2411 | 2412 | 2413 | 2414 | 2415 |
| 97 | 2416 | 2417 | 2418 | 2419 | 2420 | 2421 | 2422 | 2423 | 2424 | 2425 | 2426 | 2427 | 2428 | 2429 | 2430 | 2431 |
| 98 | 2432 | 2433 | 2434 | 2435 | 2436 | 2437 | 2438 | 2439 | 2440 | 2441 | 2442 | 2443 | 2444 | 2445 | 2446 | 2447 |
| 99 | 2448 | 2449 | 2450 | 2451 | 2452 | 2453 | 2454 | 2455 | 2456 | 2457 | 2458 | 2459 | 2460 | 2461 | 2462 | 2463 |
| 9A | 2464 | 2465 | 2466 | 2467 | 2468 | 2469 | 2470 | 2471 | 2472 | 2473 | 2474 | 2475 | 2476 | 2477 | 2478 | 2479 |
| 9B | 2480 | 2481 | 2482 | 2483 | 2484 | 2485 | 2486 | 2487 | 2488 | 2489 | 2490 | 2491 | 2492 | 2493 | 2494 | 2495 |
| 9C | 2496 | 2497 | 2498 | 2499 | 2500 | 2501 | 2502 | 2503 | 2504 | 2505 | 2506 | 2507 | 2508 | 2509 | 2510 | 2511 |
| 9D | 2512 | 2513 | 2514 | 2515 | 2516 | 2517 | 2518 | 2519 | 2520 | 2521 | 2522 | 2523 | 2524 | 2525 | 2526 | 2527 |
| 9E | 2528 | 2529 | 2530 | 2531 | 2532 | 2533 | 2534 | 2535 | 2536 | 2537 | 2538 | 2539 | 2540 | 2541 | 2542 | 2543 |
| 9F | 2544 | 2545 | 2546 | 2547 | 2548 | 2549 | 2550 | 2551 | 2552 | 2553 | 2554 | 2555 | 2556 | 2557 | 2558 | 2559 |
| A0 | 2560 | 2561 | 2562 | 2563 | 2564 | 2565 | 2566 | 2567 | 2568 | 2569 | 2570 | 2571 | 2572 | 2573 | 2574 | 2575 |
| A1 | 2576 | 2577 | 2578 | 2579 | 2580 | 2581 | 2582 | 2583 | 2584 | 2585 | 2586 | 2587 | 2588 | 2589 | 2590 | 2591 |
| A2 | 2592 | 2593 | 2594 | 2595 | 2596 | 2597 | 2598 | 2599 | 2600 | 2601 | 2602 | 2603 | 2604 | 2605 | 2606 | 2607 |
| A3 | 2608 | 2609 | 2610 | 2611 | 2612 | 2613 | 2614 | 2615 | 2616 | 2617 | 2618 | 2619 | 2620 | 2621 | 2622 | 2623 |
| A4 | 2624 | 2625 | 2626 | 2627 | 2628 | 2629 | 2630 | 2631 | 2632 | 2633 | 2634 | 2635 | 2636 | 2637 | 2638 | 2639 |
| A5 | 2640 | 2641 | 2642 | 2643 | 2644 | 2645 | 2646 | 2647 | 2648 | 2649 | 2650 | 2651 | 2652 | 2653 | 2654 | 2655 |
| A6 | 2656 | 2657 | 2658 | 2659 | 2660 | 2661 | 2662 | 2663 | 2664 | 2665 | 2666 | 2667 | 2668 | 2669 | 2670 | 2671 |
| A7 | 2672 | 2673 | 2674 | 2675 | 2676 | 2677 | 2678 | 2679 | 2680 | 2681 | 2682 | 2683 | 2684 | 2685 | 2686 | 2687 |
| A8 | 2688 | 2689 | 2690 | 2691 | 2692 | 2693 | 2694 | 2695 | 2696 | 2697 | 2698 | 2699 | 2700 | 2701 | 2702 | 2703 |
| A9 | 2704 | 2705 | 2706 | 2707 | 2708 | 2709 | 2710 | 2711 | 2712 | 2713 | 2714 | 2715 | 2716 | 2717 | 2718 | 2719 |
| AA | 2720 | 2721 | 2722 | 2723 | 2724 | 2725 | 2726 | 2727 | 2728 | 2729 | 2730 | 2731 | 2732 | 2733 | 2734 | 2735 |
| AB | 2736 | 2737 | 2738 | 2739 | 2740 | 2741 | 2742 | 2743 | 2744 | 2745 | 2746 | 2747 | 2748 | 2749 | 2750 | 2751 |
| AC | 2752 | 2753 | 2754 | 2755 | 2756 | 2757 | 2758 | 2759 | 2760 | 2761 | 2762 | 2763 | 2764 | 2765 | 2766 | 2767 |
| AD | 2768 | 2769 | 2770 | 2771 | 2772 | 2773 | 2774 | 2775 | 2776 | 2777 | 2778 | 2779 | 2780 | 2781 | 2782 | 2783 |
| AE | 2784 | 2785 | 2786 | 2787 | 2788 | 2789 | 2790 | 2791 | 2792 | 2793 | 2794 | 2795 | 2796 | 2797 | 2798 | 2799 |
| AF | 2800 | 2801 | 2802 | 2803 | 2804 | 2805 | 2806 | 2807 | 2808 | 2809 | 2810 | 2811 | 2812 | 2813 | 2814 | 2815 |
| B0 | 2816 | 2817 | 2818 | 2819 | 2820 | 2821 | 2822 | 2823 | 2824 | 2825 | 2826 | 2827 | 2828 | 2829 | 2830 | 2831 |
| B1 | 2832 | 2833 | 2834 | 2835 | 2836 | 2837 | 2838 | 2839 | 2840 | 2841 | 2842 | 2843 | 2844 | 2845 | 2846 | 2847 |
| B2 | 2848 | 2849 | 2850 | 2851 | 2852 | 2853 | 2854 | 2855 | 2856 | 2857 | 2858 | 2859 | 2860 | 2861 | 2862 | 2863 |
| B3 | 2864 | 2865 | 2866 | 2867 | 2868 | 2869 | 2870 | 2871 | 2872 | 2873 | 2874 | 2875 | 2876 | 2877 | 2878 | 2879 |
| B4 | 2880 | 2881 | 2882 | 2883 | 2884 | 2885 | 2886 | 2887 | 2888 | 2889 | 2890 | 2891 | 2892 | 2893 | 2894 | 2895 |
| B5 | 2896 | 2897 | 2898 | 2899 | 2900 | 2901 | 2902 | 2903 | 2904 | 2905 | 2906 | 2907 | 2908 | 2909 | 2910 | 2911 |
| B6 | 2912 | 2913 | 2914 | 2915 | 2916 | 2917 | 2918 | 2919 | 2920 | 2921 | 2922 | 2923 | 2924 | 2925 | 2926 | 2927 |
| B7 | 2928 | 2929 | 2930 | 2931 | 2932 | 2933 | 2934 | 2935 | 2936 | 2937 | 2938 | 2939 | 2940 | 2941 | 2942 | 2943 |
| B8 | 2944 | 2945 | 2946 | 2947 | 2948 | 2949 | 2950 | 2951 | 2952 | 2953 | 2954 | 2955 | 2956 | 2957 | 2958 | 2959 |
| B9 | 2960 | 2961 | 2962 | 2963 | 2964 | 2965 | 2966 | 2967 | 2968 | 2969 | 2970 | 2971 | 2972 | 2973 | 2974 | 2975 |
| BA | 2976 | 2977 | 2978 | 2979 | 2980 | 2981 | 2982 | 2983 | 2984 | 2985 | 2986 | 2987 | 2988 | 2989 | 2990 | 2991 |
| BB | 2992 | 2993 | 2994 | 2995 | 2996 | 2997 | 2998 | 2999 | 3000 | 3001 | 3002 | 3003 | 3004 | 3005 | 3006 | 3007 |
| BC | 3008 | 3009 | 3010 | 3011 | 3012 | 3013 | 3014 | 3015 | 3016 | 3017 | 3018 | 3019 | 3020 | 3021 | 3022 | 3023 |
| BD | 3024 | 3025 | 3026 | 3027 | 3028 | 3029 | 3030 | 3031 | 3032 | 3033 | 3034 | 3035 | 3036 | 3037 | 3038 | 3039 |
| BE | 3040 | 3041 | 3042 | 3043 | 3044 | 3045 | 3046 | 3047 | 3048 | 3049 | 3050 | 3051 | 3052 | 3053 | 3054 | 3055 |
| BF | 3056 | 3057 | 3058 | 3059 | 3060 | 3061 | 3062 | 3063 | 3064 | 3065 | 3066 | 3067 | 3068 | 3069 | 3070 | 3071 |

Table C-1. Hexadecimal-Decimal Integer Conversion (*continued*)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|----|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| C0 | 3072 | 3073 | 3074 | 3075 | 3076 | 3077 | 3078 | 3079 | 3080 | 3081 | 3082 | 3083 | 3084 | 3085 | 3086 | 3087 |
| C1 | 3088 | 3089 | 3090 | 3091 | 3092 | 3093 | 3094 | 3095 | 3096 | 3097 | 3098 | 3099 | 3100 | 3101 | 3102 | 3103 |
| C2 | 3104 | 3105 | 3106 | 3107 | 3108 | 3109 | 3110 | 3111 | 3112 | 3113 | 3114 | 3115 | 3116 | 3117 | 3118 | 3119 |
| C3 | 3120 | 3121 | 3122 | 3123 | 3124 | 3125 | 3126 | 3127 | 3128 | 3129 | 3130 | 3131 | 3132 | 3133 | 3134 | 3135 |
| C4 | 3136 | 3137 | 3138 | 3139 | 3140 | 3141 | 3142 | 3143 | 3144 | 3145 | 3146 | 3147 | 3148 | 3149 | 3150 | 3151 |
| C5 | 3152 | 3153 | 3154 | 3155 | 3156 | 3157 | 3158 | 3159 | 3160 | 3161 | 3162 | 3163 | 3164 | 3165 | 3166 | 3167 |
| C6 | 3168 | 3169 | 3170 | 3171 | 3172 | 3173 | 3174 | 3175 | 3176 | 3177 | 3178 | 3179 | 3180 | 3181 | 3182 | 3183 |
| C7 | 3184 | 3185 | 3186 | 3187 | 3188 | 3189 | 3190 | 3191 | 3192 | 3193 | 3194 | 3195 | 3196 | 3197 | 3198 | 3199 |
| C8 | 3200 | 3201 | 3202 | 3203 | 3204 | 3205 | 3206 | 3207 | 3208 | 3209 | 3210 | 3211 | 3212 | 3213 | 3214 | 3215 |
| C9 | 3216 | 3217 | 3218 | 3219 | 3220 | 3221 | 3222 | 3223 | 3224 | 3225 | 3226 | 3227 | 3228 | 3229 | 3230 | 3231 |
| CA | 3232 | 3233 | 3234 | 3235 | 3236 | 3237 | 3238 | 3239 | 3240 | 3241 | 3242 | 3243 | 3244 | 3245 | 3246 | 3247 |
| CB | 3248 | 3249 | 3250 | 3251 | 3252 | 3253 | 3254 | 3255 | 3256 | 3257 | 3258 | 3259 | 3260 | 3261 | 3262 | 3263 |
| CC | 3264 | 3265 | 3266 | 3267 | 3268 | 3269 | 3270 | 3271 | 3272 | 3273 | 3274 | 3275 | 3276 | 3277 | 3278 | 3279 |
| CD | 3280 | 3281 | 3282 | 3283 | 3284 | 3285 | 3286 | 3287 | 3288 | 3289 | 3290 | 3291 | 3292 | 3293 | 3294 | 3295 |
| CE | 3296 | 3297 | 3298 | 3299 | 3300 | 3301 | 3302 | 3303 | 3304 | 3305 | 3306 | 3307 | 3308 | 3309 | 3310 | 3311 |
| CF | 3312 | 3313 | 3314 | 3315 | 3316 | 3317 | 3318 | 3319 | 3320 | 3321 | 3322 | 3323 | 3324 | 3325 | 3326 | 3327 |
| D0 | 3328 | 3329 | 3330 | 3331 | 3332 | 3333 | 3334 | 3335 | 3336 | 3337 | 3338 | 3339 | 3340 | 3341 | 3342 | 3343 |
| D1 | 3344 | 3345 | 3346 | 3347 | 3348 | 3349 | 3350 | 3351 | 3352 | 3353 | 3354 | 3355 | 3356 | 3357 | 3358 | 3359 |
| D2 | 3360 | 3361 | 3362 | 3363 | 3364 | 3365 | 3366 | 3367 | 3368 | 3369 | 3370 | 3371 | 3372 | 3373 | 3374 | 3375 |
| D3 | 3376 | 3377 | 3378 | 3379 | 3380 | 3381 | 3382 | 3383 | 3384 | 3385 | 3386 | 3387 | 3388 | 3389 | 3390 | 3391 |
| D4 | 3392 | 3393 | 3394 | 3395 | 3396 | 3397 | 3398 | 3399 | 3400 | 3401 | 3402 | 3403 | 3404 | 3405 | 3406 | 3407 |
| D5 | 3408 | 3409 | 3410 | 3411 | 3412 | 3413 | 3414 | 3415 | 3416 | 3417 | 3418 | 3419 | 3420 | 3421 | 3422 | 3423 |
| D6 | 3424 | 3425 | 3426 | 3427 | 3428 | 3429 | 3430 | 3431 | 3432 | 3433 | 3434 | 3435 | 3436 | 3437 | 3438 | 3439 |
| D7 | 3440 | 3441 | 3442 | 3443 | 3444 | 3445 | 3446 | 3447 | 3448 | 3449 | 3450 | 3451 | 3452 | 3453 | 3454 | 3455 |
| D8 | 3456 | 3457 | 3458 | 3459 | 3460 | 3461 | 3462 | 3463 | 3464 | 3465 | 3466 | 3467 | 3468 | 3469 | 3470 | 3471 |
| D9 | 3472 | 3473 | 3474 | 3475 | 3476 | 3477 | 3478 | 3479 | 3480 | 3481 | 3482 | 3483 | 3484 | 3485 | 3486 | 3487 |
| DA | 3488 | 3489 | 3490 | 3491 | 3492 | 3493 | 3494 | 3495 | 3496 | 3497 | 3498 | 3499 | 3500 | 3501 | 3502 | 3503 |
| DB | 3504 | 3505 | 3506 | 3507 | 3508 | 3509 | 3510 | 3511 | 3512 | 3513 | 3514 | 3515 | 3516 | 3517 | 3518 | 3519 |
| DC | 3520 | 3521 | 3522 | 3523 | 3524 | 3525 | 3526 | 3527 | 3528 | 3529 | 3530 | 3531 | 3532 | 3533 | 3534 | 3535 |
| DD | 3536 | 3537 | 3538 | 3539 | 3540 | 3541 | 3542 | 3543 | 3544 | 3545 | 3546 | 3547 | 3548 | 3549 | 3550 | 3551 |
| DE | 3552 | 3553 | 3554 | 3555 | 3556 | 3557 | 3558 | 3559 | 3560 | 3561 | 3562 | 3563 | 3564 | 3565 | 3566 | 3567 |
| DF | 3568 | 3569 | 3570 | 3571 | 3572 | 3573 | 3574 | 3575 | 3576 | 3577 | 3578 | 3579 | 3580 | 3581 | 3582 | 3583 |
| E0 | 3584 | 3585 | 3586 | 3587 | 3588 | 3589 | 3590 | 3591 | 3592 | 3593 | 3594 | 3595 | 3596 | 3597 | 3598 | 3599 |
| E1 | 3600 | 3601 | 3602 | 3603 | 3604 | 3605 | 3606 | 3607 | 3608 | 3609 | 3610 | 3611 | 3612 | 3613 | 3614 | 3615 |
| E2 | 3616 | 3617 | 3618 | 3619 | 3620 | 3621 | 3622 | 3623 | 3624 | 3625 | 3626 | 3627 | 3628 | 3629 | 3630 | 3631 |
| E3 | 3632 | 3633 | 3634 | 3635 | 3636 | 3637 | 3638 | 3639 | 3640 | 3641 | 3642 | 3643 | 3644 | 3645 | 3646 | 3647 |
| E4 | 3648 | 3649 | 3650 | 3651 | 3652 | 3653 | 3654 | 3655 | 3656 | 3657 | 3658 | 3659 | 3660 | 3661 | 3662 | 3663 |
| E5 | 3664 | 3665 | 3666 | 3667 | 3668 | 3669 | 3670 | 3671 | 3672 | 3673 | 3674 | 3675 | 3676 | 3677 | 3678 | 3679 |
| E6 | 3680 | 3681 | 3682 | 3683 | 3684 | 3685 | 3686 | 3687 | 3688 | 3689 | 3690 | 3691 | 3692 | 3693 | 3694 | 3695 |
| E7 | 3696 | 3697 | 3698 | 3699 | 3700 | 3701 | 3702 | 3703 | 3704 | 3705 | 3706 | 3707 | 3708 | 3709 | 3710 | 3711 |
| E8 | 3712 | 3713 | 3714 | 3715 | 3716 | 3717 | 3718 | 3719 | 3720 | 3721 | 3722 | 3723 | 3724 | 3725 | 3726 | 3727 |
| E9 | 3728 | 3729 | 3730 | 3731 | 3732 | 3733 | 3734 | 3735 | 3736 | 3737 | 3738 | 3739 | 3740 | 3741 | 3742 | 3743 |
| EA | 3744 | 3745 | 3746 | 3747 | 3748 | 3749 | 3750 | 3751 | 3752 | 3753 | 3754 | 3755 | 3756 | 3757 | 3758 | 3759 |
| EB | 3760 | 3761 | 3762 | 3763 | 3764 | 3765 | 3766 | 3767 | 3768 | 3769 | 3770 | 3771 | 3772 | 3773 | 3774 | 3775 |
| EC | 3776 | 3777 | 3778 | 3779 | 3780 | 3781 | 3782 | 3783 | 3784 | 3785 | 3786 | 3787 | 3788 | 3789 | 3790 | 3791 |
| EC | 3792 | 3793 | 3794 | 3795 | 3796 | 3797 | 3798 | 3799 | 3800 | 3801 | 3802 | 3803 | 3804 | 3805 | 3806 | 3807 |
| EE | 3808 | 3809 | 3810 | 3811 | 3812 | 3813 | 3814 | 3815 | 3816 | 3817 | 3818 | 3819 | 3820 | 3821 | 3822 | 3823 |
| EF | 3824 | 3825 | 3826 | 3827 | 3828 | 3829 | 3830 | 3831 | 3832 | 3833 | 3834 | 3835 | 3836 | 3837 | 3838 | 3839 |

Table C-1. Hexadecimal-Decimal Integer Conversion (*continued*)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|----|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| F0 | 3840 | 3841 | 3842 | 3843 | 3844 | 3845 | 3846 | 3847 | 3848 | 3849 | 3850 | 3851 | 3852 | 3853 | 3854 | 3855 |
| F1 | 3856 | 3857 | 3858 | 3859 | 3860 | 3861 | 3862 | 3863 | 3864 | 3865 | 3866 | 3867 | 3868 | 3869 | 3870 | 3871 |
| F2 | 3872 | 3873 | 3874 | 3875 | 3876 | 3877 | 3878 | 3879 | 3880 | 3881 | 3882 | 3883 | 3884 | 3885 | 3886 | 3887 |
| F3 | 3888 | 3889 | 3890 | 3891 | 3892 | 3893 | 3894 | 3895 | 3896 | 3897 | 3898 | 3899 | 3900 | 3901 | 3902 | 3903 |
| F4 | 3904 | 3905 | 3906 | 3907 | 3908 | 3909 | 3910 | 3911 | 3912 | 3913 | 3914 | 3915 | 3916 | 3917 | 3918 | 3919 |
| F5 | 3920 | 3921 | 3922 | 3923 | 3924 | 3925 | 3926 | 3927 | 3928 | 3929 | 3930 | 3931 | 3932 | 3933 | 3934 | 3935 |
| F6 | 3936 | 3937 | 3938 | 3939 | 3940 | 3941 | 3942 | 3943 | 3944 | 3945 | 3946 | 3947 | 3948 | 3949 | 3950 | 3951 |
| F7 | 3952 | 3953 | 3954 | 3955 | 3956 | 3957 | 3958 | 3959 | 3960 | 3961 | 3962 | 3963 | 3964 | 3965 | 3966 | 3967 |
| F8 | 3968 | 3969 | 3970 | 3971 | 3972 | 3973 | 3974 | 3975 | 3976 | 3977 | 3978 | 3979 | 3980 | 3981 | 3982 | 3983 |
| F9 | 3984 | 3985 | 3986 | 3987 | 3988 | 3989 | 3990 | 3991 | 3992 | 3993 | 3994 | 3995 | 3996 | 3997 | 3998 | 3999 |
| FA | 4000 | 4001 | 4002 | 4003 | 4004 | 4005 | 4006 | 4007 | 4008 | 4009 | 4010 | 4011 | 4012 | 4013 | 4014 | 4015 |
| FB | 4016 | 4017 | 4018 | 4019 | 4020 | 4021 | 4022 | 4023 | 4024 | 4025 | 4026 | 4027 | 4028 | 4029 | 4030 | 4031 |
| FC | 4032 | 4033 | 4034 | 4035 | 4036 | 4037 | 4038 | 4039 | 4040 | 4041 | 4042 | 4043 | 4044 | 4045 | 4046 | 4047 |
| FD | 4048 | 4049 | 4050 | 4051 | 4052 | 4053 | 4054 | 4055 | 4056 | 4057 | 4058 | 4059 | 4060 | 4061 | 4062 | 4063 |
| FE | 4064 | 4065 | 4066 | 4067 | 4068 | 4069 | 4070 | 4071 | 4072 | 4073 | 4074 | 4075 | 4076 | 4077 | 4078 | 4079 |
| FF | 4080 | 4081 | 4082 | 4083 | 4084 | 4085 | 4086 | 4087 | 4088 | 4089 | 4090 | 4091 | 4092 | 4093 | 4094 | 4095 |

Table C-2. Conversion Values

| Hexadecimal | Decimal | Hexadecimal | Decimal | Hexadecimal | Decimal | Hexadecimal | Decimal |
|-------------|---------|-------------|---------|-------------|-----------|-------------|------------|
| 01 000 | 4 096 | 11 000 | 69 632 | 30 000 | 196 608 | 400 000 | 4 194 304 |
| 02 000 | 8 192 | 12 000 | 73 728 | 40 000 | 262 144 | 500 000 | 5 242 880 |
| 03 000 | 12 288 | 13 000 | 77 824 | 50 000 | 327 680 | 600 000 | 6 291 456 |
| 04 000 | 16 384 | 14 000 | 81 920 | 60 000 | 393 216 | 700 000 | 7 340 032 |
| 05 000 | 20 480 | 15 000 | 86 016 | 70 000 | 458 752 | 800 000 | 8 388 608 |
| 06 000 | 24 576 | 16 000 | 90 112 | 80 000 | 524 288 | 900 000 | 9 437 184 |
| 07 000 | 28 672 | 17 000 | 94 208 | 90 000 | 589 824 | A00 000 | 10 485 760 |
| 08 000 | 32 768 | 18 000 | 98 304 | A0 000 | 655 360 | 800 000 | 11 534 336 |
| 09 000 | 36 864 | 19 000 | 102 400 | 80 000 | 720 896 | C00 000 | 12 582 912 |
| 0A 000 | 40 960 | 1A 000 | 106 496 | C0 000 | 786 432 | D00 000 | 13 631 488 |
| 0B 000 | 45 056 | 1B 000 | 110 592 | D0 000 | 851 968 | E00 000 | 14 680 064 |
| 0C 000 | 49 152 | 1C 000 | 114 688 | E0 000 | 917 504 | F00 000 | 15 728 640 |
| 0D 000 | 53 248 | 1D 000 | 118 784 | F0 000 | 983 040 | 1 000 000 | 16 777 216 |
| 0E 000 | 57 344 | 1E 000 | 122 880 | 100 000 | 1 048 576 | 2 000 000 | 33 554 432 |
| 0F 000 | 61 440 | 1F 000 | 126 976 | 200 000 | 2 097 152 | | |
| 10 000 | 65 536 | 20 000 | 131 072 | 300 000 | 3 145 728 | | |

Hexadecimal fractions may be converted to decimal fractions as follows:

1. Express the hexadecimal fraction as an integer times 16^{-n} , where n is the number of significant hexadecimal places to the right of the hexadecimal point.

$$0.CA9BF3_{16} = CA9 BF3_{16} \times 16^{-6}$$

2. Find the decimal equivalent of the hexadecimal integer.

$$CA9 BF_{16} = 13\ 278\ 195_{10}$$

3. Multiply the decimal equivalent by 16^{-n}

| | | |
|-------|-----|----------------------|
| 13 | 278 | 195 |
| × 596 | 046 | 448×10^{16} |
| 0.791 | 442 | 096 ₁₀ |

Decimal fractions may be converted to hexadecimal fractions by successively multiplying the decimal fraction by 16_{10} . After each multiplication, the integer portion is removed to form a hexadecimal fraction by building to the right of the hexadecimal point. However, since decimal arithmetic is used in this conversion, the integer portion of each product must be converted to hexadecimal numbers.

Example:

Convert 0.895_{10} to its hexadecimal equivalent

| |
|-----------------------------|
| 0.895 |
| 16 |
| 14.320 |
| 16 |
| 5.120 |
| 16 |
| 1.920 |
| 16 |
| 0.E51E ₁₆ 14.720 |

Functions that are not intrinsic to C-128 BASIC may be calculated as in Table C-3.

Table C-3. Deriving Mathematical Functions

| Function | VIC BASIC Equivalent |
|------------------------------|--|
| Secant | $SEC(X) = 1/COS(X)$ |
| Cosecant | $CSC(X) = 1/SIN(X)$ |
| Cotangent | $COT(X) = 1/TAN(X)$ |
| Inverse sine | $ARCSIN(X) = ATN(X/SQR(-X*X + 1))$ |
| Inverse cosine | $ARCCOS(X) = -ATN(X/SQR(-X*X + 1)) + \pi/2$ |
| Inverse secant | $ARCSEC(X) = ATN(X/SQR(X*X - 1))$ |
| Inverse cosecant | $ARCCSC(X) = ATN(X/SQR(X*X - 1)) + (SGN(X) - 1)*\pi/2$ |
| Inverse cotangent | $ARCOT(X) = ATN(X) + \pi/2$ |
| Hyperbolic sine | $SINE(X) = (EXP(X) - EXP(-X))/2$ |
| Hyperbolic cosine | $COSH(X) = (EXP(X) + EXP(-X))/2$ |
| Hyperbolic tangent | $TANH(X) = EXP(-X)/EXP(X) + EXP(-X)*2 + 1$ |
| Hyperbolic secant | $SECH(X) = 2/(EXP(X) + EXP(-X))$ |
| Hyperbolic cosecant | $CSCH(X) = 2/(EXP(X) - EXP(-X))$ |
| Hyperbolic cotangent | $COTH(X) = EXP(-X)/(EXP(X) - EXP(-X)*2 + 1)$ |
| Inverse hyperbolic sine | $ARCSINH(X) = LOG(X + SQR(X*X + 1))$ |
| Inverse hyperbolic cosine | $ARCCOSH(X) = LOG(X + SQR(X*X - 1))$ |
| Inverse hyperbolic tangent | $ARCTANH(X) = LOG((1 + X)/(1 - X))/2$ |
| Inverse hyperbolic secant | $ARCSECH(X) = LOG((SQR(-X*X + 1) + 1/X))$ |
| Inverse hyperbolic cosecant | $ARCCSCH(X) = LOG((SGN(X)*SQR(X*X + 1)/X))$ |
| Inverse hyperbolic cotangent | $ARCCOTH(X) = LOG((X + 1)/(X - 1))/2$ |

Appendix D

Character Sets and Graphic Characters

Two of the more powerful functions of the C-128 are its display and sound generation capabilities. The tables in this appendix cover all of the C-128 character codes, screen POKE values, and sound register equivalents.

Table D-1 covers all of the characters and functions that are displayed using the CHR\$ instruction. In many instances, the use of the CHR\$ function is optional; however, some functions, such as RETURN and RUN/STOP, are not programmable with the PRINT function. To program using these functions you will need to use the CHR\$ function and the codes in this table.

The codes used in the CHR\$ instruction are not the same as those used in the POKE-to-screen commands. The codes shown in

Table D-2 are listed in the same order as the characters in memory. Notice that all of the control characters are omitted from this list. This is because there is no display code for them; control codes use the codes of standard reverse characters (for example, reverse-heart for CLR HOME).

Table D-3 presents the POKE values and frequencies of musical notes.

Table D-1. C-128 Character Codes
























| Prints | CHRS | Prints | CHRS | Prints | CHRS | Prints | CHRS |
|------------------|------|----------------------|------|---|------|---|------|
| | 0 | RETURN | 13 | | 27 |  | 41 |
| | 1 | SWITCH | 14 | RED | 28 |  | 42 |
| | 2 | TO LOWER- CASE | 15 | CURSOR - | 29 |  | 43 |
| | 3 | | 16 | GRN | 30 |  | 44 |
| | 4 | CURSOR | 17 | BLU | 31 |  | 45 |
| WHT | 5 | RVS ON | 18 |  | 32 |  | 46 |
| | 6 | CLR HOME | 19 |  | 33 |  | 47 |
| | 7 | INS DEL | 20 |  | 34 |  | 48 |
| DISABLE SHIFT | 8 | | 21 |  | 35 |  | 49 |
| | | | 22 |  | 36 |  | 50 |
| ENABLE SHIFT | 9 | | 23 |  | 37 |  | 51 |
| | 10 | | 24 |  | 38 |  | 52 |
| | 11 | | 25 |  | 39 |  | 53 |
| | 12 | | 26 |  | 40 |  | 54 |

Table D-1. C-128 Character Codes (*continued*)

| Prints | CHRS | Prints | CHRS | Prints | CHRS | Prints | CHRS |
|--------|------|--------|------|--------|------|--------|------|
| | 55 | | 75 | | 95 | | 115 |
| | 56 | | 76 | | 96 | | 116 |
| | 57 | | 77 | | 97 | | 117 |
| | 58 | | 78 | | 98 | | 118 |
| | 59 | | 79 | | 99 | | 119 |
| | 60 | | 80 | | 100 | | 120 |
| | 61 | | 81 | | 101 | | 121 |
| | 62 | | 82 | | 102 | | 122 |
| | 63 | | 83 | | 103 | | 123 |
| | 64 | | 84 | | 104 | | 124 |
| | 65 | | 85 | | 105 | | 125 |
| | 66 | | 86 | | 106 | | 126 |
| | 67 | | 87 | | 107 | | 127 |
| | 68 | | 88 | | 108 | | 128 |
| | 69 | | 89 | | 109 | ORANGE | 129 |
| | 70 | | 90 | | 110 | | 130 |
| | 71 | | 91 | | 111 | | 131 |
| | 72 | | 92 | | 112 | | 132 |
| | 73 | | 93 | | 113 | F1 | 133 |
| | 74 | | 94 | | 114 | F3 | 134 |

Table D-1. C-128 Character Codes (*continued*)

| Prints | CHRS | Prints | CHRS | Prints | CHRS | Prints | CHRS |
|------------------------|------|-----------|------|--------|------|--------|------|
| F5 | 135 | LT BLUE | 154 | | 174 | | 194 |
| F7 | 136 | LT GRAY | 155 | | 175 | | 195 |
| F2 | 137 | PUR | 156 | | 176 | | 196 |
| F4 | 138 | CRSR -- | 157 | | 177 | | 197 |
| F6 | 139 | YEL | 158 | | 178 | | 198 |
| F8 | 140 | CYN SPACE | 159 | | 179 | | 199 |
| SHIFT- RETURN | 141 | | 160 | | 180 | | 200 |
| SWITCH TO UPPERCASE | 142 | | 161 | | 181 | | 201 |
| | 143 | | 162 | | 182 | | 202 |
| BLK | 144 | | 163 | | 183 | | 203 |
| CURSOR ↑ | 145 | | 164 | | 184 | | 204 |
| RVS OFF | 146 | | 165 | | 185 | | 205 |
| CLR HOME | 147 | | 166 | | 186 | | 206 |
| INS DEL | 148 | | 167 | | 187 | | 207 |
| BROWN | 149 | | 168 | | 188 | | 208 |
| LT RED | 150 | | 169 | | 189 | | 209 |
| DK GRAY | 151 | | 170 | | 190 | | 210 |
| MED GRAY | 152 | | 171 | | 191 | | 211 |
| LT GREEN | 153 | | 172 | | 192 | | 212 |
| | | | 173 | | 193 | | 213 |

Table D-1. C-128 Character Codes (*continued*)











































| Prints | CHRS | Prints | CHRS | Prints | CHRS | Prints | CHRS |
|---|------|---|------|---|------|---|------|
|  | 214 |  | 225 |  | 236 |  | 247 |
|  | 215 |  | 226 |  | 237 |  | 248 |
|  | 216 |  | 227 |  | 238 |  | 249 |
|  | 217 |  | 228 |  | 239 |  | 250 |
|  | 218 |  | 229 |  | 240 |  | 251 |
|  | 219 |  | 230 |  | 241 |  | 252 |
|  | 220 |  | 231 |  | 242 |  | 253 |
|  | 221 |  | 232 |  | 243 |  | 254 |
|  | 222 |  | 233 |  | 244 |  | 255 |
|  | 223 |  | 234 |  | 245 | | |
|  | 224 |  | 235 |  | 246 | | |










Table D-2. Screen Codes

| Set 1 | Set 2 | POKE | Set 1 | Set 2 | POKE | Set 1 | Set 2 | POKE |
|-------|-------|------|-------|-------|------|-------|-------|------|
| | | 0 | | | 20 | | | 40 |
| | | 1 | | | 21 | | | 41 |
| | | 2 | | | 22 | | | 42 |
| | | 3 | | | 23 | | | 43 |
| | | 4 | | | 24 | | | 44 |
| | | 5 | | | 25 | | | 45 |
| | | 6 | | | 26 | | | 46 |
| | | 7 | | | 27 | | | 47 |
| | | 8 | | | 28 | | | 48 |
| | | 9 | | | 29 | | | 49 |
| | | 10 | | | 30 | | | 50 |
| | | 11 | | | 31 | | | 51 |
| | | 12 | | | 32 | | | 52 |
| | | 13 | | | 33 | | | 53 |
| | | 14 | | | 34 | | | 54 |
| | | 15 | | | 35 | | | 55 |
| | | 16 | | | 36 | | | 56 |
| | | 17 | | | 37 | | | 57 |
| | | 18 | | | 38 | | | 58 |
| | | 19 | | | 39 | | | 59 |

Table D-2. Screen Codes (*continued*)

| Set 1 | Set 2 | POKE | Set 1 | Set 2 | POKE | Set 1 | Set 2 | POKE |
|-------|-------|------|-------|-------|------|-------|-------|------|
| | | 60 | | | 80 | | | 100 |
| | | 61 | | | 81 | | | 101 |
| | | 62 | | | 82 | | | 102 |
| | | 63 | | | 83 | | | 103 |
| | | 64 | | | 84 | | | 104 |
| | | 65 | | | 85 | | | 105 |
| | | 66 | | | 86 | | | 106 |
| | | 67 | | | 87 | | | 107 |
| | | 68 | | | 88 | | | 108 |
| | | 69 | | | 89 | | | 109 |
| | | 70 | | | 90 | | | 110 |
| | | 71 | | | 91 | | | 111 |
| | | 72 | | | 92 | | | 112 |
| | | 73 | | | 93 | | | 113 |
| | | 74 | | | 94 | | | 114 |
| | | 75 | | | 95 | | | 115 |
| | | 76 | | | 96 | | | 116 |
| | | 77 | | | 97 | | | 117 |
| | | 78 | | | 98 | | | 118 |
| | | 79 | | | 99 | | | 119 |

Table D-2. Screen Codes (*continued*)

| Set 1 | Set 2 | POKE | Set 1 | Set 2 | POKE | Set 1 | Set 2 | POKE |
|---|---|------|---|-------|------|---|-------|------|
|  | | 120 |  | | 123 |  | | 126 |
|  | | 121 |  | | 124 |  | | 127 |
|  |  | 122 |  | | 125 | | | |

NOTE: Numbers 128 through 255 of the screen codes are reverse video of numbers 0 through 127.

Table D-3. POKE Values, Frequencies, and Musical Equivalents

| Upper Tone POKE Value | Lower Tone POKE Value | Decimal Values | Frequency Produced | Musical Note |
|--------------------------|--------------------------|-------------------|-----------------------|-----------------|
| OCTAVE ONE | | | | |
| 1 | 12 | 268 | 16 Hz | C |
| 1 | 28 | 284 | 17 Hz | C# |
| 1 | 45 | 301 | 18 Hz | D |
| 1 | 62 | 318 | 19 Hz | D# |
| 1 | 81 | 337 | 21 Hz | E |
| 1 | 101 | 357 | 22 Hz | F |
| 1 | 123 | 379 | 23 Hz | F# |
| 1 | 145 | 401 | 24 Hz | G |
| 1 | 169 | 425 | 25 Hz | G# |
| 1 | 195 | 451 | 27 Hz | A |
| 1 | 221 | 477 | 29 Hz | A# |
| 1 | 250 | 506 | 31 Hz | B |
| OCTAVE TWO | | | | |
| 2 | 24 | 536 | 32 Hz | C |
| 2 | 56 | 568 | 34 Hz | C# |
| 2 | 90 | 602 | 37 Hz | D |
| 2 | 125 | 637 | 39 Hz | D# |
| 2 | 163 | 675 | 41 Hz | E |
| 2 | 203 | 715 | 44 Hz | F |
| 2 | 246 | 758 | 46 Hz | F# |
| 3 | 35 | 803 | 49 Hz | G |

Table D-3. POKE Values, Frequencies, and Musical Equivalents
(continued)

| Upper Tone POKE Value | Lower Tone POKE Value | Decimal Values | Frequency Produced | Musical Note |
|--------------------------|--------------------------|-------------------|-----------------------|-----------------|
| 3 | 83 | 851 | 52 Hz | G# |
| 3 | 134 | 902 | 55 Hz | A |
| 3 | 187 | 955 | 58 Hz | A# |
| 3 | 244 | 1012 | 62 Hz | B |
| OCTAVE THREE | | | | |
| 4 | 48 | 1072 | 65 Hz | C |
| 4 | 112 | 1136 | 69 Hz | C# |
| 4 | 180 | 1204 | 73 Hz | D |
| 4 | 251 | 1275 | 76 Hz | D# |
| 5 | 71 | 1351 | 82 Hz | E |
| 5 | 151 | 1431 | 87 Hz | F |
| 5 | 237 | 1517 | 92 Hz | F# |
| 6 | 71 | 1607 | 98 Hz | G |
| 6 | 167 | 1703 | 104 Hz | G# |
| 7 | 12 | 1804 | 110 Hz | A |
| 7 | 119 | 1911 | 117 Hz | A# |
| 7 | 233 | 2025 | 123 Hz | B |
| OCTAVE FOUR | | | | |
| 8 | 97 | 2145 | 131 Hz | C |
| 8 | 225 | 2273 | 139 Hz | C# |
| 9 | 104 | 2408 | 147 Hz | D |
| 9 | 247 | 2551 | 156 Hz | D# |
| 10 | 143 | 2703 | 165 Hz | E |
| 11 | 47 | 2863 | 175 Hz | F |
| 11 | 218 | 3034 | 185 Hz | F# |
| 12 | 142 | 3214 | 196 Hz | G |
| 13 | 77 | 3405 | 208 Hz | G# |
| 14 | 24 | 3608 | 220 Hz | A |
| 14 | 238 | 3822 | 233 Hz | A# |
| 15 | 210 | 4050 | 247 Hz | B |
| OCTAVE FIVE | | | | |
| 16 | 195 | 4291 | 262 Hz | C |
| 17 | 194 | 4546 | 277 Hz | C# |
| 18 | 208 | 4816 | 294 Hz | D |
| 19 | 238 | 5102 | 311 Hz | D# |
| 21 | 30 | 5406 | 330 Hz | E |
| 22 | 95 | 5727 | 349 Hz | F |
| 23 | 180 | 6068 | 370 Hz | F# |
| 25 | 29 | 6429 | 392 Hz | G |
| 26 | 155 | 6811 | 415 Hz | G# |

Table D-3. POKE Values, Frequencies, and Musical Equivalents
(continued)

| Upper Tone POKE Value | Lower Tone POKE Value | Decimal Values | Frequency Produced | Musical Note |
|--------------------------|--------------------------|-------------------|-----------------------|-----------------|
| 28 | 48 | 7216 | 440 Hz | A |
| 29 | 221 | 7645 | 466 Hz | A# |
| 31 | 164 | 8100 | 494 Hz | B |
| OCTAVE SIX | | | | |
| 33 | 134 | 8582 | 523 Hz | C |
| 35 | 132 | 9092 | 554 Hz | C# |
| 37 | 161 | 9633 | 587 Hz | D |
| 39 | 221 | 10205 | 622 Hz | D# |
| 42 | 60 | 10812 | 659 Hz | E |
| 44 | 191 | 11455 | 698 Hz | F |
| 47 | 104 | 12136 | 740 Hz | F# |
| 50 | 58 | 12858 | 784 Hz | G |
| 53 | 55 | 13623 | 831 Hz | G# |
| 56 | 97 | 14433 | 880 Hz | A |
| 59 | 187 | 15291 | 932 Hz | A# |
| 63 | 72 | 16200 | 988 Hz | B |
| OCTAVE SEVEN | | | | |
| 67 | 12 | 17164 | 1046 Hz | C |
| 71 | 8 | 18184 | 1109 Hz | C# |
| 75 | 66 | 19266 | 1175 Hz | D |
| 79 | 187 | 20411 | 1244 Hz | D# |
| 84 | 121 | 21625 | 1319 Hz | E |
| 89 | 127 | 22911 | 1397 Hz | F |
| 94 | 209 | 24273 | 1480 Hz | F# |
| 100 | 117 | 25717 | 1568 Hz | G |
| 106 | 110 | 27246 | 1661 Hz | G# |
| 112 | 194 | 28866 | 1760 Hz | A |
| 119 | 118 | 30582 | 1865 Hz | A# |
| 126 | 145 | 32401 | 1976 Hz | B |
| OCTAVE EIGHT | | | | |
| 134 | 24 | 34328 | 2093 Hz | C |
| 142 | 17 | 36369 | 2217 Hz | C# |
| 150 | 132 | 38532 | 2349 Hz | D |
| 159 | 119 | 40823 | 2489 Hz | D# |
| 168 | 242 | 43250 | 2637 Hz | E |
| 178 | 254 | 45822 | 2794 Hz | F |
| 189 | 163 | 48547 | 2960 Hz | F# |
| 200 | 234 | 51434 | 3136 Hz | G |
| 212 | 220 | 54492 | 3322 Hz | G# |
| 225 | 132 | 57732 | 3520 Hz | A |
| 238 | 237 | 61165 | 3729 Hz | A# |
| 253 | 34 | 64802 | 3951 Hz | B |

Appendix E

Displaying 80-column Text on a Monochrome Composite Monitor

The C-128's 80-column RGB video output can be viewed on a low-cost monochrome composite monitor that conforms to NTSC (National Television Standards Committee) specifications. By using a monochrome composite monitor, you avoid purchasing an expensive color monitor simply to view 80-column text.

A special adapter cable must be constructed to connect the C-128's RGB output to the monitor. You will need the following components, all available at any electronics store:

- A shielded video cable with an RCA-style plug (also called a "phono plug") at one end. A good quality audio patch cord will work equally well.
- A 9-pin, D-style subminiature connector that will mate with the C-128's RGB video output plug. (It will resemble the joystick port connectors on the side of the C-128.)
- A matching hood with two screws for the D connector. (Screws normally come with the hood.)
- A "pencil" soldering iron and 60-40 rosin-core solder.
- A small screwdriver that fits the hood screws.

Strip enough insulation off the outside of the cable so that 3/8 inch of copper is visible. Carefully unravel it and twist it into one conductor. Then strip 1/4 inch of the center insulation away so that the center conductor is visible. Solder the "braid" (the braided copper shield that surrounds the center conductor's insulation) to pin 1 or 2 (or both, for that matter) of the new D connector. Then carefully solder the center conductor of the cable to pin 7 of the connector. Make sure no "solder bridges" exist between pins; only pin 1 (or 2, or 1 and 2) and pin 7 should have anything connected to them. Put the plastic cover over the D connector, being careful not to short the two wires together or to move the wires so that they accidentally touch other terminals. Attach the D connector to its mate on the C-128 and use the mounting screws to secure it to the C-128 connector.

Connect the other end of the cable (the end with the RCA/phono plug) to the round input connector on the back of your composite monitor. After powering up the C-128 with the 80-column display active, adjust the brightness and contrast controls for the best image. You will probably find that the 80-column output requires different settings for these controls than does the VIC-II (40-column) output.

Index

A

ABS, 205, 226
Absolute kernal variables, 88
ACPTR, 301
Actual bit map, 132
Address register, 10
ALT key, 5
Alternate character set, 264
Angerhauser, Michael (author), 14
APPEND, 138
Apple, 351
ARG, 226, 227
ARGSGN, 227
Arithmetic routines, 53
Array variable, 16, 123, 153
ARYTAB, 124
ASC, 206
ASCII code, 206, 207, 255, 289, 303, 338, 362
Asterisk, 175, 290
Asynchronous communications interface
adapter (ACIA), 49

ATN, 206, 233
Attribute bit assignments, 264
Attribute disable register, 273
Attribute table start register, 272
Audio/video port pinout, 373
AUTO, 139
Auto line numbering, 139
Autoboot sequence, 12, 315
Autobooting configuration routine, 340
Autobooting programs, 338-340
Auto-incrementing, 276
Auto-insert mode, 256, 259
Autostart cartridge, 4, 12
Auxiliary characters, 185
Auxiliary parameter, 197

B

Background color register, 273
Background text display colors, 276

- BACKUP, 139
 - Band pass filter, 160
 - BANK, 140, 201
 - Bank 0 RAM usage, 88
 - Banked memory, 278
 - Base 10 numbers, 130
 - Base 2 numbers, 130
 - BASIC dictionary, 138-205
 - BASIC functions, 205-225
 - BASIC jump table, 226-233
 - BASIC math functions, 226-233
 - Basic program storage, 130-138
 - BASIC ROM, 1, 4, 8, 33
 - BASIC 7.0, 3, 21
 - interpreter, 16
 - BASIC string handling from machine language, 233-251
 - BASIC token set, 135-138
 - Baud rate, 362
 - BCD, 53
 - Becker, Achim (author), 14
 - BEGIN, 140
 - Bell tone, 257, 261
 - BEND, 140, 141
 - Binary numbers, 130
 - Bit map, 265, 317
 - graphics, 132, 278
 - image, 21
 - mode register, 273
 - scaling, 195
 - Blanks (spaces), 150, 173, 182
 - BLOAD, 141, 360, 361
 - Block allocation map (BAM), 13, 145, 316, 317
 - Block copy/fill register, 272
 - Block copy operation, 275
 - Block copy start address register, 273
 - Block cursor, 259
 - Block-fill operation, 274, 275
 - Boolean connectors, 168
 - BOOT, 141
 - BOOT_CALL, 303, 304, 338, 339
 - BOX, 142
 - Braid, 398
 - BREAK, 148
 - BRK vector, 52
 - BSAVE, 142
 - BSOUT, 52, 289, 291
 - Built-in monitor program, 365
 - BUMP, 207, 327
 - Burst mode, 313
 - commands, 341-345
 - data transfer, 341-349
 - Burst status byte, 349
 - Burst transfer routines, 322, 347
 - Butterfield, Jim, 60
 - Byte boundaries, 129
- C**
- C-64 BASIC, 1
 - C-64 mode, 4, 354
 - C64MODE, 301
 - Calling address, 290
 - CAPS LOCK key, 5, 8
 - Carriage return, 169, 170
 - Carry bit, 299
 - Cartridge port, 34
 - Cartridge ROM, 23
 - Cassette buffer, 233
 - Cassette data file format, 179-180
 - Cassette interface pinout, 375
 - Cassette program format, 174
 - Cassette unit format, 193, 202
 - CATALOG, 142, 154
 - CHAR, 142
 - Character blink rate register, 272
 - Character blinking, 264
 - Character codes, 388-391
 - Character generator, 30
 - Character set address register, 273
 - Character set loader routine, 277
 - CHGUTL, 324, 339
 - CHGUTL utility commands, 319-325
 - CHRS, 207, 262, 387
 - CHROUT, 52, 289, 291
 - CINT, 9, 46, 48, 49, 297
 - CIRCLE, 143
 - CLD, 53
 - CLI, 298
 - Clock cycles, 33
 - CLOSE, 144, 299
 - CLOSE_ALL, 301
 - CLR, 144, 154, 175, 193
 - CMD, 145, 173
 - Code transferred to common RAM, 113
 - Codes returned after job completed, 327
 - Cold reset, 12, 33, 314
 - COLLECT, 145, 196
 - COLLISION, 146
 - Colon, 150, 167
 - COLOR, 146-147
 - Color source parameter, 142
 - Color storage, 21
 - Comma, 150, 183, 185, 186
 - Command channel, 320, 345
 - Commodore CP/M, 353
 - COMMODORE key, 3, 12, 47, 314
 - Common RAM, 21, 36, 37, 43, 58
 - Complex interface adapter (CIA), 25, 29, 51
 - CONCAT, 148
 - Configuration Register (CR), 26, 29-31, 307
 - Console command processor (CCP), 353, 356
 - CONT, 148, 168, 173
 - Continuous print format, 183
 - COPY, 149
 - Copy-protected software, 314, 315, 322
 - COS, 208, 233
 - CP/M
 - start-up code, 355
 - user functions, 355
 - CP/M 2.2, 351, 358
 - CP/M 3.0, 351, 358, 359
 - CP/M Plus, 351, 353, 354, 356
 - booting, 354-355
 - memory usage, 355-358

- Cross-bank operations, 58
- Cross-bank transfer routines, 78
- Cursor end scan line register, 271
- Cursor mode register, 271
- Cursor position, 259, 271
- Cursor start scan line register, 271
- Cut-off frequency, 160

- D**
- DATA, 150, 168, 190
- Data register, 10
- DCLEAR, 150
- DCLOSE, 151, 301
- Debugging session, 297
- Debugging tool, 166
- DEC, 208
- Decimal mode flag, 49, 53
- Decimal number, 208, 209
 - converting to floating-point, 131
- Decimal points, 185, 186
- DEF FN, 151-152
- DEJA-VU, 47, 298
- DELETE, 152-153
- Deriving mathematical functions, 385
- Device number, 299
- Device selection, 287
- DIGIFONT, 276, 365
- Digital Research, 354
- DIM, 124, 125, 153
- Dimensions of the array, 124
- Direct memory access (DMA), 19
- Directory, 139
 - entry for file, 320
 - information, 318
 - sector, 319
- DIRECTORY, 154
- Disk compatibility, 11-14, 314-316
- Disk data file format, 180
- Disk drive command channel, 299
- Disk formats, supported by C-128 CP/M Plus, 360
- Disk name, 139, 165, 317
- Disk-copy programs, 314, 315
- Diskette drive format, 174, 194, 203
- Display address register, 271
- Display RAM usage, 263
- DLCHR, 305
- DLOAD, 154
- DMA-accessible RAM, 41
- DMA_CALL, 159, 302
- DMA-controller commands, 358
- Dollar signs, 185
- DOPEN, 156
- DOS (disk operating system), 11
 - command function, 323
 - error messages, 202
 - internal job codes and functions, 326
 - memory usage and buffer areas, 325
 - mismatch, 316
 - ROM, 13
- Double quotation mark, 150, 207
- Double-sided disk, 13
 - organization, 316-319

- DO-WHILE/UNTIL, 158
- Downward scroll, 258
- DRAW, 156-157
- Drive device number, changing the, 324-325
- Drive-resident machine code, 328, 329
- DSAVE, 157
- D-style subminiature connector, 398
- Dual-processor scheme, 354

- E**
- EDIRQ, 53, 54, 91
- Editor control characters, 255, 261-262, 358, 359
- Editor escape sequences, 256-260
 - 8080, 354
 - 8502, 4, 8, 20, 49, 52, 53, 354, 357
 - 8522, 5
 - 8563, 9, 42, 54, 253, 265-273
 - 8564, 5, 253
 - 8722, 20, 25
 - ELSE, 167-168
 - END, 148, 158, 168
 - End-of-tape mark, 180
 - Englisch, Lothar (author), 14
 - Entry points, 119
 - for major routines, 236-251
 - ENVELOPE, 158
 - ERR\$, 208
 - Error messages, 170
 - Error statuses, 170
 - Error-laden track, 321
 - ESC key, 5, 254
 - Escape commands, 253
 - Escape sequences, 16
 - sent to printer, 254
 - Escape tokens, 134
 - EXECUTE, 327
 - EXIT, 158
 - EXP, 208
 - Expansion cartridge, 200
 - Expansion port, 4
 - Expansion port pinout, 372
 - Exponent byte, 126, 127, 129
 - Exponentiation operations, 275
 - Expression evaluation, 120, 235
 - Extended keyboard handler code, 7
 - Extended keyboard matrix arrangement, 6
 - Extended keys, 8
 - Extended VAL function, 234-235
 - External ROM, 23
 - External ROM cartridge variables, 94-97

 - F**
 - FAC, 226, 227
 - FAC1, 226
 - FAC2, 226
 - FACSGN, 227
 - FAST, 20, 159
 - Fast square root routine, 232
 - Fast-load programs, 314, 315
 - Fast-transfer protocol, 341
 - FASTLOAD, 348
 - Fetch, 122

FETCH, 159, 201, 302

1541 disk drive, 11

1571 disk drive, 1, 11

FILE EXISTS, 196

File-numbering scheme, 26

FILER, 160

Files, 139

Filler bytes, 265, 276

Filler character, 186

FIRE button, 216

First data item, 183

Five-byte floating-point format, 129

Fixed stack, 38

Flag byte, 47

Floating boundary, 120

Floating-point accumulator structure, 226

Floating-point arrays, 126

Floating-point mantissa bytes, structure, 127

Floating-point number, 121, 126-130, 211

converting to decimal, 128

Floating-point routines, 16

Floating-point values, 125

Floating-point variable, 122, 151

FOR, 160-161

Foreground color register, 273

Foreground text display colors, 276

FORMAT, 359

FORMAT DISK, 349

40/80 DISPLAY key, 34

Four-byte mantissa, 126

FRE, 209

Front end, 290

Front-end code, 52

FSDIR, 35, 301

Function keys, 170

software buffer, 58

Function ROM, 43

Function tokens, 137

G

Game port pinout, 371

Gap, 321

Garbage collection, 23, 119, 209

Garbage string, 123

GCR, 349

Generalized I/O, 287

Gertis, Klaus (author), 14

GET, 162, 168

GET#, 163, 168

GETCFG, 26, 307

GETKEY, 162

Ghosts, 23

GO64, 3, 13, 48, 163

GOSUB, 133, 164, 192

GOTO, 133, 165, 167

GO_TO, 165

GRAPHIC, 165

Graphic characters, 150

Graphic screen, 156

Graphics display, 16

Greater-than sign, 187

Group code recorded (GCR), 358

GSHAPE, 198-200

H

Head select function, 323

Header, 139, 317

checksum error, 326

HEADER, 165-166

Header/data scheme, 124

HELP key, 5, 166

HEX\$, 209

Hexadecimal number, 208, 209

Hexadecimal-decimal integer conversion, 377-383

Hidden information, 318

High (logical one), 6

High-byte/low-byte format, 125

High-pass filter, 160

High-resolution bit-mapped display, 58

High-resolution bit-mapped graphics, 273-276

Hole, 23, 29

Housekeeping functions, 274

I

ICR, 346

ID bytes, 34

ID characters, 166

Identification number, 139, 165

IF, 167-168

ILLEGAL QUANTITY ERROR, 214

Illegal track, 13

Immediate mode statements, 165

Immers, Richard (author), 14

Implied LET statement, 172

In context, 26

INDCMP, 309

Independent RAM block, 263-265

Index, 179, 307

INDFET, 59, 308

INDIN1, 86

INDIN2, 86, 87

Indirect fetch via INDEX1, 85

Indirect fetch via INDEX2, 86

Indirect fetch via TXTPR, 86

Indirect indexing, 308

Indirect-load routines, 86

INDSTA, 26, 59, 308

INDTXT, 86

Initial RAM bank selection, 4-5

INIT_STATUS, 46, 48, 49, 297, 298

INPUT, 149, 168-169

Input buffer, 78, 170

INPUT#, 168, 169-170

INQUIRE STATUS, 346

Insert mode, 256

INSTR, 210

INT, 210, 226

Integer variable, 122, 123

Interlace mode select register, 270

Internal ROM, 23

Interrupt-driven graphics functions, 285

I/O

block, 24

chips, 8

devices, 4

IOINIT, 46, 49, 297

IRQ, 6, 25, 43, 52-56, 91, 278

J

Jiffy clock, 54
 JMPFAR, 289, 307
 Job queue, 325-338
 JOY, 211
 JSRFAR, 59, 289, 307
 Jump table, 120, 287

K

Kaypro, 351
 Kernal
 banks, 30
 cross-bank routines, 59
 dictionary, 290-296
 indirect cross-bank operations, 112-114
 indirect vectors, 106
 jump table, 114-117, 288
 operating system, 16, 20
 ROM, 1, 4, 8, 15, 30, 106-112, 269, 288
 routines, 296-311
 KEY, 170-171
 Keyboard buffer, 79, 162
 Keyboard control register, 5
 Keyboard decode matrix pointer, 46
 KEYSET, 306

L

LCRA, 31
 LCRD, 31
 Least significant mantissa byte, 129
 Leemon, Sheldon (author), 14
 LEFTS, 212
 LEN, 212, 234
 LET, 171-172
 Light pen, 214, 265, 271
 Line execution, 120
 LINE FEED key, 5
 Line storage, 16
 Line-limiting parameters, 172
 Link, 133
 LIST, 172
 LKUPLA, 304
 LKUPSA, 304
 LOAD, 12, 174-175, 306, 314, 346
 Load address bytes, 361
 Load configuration registers, 31-33
 LOCATE, 175-176
 LOG, 213, 226
 Logical banks, 26, 27
 Logical file number, 163, 169, 183, 188
 Logical Kernal bank, 31
 Logical screen lines, 255
 LOOP, 158
 Low (logical zero), 6
 Low memory, 15, 78
 Low pass filter, 160
 Low-byte/high-byte format, 120

M

Machine language monitor, 176
 Machine-language demonstration program, 11

Main IRQ, 54, 56
 Mantissa, 126, 129
 Matching hood, 398
 Math operations, 127
 Media error, 166
 MEMBOT, 298
 Memory collides, 124
 Memory management unit (MMU), 5, 20, 25, 28-48
 Memory map, of C-128 system, 22
 Memory-execute (M-E) command, 328
 Memory-read command, 328
 MEMTOP, 298
 MIDS, 213
 Middle IRQ, 54, 91
 Mid-space, 29
 Minus sign, 185, 186, 200
 ML monitor, 361
 Mode configuration register, 34-35
 Modem port usage, 362
 Modes, switching between disk drive, 322
 Modified frequency modulated (MFM), 347, 349, 358
 MONITOR, 176
 Monitor dump of tokenized BASIC program, 133
 Monitor memory dump, 122
 Monitor RAM area, 93-94
 Monitor ROM, 15, 97-101
 Morrow, 351
 Most significant bit (MSB), 128, 129
 MOVIF, 227
 MOV2F, 227
 MOVSPR, 176
 Multicolor sprite colors, 197

N

National Television Standards Committee (NTSC), 297, 397
 Native mode, 1, 3, 15, 19, 34, 287, 298, 301
 Negative-edge triggered, 51
 Nested loops, 161
 Neufeld, Gerald C. (author), 14
 NEW, 177
 New line, 183
 New page one, 39
 New page zero, 39
 Newton's method, 231
 NEXT, 160-161
 NMI handler routine, 50
 NO SCROLL key, 5
 Non-array variable declaration, 121
 Non-maskable interrupt (NMI), 25, 41, 48-52
 Normalization operation, 127
 Normalized floating-point number, 127
 NTSC composite monochrome monitor, 20
 Null devices, 362
 Null string, 206
 Numb parameter, 329
 Numeric constants, 150
 Numeric keypad, 5
 Numeric variable, 16, 162, 169
 Nybble wide, 35, 40

O

OFF, 134
 ON-GOSUB, 177-178
 ON-GOTO, 178-179
 OPEN, 12, 179-181, 346, 348
 Operating system routines, 44
 Operating-system vectors, 15
 Osborne, 351
 Outboard cursor keys, 5
 Output ports, 35
 Output strings, 170

P

POH, 38
 POL, 38
 PIH, 38
 PIL, 38
 Page 1 memory usage, 59-60, 77-78
 Page 2 storage, 79-80
 Page 3 storage, 80-84
 indirect vectors, 80
 kernel indirect vectors, 81
 RAM resident indirect load, 84-87
 screen editor indirect vectors, 82-84
 Page relocation program, 39
 Page relocation techniques, 40
 Page-one relocation registers, 39
 Page-pointer registers, 38-40
 PAINT, 181
 PALCNT, 297
 PALNTS, 297
 Parameter buffer, 348
 Patch, 51
 PCQDEF, 306
 PCRA, 31, 32
 PCRB, 32
 PCRD, 31
 PEEK, 140, 159, 214
 PEN, 214
 Pencil soldering iron, 398
 PetASCII, 8, 41, 262, 289
 PFKEY, 306
 Phase-Alternating Line (PAL), 297
 PHOENIX, 4, 46, 48, 304
 Phono plug, 398
 Phony JSR, 53
 Physical address table, 304
 Physical screen lines, 255
 PIP, 363
 Pixel cursor, 175, 217
 PKYBUF, 306
 PLAY, 158, 181
 PLOT, 300
 Plotter demonstration, 274
 Plus sign, 185, 186, 200, 290, 296
 POINTER, 215
 POKE, 140, 159, 173, 181-182, 362, 387, 394-396
 POLL, 43, 46, 304
 Polled cartridges, 34

Polygon, 143
 POS, 215
 Post incrementing, 275
 POT, 215
 Pound sign, 176, 186
 Power-up activities, 15
 Preconfiguration registers, 31-33, 307
 PREND, 54
 PRIMM, 310 comparison (CMP), 309
 PRINT, 173, 182-183, 221
 formats, 182, 183
 PRINT#, 183-184
 output to cassette files, 184
 output to diskette files, 184
 output to line printer, 184
 Print Immediate, 310
 PRINT USING, 185
 PRINT#USING, 185
 Printer interface device, 12
 Printing
 a directory, 155
 a program listing, 173
 Program files, 361
 Program storage, 21, 133
 Programmable function keys, 47
 Public domain utilities, 316
 PUDEF, 187

Q

QUERY DISK FORMAT, 346
 Question mark, 168, 169
 QUIT, 134
 Quote mode, 256
 QWERTY, 6

R

RAM, 4
 bank, 37
 buffer region, 329
 configuration register, 36-38
 disk, 358
 RAMTAS, 47, 298
 Random number sequences, 218
 RCLR, 216
 RDBYT, 346, 348
 RDOT, 217
 READ, 187
 Read access, 156
 RECORD, 188-189
 Record number, 189
 Red-green-blue (RGB) color monitor, 20
 Register-write routine, 277
 Relative coordinates, 199
 Relative files, 156
 REM, 168, 189
 RENAME, 190
 RENUMBER, 190
 Reset, 25
 RESET, 46
 Resident code, 45

- RESTOR, 8, 47, 289
- RESTORE, 190, 193
- RESTORE key, 5, 49
- RESUME, 191, 202
- RESUME NEXT, 191
- Retries, controlling number of, 321-322
- Retry counter, 322
- RETURN, 168, 191-192
- RETURN key, 171, 192, 359, 387
- Reverse flag, 143
- Reverse screen register, 272
- Reverse video bit, 264
- Reverse-video mode, 256
- RGB 1 connection, 375
- RGR, 217
- RIGHTS, 217
- RND, 218
- ROM function, testing for, 322
- ROM header structure, 45
- ROM signature analysis, 322
- RREG, 134, 192, 201, 269
- RS-232 serial I/O data handling routines, 49
- RSPCOLOR, 218
- RSPPOS, 219
- RSPRITE, 219
- RUN, 192
- RUN/STOP, 387
- RUN-STOP key, 49, 54, 148, 201, 205
- RUN-STOP/RESTORE key sequence, 8, 12, 34, 47, 51, 314
- RWINDOW, 219

- S
- SAVE, 12, 193, 306, 314, 346, 361
- SCALE, 194
- Scaled value, 129
- Scientific notation, 182
- SCNCLR, 195
- SCRATCH, 195, 196
- Screen codes, 392-394
- Screen editor, 254, 289
 - code, 20
 - escape sequences, 134, 254-262
 - jump table, 101-106
- SCRORG, 300
- Second screen, 265
- Secondary addresses (channels), 304
- Sector editor, 316
- Sector interleave value, 321
- SECURE, 43
- Seek operations, 322
- SEI, 346
- Semicolon, 183
- Sequential file, 156, 361
- Serial bus I/O, 287
- Serial bus operations, 9
- Serial bus printer, 12
- Serial I/O port pinout, 373
- SETBNK, 299, 306
- SETLFS, 306
- SETNAM, 299, 306
- SGN, 220
- Shadow register, 55, 56, 91-117, 278, 285
- Shared RAM, 43, 85
- Shifted-space characters, 317
- SHIFT/RUN-STOP key combination, 306
- Sign character, 182
- Significant characters, 122
- SIN, 220, 233
- 6502, 20, 42, 120
- 6510, 20
- 6545E, 262
- SLEEP, 196
- SLOW, 9, 196
- Software library, 58
- Solder bridges, 398
- SOUND, 197, 204
- Sound generation, 16
- Sound interface device (SID), 23, 29, 46, 159, 297
- Sound parameters, 158
- Space, 173
- SPC, 220
- SPIN_SPOUT, 301
- Splat file, 52
- Split-screen graphics, 35
- Split-screen mode, 54
- SPRCOLOR, 197
- SPRDEF, 198
- SPRITE, 198
- Sprite collisions, 207
- Sprite movement, 176
- Sprite multicolor registers, 218
- Sprite-object collisions, 146
- Sprite-sprite collisions, 146
- Sprites, 207
- SPRSV, 198
- SQR, 221, 226
- SSHAPE, 198-200
- ST, 221
- Stack, 38
- Stack RAM area, 38, 40
- STASH, 200, 201, 302
- Statement tokenization, 120
- Statement tokens, 137-138
- STEP, 160-161
- STOP, 148, 168, 200
- Stop in drive mechanism, 322
- STR\$, 222
- TREND, 120
- String
 - characters, 213
 - constants, 150
 - descriptor, 121
 - storage, 124
 - variable storage, 16, 122, 123
- SWAP, 201, 302
- SWAPPER, 305
- Syntax error, 130
- SYS, 140, 192, 201, 269

SYS 49152, 10, 11
 SYS 51968, 7, 8
 System bank, 48, 289
 System guide, 134
 SYSTEM vector, 41-48
 diversion, 51
 patching, 42
 SYSTEM-VECTOR, 298
 System version register (VR), 41
 Szczepanowski, Norbert (author), 14

T

TAB, 221
 TAB key, 5
 TABS, 223
 Tabbing, 183
 TAM share, 36
 TAN, 223
 Taylor series, 233
 TEMPF1, 227
 TEMPF2, 227
 Terminator byte, 311
 Test pattern, 9
 Test/sprite graphics chip, 36
 Text tokenization, 16
 THEN, 167-168
 Three-byte descriptors, 125
 TI, 224
 TIS, 224
 TKSA, 301
 Token byte, 151
 Tokenized form, 134
 Trailing blank character, 183
 Transcendental functions, 233
 Transferring files, 360-361
 Transient program area (TPA), 358
 Translated memory map, 35
 TRAP, 191, 201
 TRI, 49
 TROFF, 202
 TRON, 202
 True address, 39
 True pages, 38
 True zero page, 38
 Turbo C-64, 9
 Two-byte integer, 121
 Two-byte vectors, 289
 Two-tiered interrupt scheme, 54

U

U0 series of disk commands, 319
 Ultrahigh-resolution graphics, 42
 Unconditional branch, 167
 Underline attribute, 264
 Underline scan line count register, 273
 UNIMPLEMENTED COMMAND ERROR,
 134
 UNTALK, 291, 301
 Update location register, 272
 Upper nybble, 41
 Upward scroll, 258

User port pinout, 374
 User-created routine, 121
 User-defined character sets, 276-278
 User-modifiable indirect vectors, 58
 USR, 140, 233
 USR(x), 224
 Utilities, 300
 Utility routines, 304

V

VAL, 224, 236
 Variable search process, 122
 Variable storage, 120-126
 VCDADR, 265, 267
 VDCDAT, 267
 VDCOUT, 269
 Vectored Kernal routine, 290
 Vectors, 289-290
 VERIFY, 202, 306
 Verify-after-write function, 362
 Vertical displayed register, 270
 Vertical sync position register, 270
 VIC-II chip, 23, 37, 253, 263, 278-286
 Video display screen, 262-263
 Virtual drive, 324
 VOL, 204

W

WAIT, 168, 204
 Wait state, 302
 Warm-start code, 49
 Warm-start entry, 46
 West, Raeto Collin (author), 14
 WIDTH, 205
 Window, 219
 WINDOW, 205
 Windowing techniques, 16
 Wraparound IRQ, 56
 Write, 32
 Write access, 156
 Write-protect drive, 324

X

X parameter, 143
 XOR, 225

Y

Y parameter, 143

Z

Z-80, 4, 20, 35, 42, 43, 253, 263, 354
 memory map for CP/M Plus, 356
 Zero value of exponent byte, 126
 Zero-page
 addressing mode, 59
 memory, 38, 59-60, 329
 pointer, 308, 309
 RAM, 57, 226, 329-338
 stack usage, 60-61
 storage, 61-76
 utility, 234

IF YOU ENJOYED THIS BOOK...

help us stay in touch with your needs and interests by filling out and returning the survey card below. Your opinions are important, and will help us to continue to publish the kinds of books you need, when you need them.



What brand of computer(s) do you own or use? _____

Where do you use your computer the most? At work At school At home

What topics would you like to see covered in future books by Osborne/McGraw-Hill?

How many other computer books do you own? _____

Why did you choose this book?

- Best coverage of the subject.
- Recognized the author from previous work.
- Liked the price.
- Other

Where did you find this book?

- Bookstore
- Computer/software store
- Department store
- Advertisement
- Catalog

Where did you hear about this book?

- Book review.
- Osborne catalog.
- Advertisement in: _____
- Found by browsing in store.
- Found/recommended in library
- Other

- Required textbook
- Library
- Gift
- Other

Where should we send your **FREE** catalog?

NAME _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____



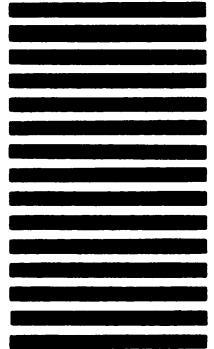
No Postage
Necessary
If Mailed
in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 3111 Berkeley, CA

Postage will be paid by addressee

Osborne McGraw-Hill

2600 Tenth Street
Berkeley, California 94710



Commodore 128™ PROGRAMMING SECRETS

Commodore 128™ Programming Secrets takes you from programming fundamentals to sophisticated techniques that enable you to create your own powerful software for the C-128.™

All C-128 operating modes are described in detail so you can learn to program effectively in the Commodore 64™ mode, the Commodore 128 "native" mode, and the CP/M® mode. Wiese explains concepts through the use of numerous, hands-on programs that are ready for immediate use.

Commodore 128™ Programming Secrets shows you how to

- Use Kernal ROM and BASIC 7.0 ROM to run faster, more powerful programs
- Gain greater control over screen displays with video RAM
- Apply bank switching tricks to store and access additional programs in extra memory banks
- Understand CP/M BIOS mapping so you can take full advantage of the C-128's CP/M mode
- Create advanced text, graphics, and sound applications
- Use memory management techniques for efficient programming
- Control C-128 disk and input/output operations for a greater variety of programming applications.

Commodore 128™ Programming Secrets also includes a *complete* BASIC 7.0 dictionary and C-128 memory maps.

Wiese's insightful programming techniques reveal the C-128's mysteries, putting you in full charge of this computer's enormous capabilities.

William M. Wiese, Jr. is a programmer and writer whose articles have appeared in *BYTE*, *COMPUTE!*, and other computer periodicals.

- Apple is a registered trademark of Apple Computer, Inc.
- Commodore 64 and Commodore 128 are trademarks of Commodore Business Machines, Inc.
- CP/M is a registered trademark of Digital Research, Inc.
- IBM is a registered trademark of IBM Corp.



¥15.95

ISBN 0-07-881250-X