# COMMODORE 128

## PROGRAMMER'S REFERENCE GUIDE

**Bantam Computer Books**
**Ask your bookseller for the books you have missed**

THE AMIGADOS MANUAL
by Commodore-Amiga, Inc.

THE APPLE //c BOOK
by Bill O'Brien

THE ART OF DESKTOP PUBLISHING
By Tony Bove, Cheryl Rhodes, and Wes Thomas

ARTIFICIAL INTELLIGENCE ENTERS THE MARKETPLACE
by Larry R. Harris and Dwight B. Davis

THE BIG TIP BOOK FOR THE APPLE II SERIES
by Bert Kersey and Bill Sanders

THE COMMODORE 64 SURVIVAL MANUAL
by Winn L. Rosch

COMMODORE 128 PROGRAMMER'S REFERENCE GUIDE
by Commodore Business Machines, Inc.

THE COMPUTER AND THE BRAIN
by Scott Ladd/
The Red Feather Press

EXPLORING ARTIFICIAL INTELLIGENCE ON YOUR APPLE II
by Tim Hartnell

EXPLORING ARTIFICIAL INTELLIGENCE ON YOUR COMMODORE 64
by Tim Hartnell

EXPLORING ARTIFICIAL INTELLIGENCE ON YOUR IBM PC
by Tim Hartnell

EXPLORING THE UNIX ENVIRONMENT
by The Waite Group/Irene Pasternack

FRAMEWORK FROM THE GROUND UP
by The Waite Group/Cynthia Spoor and Robert Warren

HOW TO GET THE MOST OUT OF COMPUSERVE, 2d ed.
by Charles Bowen and David Peyton

HOW TO GET THE MOST OUT OF THE SOURCE
by Charles Bowen and David Peyton

THE MACINTOSH
by Bill O'Brien

MACINTOSH C PRIMER PLUS
by The Waite Group/Stephen W. Prata

THE NEW *jr*: A GUIDE TO IBM'S PC*jr*
by Winn L. Rosch

ORCHESTRATING SYMPHONY
by The Waite Group/Dan Shafer with Mary Johnson

PC-DOS/MS-DOS
*User's Guide to the Most Popular Operating System for Personal Computers*
by Alan M. Boyd

POWER PAINTING: COMPUTER GRAPHICS ON THE MACINTOSH
by Verne Bauman and Ronald Kidd/
illustrated by Gasper Vaccaro

SMARTER TELECOMMUNICATIONS
*Hands-On Guide to On-Line Computer Services*
by Charles Bowen and Stewart Schneider

SWING WITH JAZZ: LOTUS JAZZ ON THE MACINTOSH
by Datatech Publications Corp./S. Michael McCarty

UNDERSTANDING EXPERT SYSTEMS
by The Waite Group/Mike Van Horn

USER'S GUIDE TO THE AT&T PC 6300 PERSONAL COMPUTER
by David B. Peatroy, Ricardo A. Anzaldua, H. A. Wohlwend, and Datatech Publications Corp.

# COMMODORE 128

## PROGRAMMER'S REFERENCE GUIDE

COMMODORE BUSINESS MACHINES, INC.

BANTAM BOOKS

COMMODORE 128 PROGRAMMER'S REFERENCE GUIDE
*A Bantam Book / February 1986*

*Commodore 64 and Commodore 128 are registered trademarks of Commodore Electronics, Ltd.*

*CP/M and CP/M Plus Version 3.0 are registered trademarks of Digital Research Inc.*

*Perfect is a registered trademark of Perfect Software.*

*TouchTone is a registered trademark of AT&T.*

*WordStar is a registered trademark of MicroPro International Corporation.*

*Grateful acknowledgment is made for permission to reprint two bars of Invention 13 (Inventio 13) by Johann Sebastian Bach. Sheet music copyright © C. F. Peters, Corp., New York.*

*Book design by Ann Gold.*

*Cover design by Jo Ellen Temple.*

*All rights reserved.*
*Copyright © 1986 by Commodore Capital, Inc.*
*This book may not be reproduced in whole or in part, by mimeograph or any other means, without permission.*
*For information address: Bantam Books, Inc.*

ISBN 0-553-34292-4

*Published simultaneously in the United States and Canada*

---

*Bantam Books are published by Bantam Books, Inc. Its trademark, consisting of the words "Bantam Books" and the portrayal of a rooster, is Registered in U.S. Patent and Trademark Office and in other countries. Marca Registrada. Bantam Books, Inc., 666 Fifth Avenue, New York, New York 10103.*

---

PRINTED IN THE UNITED STATES OF AMERICA

HL 0 9 8 7 6 5 4 3

# CONTENTS


Chapter 1
Introduction 1

Chapter 2
BASIC Building Blocks and BASIC 7.0 Encyclopedia 11

Chapter 3
One Step Beyond Simple BASIC 91

Chapter 4
Commodore 128 Graphics Programming 109

Chapter 5
Machine Language on the Commodore 128 123

Chapter 6
How to Enter Machine Language Programs Into the Commodore 128 181

Chapter 7
Mixing Machine Language and BASIC 197

Chapter 8
The Power Behind Commodore 128 Graphics 207

Chapter 9
Sprites 265

Chapter 10
Programming the 80-Column (8563) Chip 291

Chapter 11
Sound and Music on the Commodore 128 335

Chapter 12

Chapter 13
The Commodore 128 Operating System 401

Chapter 14
CP/M 3.0 on the Commodore 128 477

Chapter 15
The Commodore 128 and Commodore 64 Memory Maps 501

Chapter 16
C128 Hardware Specifications 555

Appendixes 643

Glossary 731

Index 739

# ACKNOWLEDGMENTS

Written by Larry Greenley
and
Fred Bowen
Bil Herd
Dave Haynie
Terry Ryan
Von Ertwine
Kim Eckert
Mario Eisenbacher
Norman McVey

The authors are deeply indebted to the many people who have contributed to the preparation of this book. Special thanks go to Jim Gracely of Commodore Publications, who reviewed the entire manuscript for technical accuracy and provided important corrections, clarifications, and user-oriented suggestions, and to Steve Beats and Dave Middleton of Commodore Software Engineering for their programming assistance and expertise.

We also want to recognize the contributions of Frank Palaia of Commodore Hardware Design, who provided expertise in the operation of the Z80 hardware, and of Dave DiOrio of Commodore Integrated Circuit Design, who provided insight into the design of the Memory Management Unit and the C128 VIC chip enhancements.

For their extensive technical reviews of the manuscript, we wish to thank Bob Albright, Pete Bowman, Steve Lam and Tony Porrazza of Commodore Engineering. We also thank Dan Baker, Dave Street and Carolyn Scheppner of Commodore Software Technical Support for providing an always available source of technical assistance. In addition, we want to acknowledge the valuable contributions of members of Commodore Software Quality Assurance, especially Mike Colligon, Karen Mackenzie, Pat McAllister, Greg Rapp, Dave Resavy, and Stacy English.

We also thank Carol Sullivan and Donald Bein for carefully proofreading various sections of the text, Michelle Dreisbach for typing the manuscript, Marion Dooley for preparing the art, Jo-Ellen Temple for the cover design, and Nancy Zwack for overall coordination assistance.

Finally, we would like to acknowledge the unflagging support and guidance provided by senior Commodore executives Paul Goheen, Harry McCabe and Bob Kenney.

# 1

# INTRODUCTION

The Commodore 128 Personal Computer is a versatile, multimode computer. The Commodore 128 is the successor to the commercially successful Commodore 64 computer. The principal features of the Commodore 128 are:

- 128K bytes of RAM, optionally expandable to 256K or 640K
- 80-column horizontal screen display
- Hardware and software compatibility with Commodore 64
- CP/M 3.0 operation
- Enhanced BASIC language

As this Guide shows, the Commodore 128 has many other new or expanded capabilities and features. Those listed above, however, are the most significant when assessing the Commodore 128's capabilities against those of the Commodore 64 and other microcomputers.

The Commodore 128 is actually three computers in one, with the following three primary operating modes:

- C128 Mode
- C64 Mode
- CP/M Mode

Two of these primary modes (C128 and CP/M) can operate using either a 40- or 80-column screen display. Following is a summary of the major features of each of the three primary operating modes.

## C128 MODE

In C128 Mode, the Commodore 128 Personal Computer provides the capabilities and memory needed to run sophisticated applications, such as word processing, spreadsheets, and database programs.

C128 Mode features include:

- 8502 processor running at 1.02 or 2.04 MHz
- New, enhanced C128 Kernal
- Built-in machine language monitor
- Commodore BASIC 7.0 language, with over 140 commands and functions
- Special new BASIC 7.0 commands that provide better, quicker and easier ways to create complex graphics, animation, sound and music programs
- 40-column text and bit map screen output using VIC II chip
- 80-column text screen output using 8563 chip

**NOTE:** The 40- and 80-column screen displays can be used either singly or simultaneously with two monitors.

- Sound (three voices) using SID chip
- A 92-key keyboard that includes a full numeric keypad and ESCAPE, TAB, ALT, CAPS LOCK, HELP, LINE FEED, 40/80 DISPLAY, and NO SCROLL keys
- Access to the full capabilities of the new peripheral devices from Commodore (1571 fast disk drive, 1902 dual 40/80-column RGBI monitor, etc.)
- Access to all standard Commodore serial peripherals
- RAM expansion to 256 or 640K with optional RAM expansion modules

# C64 MODE

In C64 Mode, the Commodore 128 retains all the capabilities of the Commodore 64, thus allowing you to use the wide range of available Commodore 64 software.

C64 Mode features include:

- 8502 processor running at 1.02 MHz
- Standard C64 Kernal
- BASIC 2.0 language
- 64K of RAM
- 40-column output using VIC II chip
- Sound (three voices) using SID chip
- Standard Commodore 64 keyboard layout except for function keys
- All standard Commodore 64 keyboard functions
- Access to all Commodore 64 graphics, color and sound capabilities, used as on a Commodore 64
- Compatibility with standard Commodore 64 peripherals, including user port and serial devices, Datassette, joysticks, composite video monitors, and RF (TV) output devices

**NOTE:** The 1571 disk drive will function in C64 Mode, but only at standard 1541 speed. C64 compatibility requirements make it impossible for the 1571 to operate in C64 Mode at fast speed.

# CP/M MODE

In CP/M Mode, an onboard Z80 microprocessor gives you access to the capabilities of Digital Research's CP/M Version 3.0, plus a number of new capabilities added by Commodore.

CP/M Mode features include:

- Integral Z80 processor running at 2.04 MHz
- Disk-based CP/M 3.0 System
- 128K bytes of RAM (in 64K banks)
- 40-column screen output using VIC II chip
- 80-column screen output using 8563 chip
- Access to the full keyboard, including the numeric keypad and special keys
- Access to the new fast serial disk drive (1571) and the standard serial peripherals
- Ability to redefine almost any key
- Ability to emulate several terminals (Lear-Siegler ADM31, ADM3A)
- Support for various MFM disk formats (IBM, Kaypro, Epson, Osborne)
- RAM expansion to 256 or 640K RAM with optional RAM expansion modules

The incorporation of CP/M 3.0 (also called CP/M Plus) into the Commodore 128 makes thousands of popular commercial and public domain software programs available to the user.

# HARDWARE COMPONENTS

The Commodore 128 Personal Computer incorporates the following major hardware components:

## PROCESSORS

***8502:*** Main processor in C128, C64 Modes; I/O support for CP/M; 6502 software-compatible; runs at 1.02 or 2.04 MHz

***Z80:*** CP/M Mode only; runs at 2.04 MHz

## MEMORY

***ROM:*** 64K standard (C64 Kernal plus BASIC; C128 Kernal plus BASIC, character ROMs and CP/M BIOS); one 32K slot available for software

***RAM:*** 128K in two 64K banks; 16K display RAM for 8563 video chip; 2K × 4 Color RAM

## VIDEO

***8564:*** 40-column video (separate versions for NTSC and PAL TV standards)

***8563:*** 80-column video

## SOUND

***6581:*** SID Chip

## INPUT/OUTPUT

***6526:*** Joystick ports/keyboard scan/cassette
***6526:*** User and serial ports

## MEMORY MANAGEMENT

***8921:*** PLA (C64 plus C128 mapping modes)
***8922:*** MMU (Custom gate array)

For details on these and other hardware components see Chapter 16, Commodore 128 Hardware Specifications.

# COMPATIBILITY WITH COMMODORE 64

The Commodore 128 system is designed as an upgrade to the Commodore 64. Accordingly, one of the major features of the Commodore 128 design is hardware and software compatibility with the Commodore 64 when operating in C64 Mode. This means that in C64 Mode the Commodore 128 is capable of running Commodore 64 application software. Also, the Commodore 128 in C64 Mode supports Commodore 64 peripherals except the CP/M 2.2 cartridge. (NOTE: The Commodore 128's built-in CP/M 3.0 capability supersedes that provided by the external cartridge. This cartridge should not be used with the Commodore 128 in any mode.)

The C128 Mode is designed as a compatible superset to the C64. Specifically, all Kernal functions provided by the Commodore 64 are provided in the C128 Kernal. These functions are also provided at the same locations in the jump table of the C128 Kernal to provide compatibility with existing programs. Zero page and other system variables are maintained at the same addresses they occupy in C64 Mode. This simplifies interfacing for many programs.

Providing Commodore 64 compatibility means that the new features of the Commodore 128 cannot be accessed in C64 Mode. For example, compatibility and memory constraints preclude modifying the C64 Mode Kernal to support the 1571 fast serial disk drive. As noted previously, C64 Mode sees this drive as a standard serial disk drive. For the same reason, C64 Mode does not have an 80-column screen editor, and C64 Mode BASIC 2.0 cannot use the second 64K bank of memory.

# SWITCHING FROM MODE TO MODE

As mentioned before, in the C128 and CP/M Modes the Commodore 128 can provide both 40-column and 80-column screen displays. This means that the Commodore 128 actually has five operating modes, as follows:

- C128 Mode with 80-column display
- C128 Mode with 40-column display
- C64 Mode (40-column display only)
- CP/M Mode with 80-column display
- CP/M Mode with 40-column display

Figure 1-1 summarizes the methods used to switch from mode to mode.

| TO \ FROM | OFF | C128 40 COL | C128 80 COL | C64 | CP/M 40 COL | CP/M 80 COL |
|---|---|---|---|---|---|---|
| **C128 40 COL** | 1. Check that 40/80 key is UP.<br>2. Make sure that:<br>a)No CP/M system disk is in drive<br>b)No C64 cartridge is in expansion port<br>3. Turn computer ON. | | 1. Press ESC key; release.<br>2. Press X key.<br>OR<br>1. Check that 40/80 key is UP.<br>2. Press RESET button. | 1. Check that 40/80 key is UP.<br>2. Turn computer OFF, then ON.<br>3. Remove cartridge if present | 1. Check that 40/80 key is UP.<br>2. Turn computer OFF, then ON. | 1. Check that 40/80 key is UP.<br>2. Turn computer OFF, then ON. |
| **C128 80 COL** | 1. Press 40/80 key DOWN.<br>2. Turn computer ON. | 1. Press ESC key; release.<br>2. Press X key.<br>OR<br>1. Press 40/80 key DOWN.<br>2. Press RESET button. | | 1. Press 40/80 key DOWN.<br>2. Turn computer OFF, then ON.<br>3. Remove cartridge if present. | 1. Press 40/80 key DOWN.<br>2. Remove CP/M system disk from drive, if necessary.<br>3. Turn computer OFF, then ON. | 1. Check that 40/80 key is DOWN.<br>2. Remove CP/M system disk from drive, if necessary.<br>3. Turn computer OFF, then ON. |

**Figure 1-1. Commodore 128 Mode Switching Chart**

| TO | FROM: OFF | C128 40 COL | C128 80 COL | C64 | CP/M 40 COL | CP/M 80 COL |
|---|---|---|---|---|---|---|
| **C64** | 1. Hold " C= " key DOWN.<br>2. Turn computer ON.<br>OR<br>1. Insert C64 cartridge.<br>2. Turn computer ON. | 1. Type GO 64; press RETURN.<br>2. The computer responds: ARE YOU SURE? Type Y; press RETURN. | 1. Type GO 64; press RETURN.<br>2. The computer responds: ARE YOU SURE? Type Y; press RETURN. | | 1. Turn computer OFF.<br>2. Check that 40/80 key is UP.<br>3. Hold DOWN C= key while turning computer ON.<br>OR<br>1. Turn computer OFF.<br>2. Insert C64 cartridge.<br>3. Turn power ON. | 1. Turn computer OFF.<br>2. Check that 40/80 key is UP.<br>3. Hold DOWN C= key while turning computer ON.<br>OR<br>1. Turn computer OFF.<br>2. Insert C64 cartridge.<br>3. Turn power ON. |
| **CP/M 40 COL** | 1. Turn disk drive ON.<br>2. Insert CP/M system disk in drive.<br>3. Check that 40/80 key is UP.<br>4. Turn computer ON. | 1. Turn disk drive ON.<br>2. Insert CP/M system disk in drive.<br>3. Check that 40/80 key is UP.<br>4. Type: BOOT<br>5. Press RETURN. | 1. Turn disk drive ON.<br>2. Insert CP/M system disk in drive.<br>3. Check that 40/80 key is UP.<br>4. Type: BOOT<br>5. Press RETURN. | 1. Check that 40/80 key is UP.<br>2. Turn disk drive ON.<br>3. Insert CP/M system disk in drive.<br>4. Turn computer OFF. | | 1. Insert CP/M utilities disk in drive.<br>2. At screen prompt, A> type: DEVICE CONOUT: = 40 COL<br>3. Press RETURN. |
| **CP/M 80 COL** | 1. Turn disk drive ON.<br>2. Insert CP/M system disk in drive.<br>3. Press 40/80 key DOWN.<br>4. Turn computer ON. | 1. Turn disk drive ON.<br>2. Insert CP/M system disk in drive.<br>3. Press 40/80 key DOWN.<br>4. Type: BOOT<br>5. Press RETURN. | 1. Turn disk drive ON.<br>2. Insert CP/M system disk in drive.<br>3. Check that 40/80 key is DOWN.<br>4. Type: BOOT.<br>5. Press RETURN. | 1. Press 40/80 key DOWN.<br>2. Turn disk drive ON.<br>3. Insert CP/M system disk in drive.<br>4. Turn computer OFF. | I. Insert CP/M utilities disk in drive.<br>2. At screen prompt, A> type: DEVICE CONOUT = 80 COL<br>3. Press RETURN. | |

**Figure 1–1. Commodore 128 Mode Switching Chart (continued)**

**NOTE:** If you are using a Commodore 1902 dual monitor, remember to move the video switch on the monitor from COMPOSITE or SEPARATED to RGBI when switching from 40-column to 80-column display; reverse this step when switching from 80 to 40 columns. Also, when switching between modes remove any cartridges from the expansion port. You may also have to remove any disk (e.g., CP/M) from the disk drive.

## CP/M 3.0 SYSTEM RELEASES

When you send in your C128 warranty card, your name will be added to a list which makes you eligible for CP/M system release dates.

## HOW TO USE THIS GUIDE

This guide is designed to be a reference tool that you can consult whenever you need detailed technical information on the structure and operation of the Commodore 128 Personal Computer. Since many of the design features of the Commodore 128 can be viewed from various aspects, it may be necessary to consult several different chapters to find the information you want. Note that certain groups of chapters form logical sequences that cover in detail an extended topic like BASIC, graphics, or machine language.

The following chapter summaries should help you pinpoint what chapter or chapters are most likely to provide the answer to a specific question or problem.

**CHAPTER 2.** BASIC BUILDING BLOCKS AND BASIC 7.0 ENCYCLOPEDIA—Defines and describes the structural and operational components of the BASIC language, including constants, variables and arrays, and numeric and string expressions and operations.

**CHAPTER 3.** ONE STEP BEYOND SIMPLE BASIC—Provides routines (menu, keyboard buffer, loading, programming function keys) and techniques ("crunching" or saving memory; debugging and merging programs; relocating BASIC) that can be incorporated in your own programs. Provides modem-related information (how to generate TouchTone® frequencies, how to detect telephone ringing, etc.) plus technical specifications for Commodore Modem/1200 and Modem/300.

**CHAPTER 4.** COMMODORE 128 GRAPHICS PROGRAMMING—Describes the general BASIC 7.0 graphics commands (COLOR, GRAPHIC, DRAW, LOCATE, BOX, CIRCLE, PAINT) and gives annotated examples of use, including programs. Describes the structure and general function of the color modes and character and bit map graphics modes that are fundamental to Commodore 128 graphics.

**CHAPTER 5.** MACHINE LANGUAGE ON THE COMMODORE 128—Defines, with examples, machine language (ML) and associated topics, including the Kernal; the 8502 registers, binary and hexadecimal numbers, and addressing modes. Defines, with examples, types of ML instructions (op codes, etc.). Includes 8502 instruction and addressing table.

**CHAPTER 6.** HOW TO ENTER MACHINE LANGUAGE PROGRAMS INTO THE COMMODORE 128—Describes, with examples, how to enter ML programs by using the built-in Machine Language Monitor or by POKEing decimal op-code values with a BASIC program. Defines, with examples, the ML Monitor commands.

**CHAPTER 7.** MIXING MACHINE LANGUAGE AND BASIC—Describes, with examples, how to combine BASIC and ML instructions in the same program by using BASIC READ, DATA, POKE and SYS commands. Shows where to place ML programs in memory.

**CHAPTER 8.** THE POWER BEHIND COMMODORE 128 GRAPHICS—Describes the C128 Mode memory banking concept and tells how to manage banked memory. Defines the use of shadow registers. Describes how screen, color and character memory are handled in BASIC and machine language, for both character and bit map modes. Shows how to redefine the character set. Describes use of split-screen modes. Includes a tabular graphics programming summary.

**CHAPTER 9.** SPRITES—Describes programming of sprites or MOBs (movable object blocks). Defines and shows how to use the BASIC 7.0 sprite-related commands (SPRITE, SPRDEF, MOVSPR, SSHAPE, GSHAPE, SPRSAV). Provides annotated examples of use, including programs.

**CHAPTER 10.** PROGRAMMING THE 80-COLUMN (8563) CHIP—Defines the 8563 registers and describes, with machine language examples, how to program the 80-column screen in character and bit map modes.

**CHAPTER 11.** SOUND AND MUSIC ON THE COMMODORE 128—Defines the BASIC 7.0 sound and music commands (SOUND, ENVELOPE, VOL, TEMPO, PLAY, FILTER). Describes how to code a song in C128 Mode. Defines in detail the Sound Interface Device (SID) and how to program it in machine language.

**CHAPTER 12.** INPUT/OUTPUT GUIDE—Describes software control of peripheral devices connected through I/O ports, including disk drives, printers, other User Port and Serial Port devices, the Datassette, and Controller Port devices. Provides pin-out diagrams and pin descriptions for all ports.

**CHAPTER 13.** THE COMMODORE 128 OPERATING SYSTEM—Describes, with examples, the operating system (Kernal), which controls the functioning of the Commodore 128; includes the Kernal Jump Table, which lists the ROM entry points used to call the Kernal routines; defines each Kernal routine; defines the C128 Screen Editor. Describes the Memory Management Unit (MMU), defines the MMU registers, tells how to select and switch banks in BASIC and ML, and tells how to predefine memory configurations.

**CHAPTER 14.** CP/M 3.0 ON THE COMMODORE 128—Summarizes the Commodore version of CP/M 3.0. Defines the general system layout and the operating system components (CCP, BIOS and BDOS). Describes the Commodore enhancements to CP/M 3.0. (Additional details on CP/M 3.0 are given in Appendix K.)

**CHAPTER 15.** COMMODORE 128 AND COMMODORE 64 MEMORY MAPS—Provides detailed memory maps for C128 and C64 modes. (The Z80 memory map is shown in Appendix K.)

**CHAPTER 16.** HARDWARE SPECIFICATIONS—Includes technical specifications for Commodore 128 hardware components (8563, 8564, etc.).

**APPENDIXES A through L**—Provide additional technical information and/or a more convenient grouping of information supplied elsewhere in the Guide (e.g., pinout diagrams).

**GLOSSARY**—Provides standard definitions of technical terms.

# 2

# BASIC BUILDING BLOCKS AND BASIC 7.0 ENCYCLOPEDIA

The **BASIC** language is composed of commands, operators, constants, variables, arrays and strings. Commands are instructions that the computer follows to perform an operation. The other elements of BASIC perform a variety of functions, such as assigning values to a quantity, passing values to the computer, or directing the computer to perform a mathematical operation. This section describes the structure and functions of the elements of the BASIC language.

## COMMANDS AND STATEMENTS

By definition, commands and statements have the following distinctions. A command is a BASIC verb which is used in immediate mode. It is not preceded by a program line number and it executes immediately after the RETURN key is pressed. A statement is a BASIC verb which is contained within a program and is preceded by a line number. Program statements are executed with the RUN command followed by the RETURN key.

Most commands can be used within a program. In this case the command is preceded by a line number and is said to be used in program mode. Many commands also can be used outside a program in what is called direct mode. For example, **LOAD** is an often-used direct mode command, but you can also include LOAD in a program. **GET** and **INPUT** are commands that only can be used in a program; otherwise, an **ILLEGAL DIRECT ERROR** occurs. While **PRINT** is usually included within a program, you can also use PRINT in direct mode to output a message or numeric value to the screen, as in the following example:

PRINT "The Commodore 128" **RETURN**

Notice that the message is displayed on the screen as soon as you press the return key. The following two lines display the same message on the screen. The first line is a program mode statement; the second line is a direct mode command.

10 PRINT "The Commodore 128" **RETURN**

RUN **RETURN**

It is important to know about the concepts behind memory storage before examining the Commodore BASIC language in detail. Specifically, you need to understand constants, variables and arrays.

## NUMERIC MEMORY STORAGE: CONSTANTS, VARIABLES AND ARRAYS

There are three ways to store numeric information in Commodore BASIC. The first way is to use a *constant*. A constant is a form of memory storage in which the contents remain the same throughout the course of a program. The second type of memory storage unit is a *variable*. As the name indicates, a variable is a memory storage cell in

which the contents vary or change throughout the course of a program. The last way to store information is to use an *array*, a series of related memory locations consisting of variables.

Each of these three units of memory storage can have three different types of information or data assigned. The three data types are INTEGER, FLOATING-POINT or STRING. *Integer* data is numeric, whole number data—that is, numbers without decimal points. *Floating-point* is numeric data including fractional parts indicated with a decimal point. *String* data is a sequential series of alphanumeric letters, numbers and symbols referred to as character strings. The following paragraphs describe these three data types and the way each memory storage unit is assigned different data type values.

# CONSTANTS: INTEGER, FLOATING-POINT AND STRING

## INTEGER CONSTANTS

The value assigned to a constant remains unchanged or constant throughout a program. Integer constants can contain a positive or negative value ranging from –32768 through +32767. If the plus sign is omitted, the CI28 assumes that the integer is positive. Integer constants do not contain commas or decimal points between digits. Leading zeros are ignored. Integers are stored in memory as two-byte binary numbers, which means a constant requires 16 bits or two bytes of memory to store the integer as a base two number. The following are examples of integer constants:

1
1000
–32
0
–32767

## FLOATING-POINT CONSTANTS

Floating-point constants contain fractional parts that are indicated by a decimal point. They do not contain commas to separate digits. Floating-point constants may be positive or negative. If the plus sign is omitted, it is assumed that the number is positive. Again, leading zeros are unnecessary and ignored. Floating-point constants are represented in two ways depending on their value:

1. Simple Number Notation
2. Scientific Notation

In simple number notation, the floating-point number is calculated to ten digits of precision and stored using five bytes, but only nine digits are displayed on the screen or printer. If the floating-point number is greater than nine digits, it is rounded according to the tenth digit. If the tenth digit is greater than five, the ninth digit is rounded to the next higher digit. If the tenth digit is less than five, the ninth digit is rounded to the next lower digit. The rounding of floating-point numbers may become a factor when calculat-

ing values based upon floating-point numbers greater than nine digits. Your program should test floating-point results and take them into consideration when basing these values on future calculations.

As mentioned, floating-point numbers are displayed as nine digits. If the value of a floating-point constant is less than .01 or greater than 999999999, the number is displayed on the screen or printer in scientific notation. For example, the number 12345678901 is displayed as 1.23456789E+10. Otherwise, the simple number notation is displayed. A floating-point constant in scientific notation appears in three parts:

1. The mantissa is the leftmost number separated by a decimal point.
2. The letter E indicates that the number is displayed in exponential (scientific) notation.
3. The exponent specifies the power of ten to which the number is raised and the number of places the decimal point is moved in order to represent the number in simple number notation.

The mantissa and exponent can be positive or negative. The exponent can be within the range –39 to +38. If the exponent is negative, the decimal point moves to the left representing it as a simple number. If the exponent is positive, the decimal point moves to the right representing it in simple number notation.

The Commodore 128 limits the size of floating-point numbers. The highest number you can represent in scientific notation is 1.70141183E+38. If you try to represent a number larger than that, an **OVERFLOW ERROR** occurs. The smallest number you can represent in scientific notation is 2.93873588E–39. If you try to represent a number smaller than that, no error occurs but a zero is returned as the value. You should therefore test floating-point values in your programs if your calculations are based on very small numbers and the results depend on future calculations. Here are examples of floating-point constants in simple number notation and others in scientific notation:

| SIMPLE NUMBER | SCIENTIFIC |
|---|---|
| **9.99** | **22.33E+20** |
| **.0234** | **99999.234E–23** |
| **+10.01** | **–45.89E–11** |
| **–90.23** | **–3.14E+17** |

**NOTE:** The values in either column are not equivalent.

## STRING CONSTANTS

A string constant, as mentioned, is a sequential series of alphanumeric characters (numbers, letters and symbols). A string constant can be as long as a 160-character input

line, minus the line number and any other information appearing on that program line. By concatenating strings together, you may form a string as long as 255 characters. The string may contain numbers, letters, and even decimal points and commas. However, the string *cannot* contain the double quote ("") character, since this character delimits or marks the beginning or ending of the string. You can represent a double quote character within a string using **CHR$(34).** You can omit the closing double quote character of a string if it is the last statement in a line of a program.

A string can even be assigned a null value, meaning no characters are actually assigned to it. Assign a string a null value by omitting characters between the double quotes and follow the opening double quote directly with a closing double quote. Here are some examples of string constants:

"Commodore 128"
"qwer1234!#$%()*.:,"
"" (null string)
"John and Joan"

## VARIABLES: INTEGER, FLOATING-POINT AND STRING

Variables are units of memory storage that represent varying data values within a program. Unlike constants, variables may change in value throughout the course of a program. The value assigned to a variable can be an integer, a floating-point number, or a string. You can assign a value to a variable as the result of a mathematical calculation. Variables are assigned values using an equals sign. The variable name appears to the left of the equals sign and the constant or calculation appears to the right. When you refer to a variable in a program before you assign it a value, the variable value becomes zero if it is an integer or floating-point number. It becomes a null string if the variable is a string.

Variable names can be any length, but for efficiency you should limit the size of the variable to a few characters. Only the first two characters of a variable name are significant. Therefore, do not begin the names of two different variables with the same two characters. If you do, the C128 will interpret them as the same variable name.

The first character of a variable name must be a letter. The rest of the variable name can be any letter or number from zero to nine. A variable name must not contain any BASIC keyword. If you include a BASIC keyword in a variable name, a **SYNTAX ERROR** occurs. BASIC keywords include all BASIC statements, commands, function names, logical operator names and reserved variables.

You can specify the data type of a variable by following the variable name with a percent sign (%) if the variable is an integer value, or a dollar sign if the variable is a string. If no character is specified, the C128 assumes that the variable value is a floating-point number. Here are some examples of variables and how they are assigned:

A = 3.679 (floating-point)
Z% = 714 (integer)
F$ = "CELEBRATE THE COMMODORE 128" (string)
T = A + Z% (floating-point)
Count % = Count % + 1 (integer)
G$ = "SEEK A HIGHER LEVEL OF CONSCIOUSNESS" (string)
H$ = F$ + G$ (string)

## ARRAYS: INTEGER, FLOATING-POINT AND STRING

Although arrays were defined earlier in this chapter as series of related variables or constants, you refer to them with a single integer, floating point or string variable name. All elements have the same data type as the array name. To access successive elements within the array, BASIC uses subscripts (indexed variables) to refer to each unique storage compartment in the array. For example, the alphabet has twenty-six letters. Assume an array called "ALPHA" is constructed and includes all the letters of the alphabet. To access the first element of the array, which is also the first letter of the alphabet (A), label Alpha with a subscript of zero:

```
ALPHA$(0)     A
```

To access the letter B, label Alpha with a subscript of one:

```
ALPHA$(1)     B
```

Continue in the same manner to access all of the elements of the array ALPHA, as in the following:

```
ALPHA$(2)     C
ALPHA$(3)     D
ALPHA$(4)     E
ALPHA$(5)     Z
```

Subscripts are a convenient way to access elements within an array. If subscripts did not exist, you would have to assign separate variables for all the data that would normally be accessed with a subscript. The first subscript within an array is zero.

Although arrays are actually stored sequentially in memory, they can be multi-dimensional. Tables and matrices are easily manipulated with two-dimensional arrays. For example, assume you have a matrix with ten rows and ten columns. You need 100 storage locations or array elements in order to store the whole matrix. Even though your matrix is ten by ten, the elements in the array are stored in memory one after the other for 100 hundred locations.

You specify the number of dimensions in the arrays with the **DIM** statement. For example:

```
10 DIM A(99)
```

dimensions a one-dimensional floating-point array with 100 elements. The following are examples of two-, three- and four-dimensional integer arrays:

```
20 DIM B(9, 9)          (100 elements)
30 DIM C(20,20,20)      (9261 elements)
40 DIM D(10,15,15,10)   (30976 elements)
```

In theory the maximum number of dimensions in an array is 255, but you cannot fit a DIMension statement that long on a 160-character line. The maximum number of DIMension statements you can fit on a 160-character line is approximately fifty. The maximum number of elements allowed in each dimension is 32767. In practice, the size of an array is limited to the amount of available memory. Most arrays are one-, two- or three-dimensional. If an array contains fewer than ten elements, there is no need for a DIM statement since the C128 automatically dimensions variable names to ten elements. The first time you refer to the name of the undimensioned array (variable) name, the C128 assigns zero to the value if it is a numeric array, or a null string if it is a string array.

You must separate the subscript for each dimension in your DIMension statement with a comma. Subscripts can be integer constants, variables, or the integer result of an arithmetic operation. Legal subscript values can be between zero and the highest dimension assigned in the DIMension statement. If the subscript is referred to outside of this range, a **BAD SUBSCRIPT ERROR** results.

The type of array determines how much memory is used to store the integer, floating-point or string data.

Floating-point string arrays take up the most memory; integer arrays require the least amount of memory. Here's how much memory each type of array requires:

```
        5 bytes for the array name
    +   2 bytes for each dimension
    +   2 bytes for each integer array element
OR  +   5 bytes for each floating-point element
OR  +   3 bytes for each string element
AND +   1 byte per character in each string element
```

Keep in mind the amount of storage required for each type of array. If you only need an integer array, specify that the array be the integer type, since floating-point arrays require much more memory than does the integer type.

Here are some example arrays:

```
A$(0) = "GROSS SALES"          (string array)
MTH$(K%) = "JAN"               (string array)
G2%(X) = 5                     (integer array)
CNT%(G2%(X)) = CNT%(I)-2       (integer array)
FP(12*K%) = 24.8               (floating-point array)
```

| | |
|---|---|
| SUM(CNT%(1)) = FP*K% | (floating-point array) |
| A(5) = 0 | Sets the 5th element in the 1 dimensional array called "A" equal to 0 |
| B(5,6) = 26 | Sets the element in row position 5 and column position 6 in the 2 dimensional array called "B" equal to 26 |
| C(1,2,3) = 100 | Sets the element in row position 1, column position 2, and depth position 3 in the 3 dimensional array called "C" equal to 100 |

# EXPRESSIONS AND OPERATORS

Expressions are formed using constants, variables and/or arrays. An expression can be a single constant, simple variable, or an array variable of any type. It also can be a combination of constants and variables with arithmetic, relational or logical operators designed to produce a single value. How operators work is explained below. Expressions can be separated into two classes:

1. ARITHMETIC
2. STRING

Expressions have two or more data items called *operands*. Each operand is separated by a single *operator* to produce the desired result. This is usually done by assigning the value of the expression to a variable name.

An operator is a special symbol the BASIC Interpreter in your Commodore 128 recognizes as representing an operation to be performed on the variables or constant data. One or more operators, combined with one or more variables and/or constants form an expression. Arithmetic, relational and logical operators are recognized by Commodore 128 BASIC.

## ARITHMETIC EXPRESSIONS

Arithmetic expressions yield an integer or floating-point value. The arithmetic operators (+,–,*,/, ↑) are used to perform addition, subtraction, multiplication, division and exponentiation operations, respectively.

## ARITHMETIC OPERATIONS

An arithmetic operator defines an arithmetic operation which is performed on the two operands on either side of the operator. Arithmetic operations are performed using floating-point numbers. Integers are converted to floating-point numbers before an arithmetic operation is performed. The result is converted back to an integer if it is assigned to an integer variable name.

## ADDITION (+)

The plus sign (+) specifies that the operand on the right is added to the operand on the left.

### EXAMPLES:

```
2 + 2
A + B + C
X% + 1
BR + 10E-2
```

## SUBTRACTION (–)

The minus sign (–) specifies that the operand on the right is subtracted from the operand on the left.

### EXAMPLES:

```
4–1
100–64
A–B
55–142
```

The minus also can be used as a unary minus which is the minus sign in front of a negative number. This is equal to subtracting the number from zero (0).

### EXAMPLES:

```
–5
–9E4
–B
4–(–2) (same as 4 + 2)
```

## MULTIPLICATION (*)

An asterisk (*) specifies that the operand on the left is multiplied by the operand on the right.

### EXAMPLES:

```
100*2
50*0
A*X1
R%*14
```

## DIVISION (/)

The slash (/) specifies that the operand on the left is divided by the operand on the right.

**EXAMPLES:**

10/2
6400/4
A/B
4E2/XR

## EXPONENTIATION ( ↑ )

The up arrow ( ↑ ) specifies that the operand on the left is raised to the power specified by the operand on the right (the exponent). If the operand on the right is a 2, the number on the left is squared; if the exponent is a 3, the number on the left is cubed, etc. The exponent can be any number as long as the result of the operation gives a valid floating-point number.

**EXAMPLES:**

2 ↑ 2 Equivalent to 2*2
3 ↑ 3 Equivalent to 3*3*3
4 ↑ 4 Equivalent to 4*4*4*4
AB ↑ CD
3 ↑ –2 Equivalent to ⅓*⅓

# RELATIONAL OPERATORS

The relational operators (<, =, >, <=, >=, <>) are primarily used to compare the values of two operands, but they also produce an arithmetic result. The relational operators and the logical operators (**AND, OR**, and **NOT**), when used in comparisons, produce an arithmetic true/false evaluation of an expression. If the relationship stated in the expression is true, the result is assigned an integer value of –1. If it's false a value of 0 is assigned. Following are the relational operators:

| | |
|---|---|
| < | LESS THAN |
| = | EQUAL TO |
| > | GREATER THAN |
| <= | LESS THAN OR EQUAL TO |
| >= | GREATER THAN OR EQUAL TO |
| <> | NOT EQUAL TO |

**EXAMPLES:**

5–4 = 1 result true (–1)
14>66 result false (0)
15>= 15 result true (–1)

Relational operators may be used to compare strings. For comparison purposes, the letters of the alphabet have the order A<B<C<D, etc. Strings are compared by

evaluating the relationship between corresponding characters from left to right (see string operations).

**EXAMPLES:**

| | |
|---|---|
| "A" < "B" | result true (–1) |
| "X" = "YY" | result false (0) |
| BB$ <> CC$ | result false (0) if they are the same |

Numeric data items can only be compared (or assigned) with other numeric items. The same is true when comparing strings; otherwise, the BASIC error message **?TYPE MISMATCH** occurs. Numeric operands are compared by first converting the values of either or both operands from integer to floating-point form, as necessary. Then the relationship between the floating-point values is evaluated to give a true/false result.

At the end of all comparisons, you get an integer regardless of the data type of the operand (even if both are strings). Because of this, a comparison of two operands can be used as an operand in performing calculations. The result will be –1 or 0 and can be used as anything but a divisor, since division by zero is illegal.

## LOGICAL OPERATORS

The logical operators (AND, OR, NOT) can be used to modify the meaning of the relational operators or to produce an arithmetic result. Logical operators can produce results other than –1 and 0, although any nonzero result is considered true when testing for a true/false condition.

The logical operators (sometimes called *Boolean* operators) also can be used to perform logical operations on individual binary digits (bits) in two operands. But when you're using the NOT operator, the operation is performed only on the single operand to the right. The operands must be in the integer range of values (–32768 to +32767) (floating-point numbers are converted to integers) and logical operations give an integer result.

Logical operations are performed bit-by-corresponding-bit on the two operands. The logical AND produces a bit result of 1 only if both operand bits are 1. The logical OR produces a bit result of 1 if either operand bit is 1. The logical NOT is the opposite value of each bit as a single operand. In other words, "If it's NOT 1 then it is 0. If it's NOT 0 then it is 1."

The exclusive OR IF (**XOR**) doesn't have a logical operator but it is performed as part of the **WAIT** statement or as the XOR function. Exclusive-OR means that if the bits of two operands are set and equal, then the result is 0; otherwise the result is 1.

Logical operations are defined by groups of statements which, when taken together, constitute a Boolean "truth table" as shown in Table 2–1.

**The AND operation results in a 1 only if both bits are 1:**

**1 AND 1 = 1**
**0 AND 1 = 0**
**1 AND 0 = 0**
**0 AND 0 = 0**

**The OR operation results in a 1 if either bit is 1:**

**1 OR 1 = 1**
**0 OR 1 = 1**
**1 OR 0 = 1**
**0 OR 0 = 0**

**The NOT operation logically complements each bit:**

**NOT 1 = 0**
**NOT 0 = 1**

**The exclusive OR (XOR) is a function (not a logical operator):**

**1 XOR 1 = 0**
**1 XOR 0 = 1**
**0 XOR 1 = 1**
**0 XOR 0 = 0**

**Table 2–1 Boolean Truth Table**

The logical operators AND, OR and NOT specify a Boolean arithmetic operation to be performed on the two operand expressions on either side of the operator. In the case of NOT, *only* the operand on the *right* is considered. Logical operations (or Boolean arithmetic) aren't performed until all arithmetic and relational operations in an expression have been evaluated.

### EXAMPLES:

```
IF A = 100 AND B = 100 THEN 10   (if both A and B have a value of 100 then
                                  the result is true)
A = 96 AND 32: PRINT A           (A = 32)
IF A = 100 OR B = 100 THEN 20    (if A or B is 100 then the result is true)
A = 64 OR 32: PRINT A            (A = 96)
X = NOT 96                       (result is –97 (two's complement))
```

## HIERARCHY OF OPERATIONS

All expressions perform the different types of operations according to a fixed hierarchy. Certain operations have a higher priority and are performed before other operations. The normal order of operations can be modified by enclosing two or more operands within

parentheses ( ), creating a "subexpression." The parts of an expression enclosed in parentheses will be reduced to a single value before evaluating parts outside the parentheses.

When you use parentheses in expressions, pair them so that you always have an equal number of left and right parentheses. If you don't, the BASIC error message **?SYNTAX ERROR** will occur.

Expressions that have operands inside parentheses may themselves be enclosed in parentheses, forming complex expressions of multiple levels. This is called *nesting*. Parentheses can be nested in expressions to a maximum depth of ten levels—ten matching sets of parentheses. The innermost expression has its operations performed first. Some examples of expressions are:

```
A+B
C↑(D+E)/2
((X–C↑ (D+E)/2)*10)+1
GG$>HH$
JJ$+"MORE"
K%=1 AND M<>X
K%=2 OR (A=B AND M<X)
NOT (D=E)
```

The BASIC Interpreter performs operations on expressions by performing arithmetic operations first, then relational operations, and logical operations last. Both arithmetic and logical operators have an order of precedence (or hierarchy of operations) within themselves. Relational operators do not have an order of precedence and will be performed as the expression is evaluated from left to right.

If all remaining operators in an expression have the same level of precedence, then operations are performed from left to right. When performing operations on expressions within parentheses, the normal order of precedence is maintained. The hierarchy of arithmetic and logical operations is shown in Table 2-2 from first to last in order of precedence. Note that scientific notation is resolved first.

| OPERATOR | DESCRIPTION | EXAMPLE |
|---|---|---|
| ↑ | **Exponentiation** | **BASE ↑ EXP** |
| – | **Negation (Unary Minus)** | **–A** |
| */ | **Multiplication**<br>**Division** | **AB * CD**<br>**EF / GH** |
| + | **Addition** | **CNT + 2** |
| – | **Subtraction** | **JK – PQ** |
| > = < | **Relational Operations** | **A <= B** |
| **NOT** | **Logical NOT (Integer Two's Complement)** | **NOT K%** |
| **AND** | **Logical AND** | **JK AND 128** |
| **OR** | **Logical OR** | **PQ OR 15** |

**Table 2–2 Hierarchy of Operations Performed on Expressions**

## STRING OPERATIONS

Strings are compared using the same relational operators (=, <>,<=, >=, <,>) that are used for comparing numbers. String comparisons are made by taking one character at a time (left-to-right) from each string and evaluating each character code position from the character set. If the character codes are the same, the characters are equal. If the character codes differ, the character with the lower CBM ASCII code number is lower in the character set. The comparison stops when the end of either string is reached. All other factors being equal, the shorter string is considered less than the longer string. *Leading* or *trailing blanks* are significant in string evaluations.

Regardless of the data types, all comparisons yield an integer result. This is true even if both operands are strings. Because of this, a comparison of two string operands can be used as an operand in performing calculations. The result will be –1 or 0 (true or false) and can be used in any mathematical operation but division since division by zero is illegal.

## STRING EXPRESSIONS

Expressions are treated as if an implied "<>0" follows them. This means that if an expression is true, the next BASIC statement on the same program line is executed. If the expression is false, the rest of the line is ignored and the next line in the program is executed.

Just as with numbers, you can perform operations on string variables. The only arithmetic string operator recognized by BASIC 7.0 is the plus sign (+) which is used to perform concatenation of strings. When strings are concatenated, the string on the right of the plus sign is appended to the string on the left, forming a third string. The result can be printed immediately, used in a comparison, or assigned to a variable name. If a string data item is compared with (or set equal to) a numeric item, or vice-versa, the BASIC error message **?TYPE MISMATCH** occurs. Some examples of string expressions and concatenation are:

```
10 A$="FILE": B$="NAME"
20 NAM$ = A$ + B$              (yields the string "FILENAME")
30 RES$ = "NEW" + A$ + B$  (yields the string "NEWFILENAME")
```

# ORGANIZATION OF THE BASIC 7.0 ENCYCLOPEDIA

This section of Chapter 2 lists BASIC 7.0 language elements in an encyclopedia format. It provides an abbreviated list of the rules (syntax) of Commodore 128 BASIC 7.0, along with a concise description of each. Consult the *Commodore 128 System Guide BASIC 7.0 Encyclopedia* (Chapter 5) included with your computer for a

more detailed description of each command. BASIC 7.0 includes all the elements of BASIC 2.0.

The different types of BASIC operations are listed in individual sections, as follows:

1. **Commands** and **Statements:** the commands used to edit, store and erase programs, and the BASIC program statements used in the numbered lines of a program.
2. **Functions:** the string, numeric and print functions.
3. **Reserved Words and Symbols:** the words and symbols reserved for use by the BASIC 7.0 language, which cannot be used for any other purpose.

# COMMAND AND STATEMENT FORMAT

The command and statement definitions in this encyclopedia are arranged in the following format:

---

Command name— **AUTO**

Brief definition— Enable/disable automatic line numbering

Command format— **AUTO [line#]**

Discussion of format and use— This command turns on the automatic line-numbering feature. This eases the job of entering programs, by automatically typing the line numbers for the user. As each program line is entered by pressing RETURN, the next line number is printed on the screen, and the cursor is positioned two spaces to the right of the line number. The line number argument refers to the desired increment between line numbers. AUTO without an argument turns off the auto line numbering, as does RUN. This statement can be used only in direct mode (outside of a program).

**EXAMPLES:**

Example(s)—

| | |
|---|---|
| AUTO 10 | Automatically numbers program lines in increments of 10. |
| AUTO 50 | Automatically numbers lines in increments of 50. |
| AUTO | Turns off automatic line numbering. |

---

The boldface line that defines the format consists of the following elements:

**DLOAD "program name" [,D0,U8]**

**keyword** — **argument** — **additional arguments (possibly optional)**

The parts of the command or statement that must be typed exactly as shown are in capital letters. Words the user supplies, such as the name of a program, are not capitalized.

When quote marks (" ") appear (usually around a program name or filename), the user should include them in the appropriate place, according to the format example.

**Keywords** are words that are part of the BASIC language. They are the central part of a command or statement, and they tell the computer what kind of action to take. These words cannot be used as variable names. A complete list of reserved words and symbols is given at the end of this chapter.

Keywords, also called reserved words, appear in upper-case letters. Keywords may be typed using the full word or the approved abbreviation. (A full list of abbreviations is given in Appendix I). The keyword or abbreviation must be entered correctly or an error will result. The BASIC and DOS error messages are defined in Appendices A and B, respectively.

**Arguments,** also called parameters, appear in lower-case letters. Arguments complement keywords by providing specific information to the command or statement. For example, the keyword LOAD tells the computer to load a program while the argument "program name" tells the computer which specific program to load. A second argument specifies from which drive to load the program. Arguments include filenames, variables, line numbers, etc.

**Square Brackets [ ]** show optional arguments. The user selects any or none of the arguments listed, depending on requirements.

**Angle Brackets** <> indicate the user MUST choose one of the arguments listed.

**A Vertical Bar** | separates items in a list of arguments when the choices are limited to those arguments listed. When the vertical bar appears in a list enclosed in SQUARE BRACKETS, the choices are limited to the items in the list, but the user still has the option not to use any arguments. If a vertical bar appears within angle brackets, the user MUST choose one of the listed arguments.

**Ellipsis . . .** (a sequence of three dots) means an option or argument can be repeated more than once.

**Quotation Marks " "** enclose character strings, filenames and other expressions. When arguments are enclosed in quotation marks, the quotation marks must be included in the command or statement. In this encyclopedia, quotation marks are not conventions used to describe formats; they are required parts of a command or statement.

**Parentheses ( )** When arguments are enclosed in parentheses, they must be included in the command or statement. Parentheses are not conventions used to describe formats; they are required parts of a command or statement.

**Variable** refers to any valid BASIC variable names, such as X, A$, T%, etc.
**Expression** refers to any valid BASIC expressions, such as A+B+2,.5*(X+3), etc.

> **NOTE:** For all DOS commands, variables and expressions used as arguments must be endorsed in parentheses.

# BASIC COMMANDS AND STATEMENTS

## APPEND

Append data to the end of a sequential file

**APPEND #logical file number,"filename"[,Ddrive number][<ON|,>Udevice]**

### EXAMPLES:

| | |
|---|---|
| Append # 8, "MYFILE" | OPEN logical file 8 called "MYFILE", and prepare to append with subsequent PRINT # statements. |
| Append #7,(A$),D0,U9 | OPEN logical file named by the variable in A$ on drive 0, device number 9, and prepare to APPEND. |

## AUTO

Enable/disable automatic line numbering

**AUTO [line#]**

### EXAMPLES:

| | |
|---|---|
| AUTO 10 | Automatically numbers program lines in increments of 10. |
| AUTO 50 | Automatically numbers lines in increments of 50. |
| AUTO | Turns off automatic line numbering. |

## BACKUP

Copy the entire contents from one disk to another on a dual disk drive

**BACKUP source Ddrive number TO destination Ddrive number [<ON|,> Udevice]**

> **NOTE:** This command can be used only with a dual-disk drive.

**EXAMPLES:**

| | |
|---|---|
| BACKUP D0 TO D1 | Copies all files from the disk in drive 0 to the disk in drive 1, in dual disk drive unit 8. |
| BACKUP D0 TO D1 ON U9 | Copies all files from drive 0 to drive 1, in disk drive unit 9. |

## BANK

Select one of the 16 BASIC banks (default memory configurations), numbered 0–15 to be used during PEEK, POKE, SYS, and WAIT commands.

**BANK bank number**

Here is a table of available BANK configurations in the Commodore 128 memory:

| BANK | CONFIGURATION |
|---|---|
| 0 | **RAM(0) only** |
| 1 | **RAM(1) only** |
| 2 | **RAM(2) only (same as 0)** |
| 3 | **RAM(3) only (same as 1)** |
| 4 | **Internal ROM , RAM(0), I/O** |
| 5 | **Internal ROM , RAM(1), I/O** |
| 6 | **Internal ROM , RAM(2), I/O (same as 4)** |
| 7 | **Internal ROM , RAM(3), I/O (same as 5)** |
| 8 | **External ROM , RAM(0), I/O** |
| 9 | **External ROM , RAM(1), I/O** |
| 10 | **External ROM , RAM(2), I/O (same as 8)** |
| 11 | **External ROM , RAM(3), I/O (same as 9)** |
| 12 | **Kernal and Internal ROM (LOW), RAM(0), I/O** |
| 13 | **Kernal and External ROM (LOW), RAM(0), I/O** |
| 14 | **Kernal and BASIC ROM, RAM(0), Character ROM** |
| 15 | **Kernal and BASIC ROM, RAM(0), I/O** |

Banks are described in detail in Chapter 8, The Power Behind Commodore 128 Graphics and Chapter 13, The Commodore 128 Operating System.

## BEGIN / BEND

A conditional statement like IF . . . THEN: ELSE, structured so that you can include several program lines between the start (BEGIN) and end (BEND) of the structure. Here's the format:

**IF condition THEN BEGIN : statement**
**statement**
**statement BEND : ELSE BEGIN**
**statement**
**statement BEND**

**EXAMPLE**

```
10 IF X = 1 THEN BEGIN: PRINT "X = 1 is True"
20 PRINT "So this part of the statement is performed"
30 PRINT "When X equals 1"
40 BEND: PRINT "End of BEGIN/BEND structure":GO to 60
50 PRINT "X does not equal 1":PRINT "The statements between BEGIN/
BEND are skipped"
60 PRINT "Rest of Program"
```

## BLOAD

Load a binary file starting at the specified memory location

**BLOAD "filename"[,Ddrive number][<ON|,U>device number] [,Bbank number] [,Pstart address]**

where:

- filename is the name of your file
- bank number selects one of the 16 BASIC banks (default memory configurations)
- start address is the memory location where loading begins

**EXAMPLES:**

| | |
|---|---|
| BLOAD "SPRITES", B0, P3584 | LOADS the binary file "SPRITES" starting in location 3584 (in BANK 0). |
| BLOAD "DATA1", D0, U8, B1, P4096 | LOADS the binary file "DATA 1" into location 4096 (BANK 1) from Drive 0, unit 8. |

## BOOT

Load and execute a program which was saved as a binary file

**BOOT "filename"[,Ddrive number][<ON|,>Udevice][,Palt LOAD ADD]**

**EXAMPLE:**

| | |
|---|---|
| BOOT | BOOT a bootable disk (CP/M Plus for example). |

| | |
|---|---|
| BOOT ''GRAPHICS 1'', D0, U9 | LOADS the binary program ''GRAPHICS 1'' from unit 9, drive 0, and executes it. |

## BOX

Draw box at specified position on screen

**BOX [color source], X1, Y1[,X2,Y2][,angle][,paint]**

where:

| | |
|---|---|
| **color source** | 0 = Background color<br>1 = Foreground color (DEFAULT)<br>2 = Multi-color 1<br>3 = Multi-color 2 |
| **X1,Y1** | Corner coordinate (scaled) |
| **X2,Y2** | Corner diagonally opposite X1, Y1, (scaled); default is the PC location. |
| **angle** | Rotation in clockwise degrees; default is 0 degrees |
| **paint** | Paint shape with color<br>0 = Do not paint<br>1 = Paint<br>(default = 0) |

### EXAMPLES:

| | |
|---|---|
| BOX 1, + 10, + 10 | Draw a box 10 pixels to the right and 10 down from the current pixel cursor location. |
| BOX 1, 10, 10, 60, 60 | Draws the outline of a rectangle. |
| BOX , 10, 10, 60, 60, 45, 1 | Draws a painted, rotated box (a diamond). |
| BOX , 30, 90, , 45, 1 | Draws a filled, rotated polygon. |

Any parameter can be omitted but you must include a comma in its place, as in the last two examples.

**NOTE:** Wrapping occurs if the degree is greater than 360.

# BSAVE

Save a binary file from the specified memory locations

**BSAVE "filename"[,Ddrive number][<ON|,U>device number] [,Bbank number],Pstart address TO Pend address**

where:

- start address is the starting address where the program is SAVEd from
- end address is the last address + 1 in memory which is SAVEd

This is similar to the SAVE command in the Machine Language MONITOR.

**EXAMPLES:**

| | |
|---|---|
| BSAVE "SPRITE DATA",B0, P3584 TO P4096 | Saves the binary file named "SPRITE DATA", starting at location 3584 through 4095 (BANK 0). |
| BSAVE "PROGRAM.SCR",D0, U9,B0,P3182 TO P7999 | Saves the binary file named "PROGRAM. SCR" in the memory address range 3182 through 7999 (BANK 0) on drive 0, unit 9. |

# CATALOG

Display the disk directory

**CATALOG [Ddrive number][<ON|,>Udevice number][,wildcard string]**

**EXAMPLE:**

CATALOG Displays the disk directory on drive 0.

# CHAR

Display characters at the specified position on the screen

**CHAR [color source],X,Y[,string][,RVS]**

This is primarily designed to display characters on a bit mapped screen, but it can also be used on a text screen. Here's what the parameters mean:

| | |
|---|---|
| **color source** | 0 = Background<br>1 = Foreground |
| **X** | Character column (0–39) (VIC screen)<br>(0–79) (8563) screen |

| | |
|---|---|
| **Y** | Character row (0–24) |
| **string** | String to print |
| **reverse** | Reverse field flag (0 = off, 1 = on) |

**EXAMPLE:**

```
10 COLOR 2,3: REM MULTI-COLOR 1 = RED
20 COLOR 3,7: REM MULTI-COLOR 2 = BLUE
30 GRAPHIC 3,1
30 CHAR 0,10,10, "TEXT",0
```

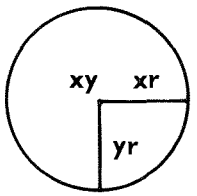
## CIRCLE

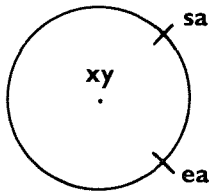
Draw circles, ellipses, arcs, etc., at specified positions on the screen

**CIRCLE [color source],X,Y[,Xr][,Yr] [,sa][,ea][,angle][,inc]**

where:

| | |
|---|---|
| **color source** | 0 = background color<br>1 = foreground color<br>2 = multi-color 1<br>3 = multi-color 2 |
| **X,Y** | Center coordinate of the CIRCLE |
| **Xr** | X radius (scaled); (default = 0) |
| **Yr** | Y radius (sealed default is Xr) |
| **sa** | Starting arc angle (default 0 degrees) |
| **ea** | Ending arc angle (default 360 degrees) |
| **angle** | Rotation is clockwise degrees (default is 0 degrees) |
| **inc** | Degrees between segments (default is 2 degrees) |




**EXAMPLES:**

CIRCLE1, 160,100,65,10 Draws an ellipse.

CIRCLE1, 160,100,65,50 Draws a circle.

| | |
|---|---|
| CIRCLE1, 60,40,20,18,,,,45 | Draws an octagon. |
| CIRCLE1, 260,40,20,,,,,90 | Draws a diamond. |
| CIRCLE1, 60,140,20,18,,,,120 | Draws a triangle. |
| CIRCLE 1,+2,+2,50,50 | Draws a circle (two pixels down and two to the right) relative to the original coordinates of the pixel cursor. |
| CIRCLE1, 30;90 | Draws a circle 30 pixels and 90 degrees to the right of the current pixel cursor coordinate position. |

You may omit a parameter, but you must still place a comma in the appropriate position. Omitted parameters take on the default values.

## CLOSE

Close logical file

**CLOSE file number**

### EXAMPLE:

CLOSE 2 Logical file 2 is closed.

## CLR

Clear program variables

**CLR**

## CMD

Redirect screen output to a logical disk or print file.

**CMD logical file number [,write list]**

### EXAMPLE:

| | |
|---|---|
| OPEN 1,4 | Opens device 4 (printer). |
| CMD 1 | All normal output now goes to the printer. |
| LIST | The LISTing goes to the printer, not the screen—even the word READY. |
| PRINT#1 | Sends output back to the screen. |
| CLOSE 1 | Closes the file. |

# COLLECT

Free inaccessible disk space

**COLLECT [Ddrive number][<ON|,>Udevice]**

**EXAMPLE:**

COLLECT D0 Free all available space which has been incorrectly allocated to improperly closed files. Such files are indicated with an asterisk on the disk directory.

# COLLISION

Define handling for sprite collision interrupt

**COLLISION type [,statement]**

**type** Type of interrupt, as follows:
1 = Sprite-to-sprite collision
2 = Sprite-to-display data collision
3 = Light pen (VIC screen only)

**statement** BASIC line number of a subroutine

**EXAMPLE:**

Collision 1, 5000 Enables a sprite-to-sprite collision and program control sent to subroutine at line 5000.

Collision 1 Stops interrupt action which was initiated in above example.

Collision 2, 1000 Enables a sprite-to-data collision and program control directed to subroutine in line 1000.

# COLOR

Define colors for each screen area

**COLOR source number, color number**

This statement assigns a color to one of the seven color areas:

| AREA | SOURCE |
|---|---|
| **0** | **40-column (VIC) background** |
| **1** | **40-column (VIC) foreground** |
| **2** | **multicolor 1** |
| **3** | **multicolor 2** |
| **4** | **40-column (VIC) border** |
| **5** | **character color (40- or 80-column screen)** |
| **6** | **80-column background color** |

Colors that are usable are in the range 1-16.

| COLOR CODE | COLOR | COLOR CODE | COLOR |
|---|---|---|---|
| 1 | Black | 9 | Orange |
| 2 | White | 10 | Brown |
| 3 | Red | 11 | Light Red |
| 4 | Cyan | 12 | Dark Gray |
| 5 | Purple | 13 | Medium Gray |
| 6 | Green | 14 | Light Green |
| 7 | Blue | 15 | Light Blue |
| 8 | Yellow | 16 | Light Gray |

**Color Numbers in 40-Column Output**

| | | | |
|---|---|---|---|
| 1 | Black | 9 | Dark Purple |
| 2 | White | 10 | Dark Yellow |
| 3 | Dark Red | 11 | Light Red |
| 4 | Light Cyan | 12 | Dark Cyan |
| 5 | Light Purple | 13 | Medium Gray |
| 6 | Dark Green | 14 | Light Green |
| 7 | Dark Blue | 15 | Light Blue |
| 8 | Light Yellow | 16 | Light Gray |

**Color Numbers in 80-Column Output**

### EXAMPLES:

COLOR 0, 1: Changes background color of 40-column screen to black.

COLOR 5, 8: Changes character color to yellow.

## CONCAT

Concatenate two data files

**CONCAT "file 2" [,Ddrive number] TO "file 1"**
**[,Ddrive number][<ON|,>Udevice]**

### EXAMPLE:

Concat "File B" to "File A" FILE B is attached to FILE A, and the combined file is designated FILE A.

Concat (A$) to (B$), D1, U9 The file named by B$ becomes a new file with the same name with the file named by A$ attached to the end of B$. This is performed on Unit 9, drive 1 (a dual disk drive).

Whenever a variable is used as a filename, as in the last example, the filename variable must be within parentheses.

## CONT

Continue program execution

**CONT**

## COPY

Copy a file from one drive to another within a dual disk drive. Copy one file to another with a different name within a single drive

**COPY [Ddrive number,]"source filename"TO[Ddrive number,]"destination filename"[<ON|,>Udevice]**

> **NOTE:** Copying between two single or double disk drive units cannot be done. This command does not support unit-to-unit copying.

### EXAMPLES:

| | |
|---|---|
| COPY D0, "TEST" TO D1, "TEST PROG" | Copies "test" from drive 0 to drive 1, renaming it "test prog" on drive 1. |
| COPY D0, "STUFF" TO D1, "STUFF" | Copies "STUFF" from drive 0 to drive 1. |
| COPY D0 TO D1 | Copies all files from drive 0 to drive 1. |
| COPY "WORK.PROG" TO "BACKUP" | Copies "WORK.PROG" as a file called "BACKUP" on the same disk (drive 0). |

## DATA

Define data to be used by a program

**DATA list of constants**

### EXAMPLE:

DATA 100, 200, FRED, "HELLO, MOM",, 3, 14, ABC123

# DCLEAR

Clear all open channels on disk drive

**DCLEAR [Ddrive number][<ON|,>Udevice]**

**EXAMPLES:**

DCLEAR D0 Clears all open files on drive 0, device number 8.

DCLEAR D1,U9 Clears all open files (channels) on drive 1, device number 9.

# DCLOSE

Close disk file

**DCLOSE [#logical file number][<ON|,>Udevice]**

**EXAMPLES:**

DCLOSE Closes all channels currently open on unit 8.

DCLOSE #2 Closes the channel associated with the logical file number 2 on unit 8.

DCLOSE ON U9 Closes all channels currently open on unit 9.

# DEF FN

Define a user function

**DEF FN name (variable) = expression**

**EXAMPLE:**

```
10 DEF FNA(X) = 12*(34.75-X/.3) + X
20 PRINT FNA(7)
```

The number 7 is inserted each place X is located in the formula given in the DEF statement. In the example above, the answer returned is 144.

> **NOTE:** If you plan to define a function in a program that will use BASIC 7.0 graphics commands, invoke the GRAPHIC command before defining your function. The portion of memory where functions are defined and where the graphics screen is located is shared. Once you allocate your graphics area, the function definitions are safely placed somewhere else in memory. If you don't take this precaution and you invoke the GRAPHIC command **after** you define a function, the function definition (between $1C00 and $4000) is destroyed.

# DELETE

Delete lines of a BASIC program in the specified range

**DELETE [first line] [-last line]**

### EXAMPLES:

| | |
|---|---|
| DELETE 75 | Deletes line 75. |
| DELETE 10-50 | Deletes lines 10 through 50, inclusive. |
| DELETE-50 | Deletes all lines from the beginning of the program up to and including line 50. |
| DELETE 75- | Deletes all lines from 75 to the end of the program, inclusive. |

# DIM

Declare number of elements in an array

**DIM variable (subscripts) [,variable(subscripts)] . . .**

### EXAMPLE:

10 DIM A$(40),B7(15),CC%(4,4,4)

Dimension three arrays where arrays A$, B7 and CC% have 41 elements, 16 elements and 125 elements respectively.

# DIRECTORY

Display the contents of the disk directory on the screen

**DIRECTORY [Ddrive number][<ON|,>Udevice][,wildcard]**

### EXAMPLES:

| | |
|---|---|
| DIRECTORY | Lists all files on the disk in unit 8. |
| DIRECTORY D1, U9, "WORK" | Lists the file named "WORK," on drive 1 of unit 9. |
| DIRECTORY "AB*" | Lists all files starting with the letters "AB" like ABOVE, ABOARD, etc. on unit 8. The asterisk specifies a wild card, where all files starting with "AB" are displayed. |
| DIRECTORY D0, "?.BAK" | The ? is a wild card that matches any single character in that position. For example: FILE 1.BAK, FILE 2.BAK, FILE 3.BAK all match the string. |

DIRECTORY D1,U9,(A$) LISTS the filename stored in the variable A$ on device number 9, drive 1. Remember, whenever a variable is used as a filename, put the variable in parentheses.

> **NOTE:** To print the DIRECTORY of the disk in drive 0, unit 8, use the following example:
>
> LOAD"$0",8
> OPEN4,4:CMD4:LIST
> PRINT#4:CLOSE4

## DLOAD

Load a BASIC program from the disk drive, device 8.

**DLOAD "filename" [,Ddrive number][<ON|,>Udevice number]**

### EXAMPLES:

DLOAD "BANKRECS" Searches the disk for the program "BANKRECS" and LOADS it.

DLOAD (A$) LOADS a program from disk in which the name is stored in the variable A$. An error message is given if A$ is null. Remember, when a variable is used as a filename, it must be enclosed in parentheses.

## DO / LOOP / WHILE / UNTIL / EXIT

Define and control a program loop

**DO [UNTIL condition | WHILE condition]**
**statements [EXIT]**
**LOOP [UNTIL condition | WHILE condition]**

This loop structure performs the statements between the DO statement and the LOOP statement. If no UNTIL or WHILE modifies either the DO or the LOOP statement, execution of the statements in between continues indefinitely. If an EXIT statement is encountered in the body of a DO loop, execution is transferred to the first statement following the LOOP statement. DO loops may be nested, following the rules defined by the FOR-NEXT structure. If the UNTIL parameter is specified, the program continues looping until the condition is satisfied (becomes true). The WHILE parameter is the opposite of the UNTIL parameter: the program continues looping as long as the condition is TRUE. As soon as the condition is no longer true, program control resumes with the statement immediately following the LOOP statement. An example of a condition (boolean argument) is A = 1, or G>65.

**EXAMPLES:**

```
10 X = 25
20 DO UNTIL X = 0
30 X = X–1
40 PRINT ''X='';X
50 LOOP
60 PRINT ''End of Loop''
```

This example performs the statements X = X-1 and PRINT ''X='';X until X = 0. When X = 0 the program resumes with the PRINT ''End of Loop'' statement immediately following LOOP.

```
10 DO WHILE A$<> CHR$ (13):GETKEY A$:PRINT A$:LOOP
20 PRINT ''THE RETURN KEY HAS BEEN PRESSED''
```

This DO loop waits for a key to be pressed, receives input from the keyboard one character at a time and prints the letter of the key which is pressed. If the RETURN key is pressed, control is transferred out of the loop and line 20 is executed.

```
10 DOPEN #8, ''SEQFILE''
20 DO
30 GET #8,A$
40 PRINT A$;
50 LOOP UNTIL ST
60 DCLOSE #8
```

This program opens file ''SEQFILE'' and gets data until the ST system variable indicates all data is input.

# DOPEN

Open a disk file for a read and/or write operation

**DOPEN # logical file number,''filename[,<type>]''[,Lrecord length] [,Ddrive number][<ON|,>Udevice number][,W]**

where **type** is:

**S** = Sequential File Type
**P** = Program File Type
**U** = User File Type
**R** = Relative File Type
**L** = Record Length = the length of records in a relative file only
**W** = Write Operation (if not specified a read operation occurs)

**EXAMPLES:**

```
DOPEN#1, ''ADDRESS'',W
```

Create the sequential file number 1 (ADDRESS) for a write operation

```
DOPEN#2 ''RECIPES'',D1,U9
```

Open the sequential file number 2 (RECIPES) for a read operation on device number 9, drive 1

# DRAW

Draw dots, lines and shapes at specified positions on the screen

**DRAW [color source] [,X1, Y1][TO X2, Y2] . . .**

where:

| | |
|---|---|
| **Color source** | 0 = Bit map background |
| | 1 = Bit map foreground |
| | 2 = Multi-color 1 |
| | 3 = Multi-color 2 |
| **X1,Y1** | Starting coordinate (0,0 through 319,199) |
| **X2,Y2** | Ending coordinate (0,0 through 319,199) |

**EXAMPLES:**

| | |
|---|---|
| DRAW 1, 100, 50 | Draw a dot. |
| DRAW , 10,10 TO 100,60 | Draw a line. |
| DRAW , 10,10 TO 10,60 TO 100,60 TO 10,10 | Draw a triangle. |
| DRAW 1, 120;45 | Draw a dot 45° relative and 120 pixels away from the current pixel cursor position. |
| DRAW | Draw a dot at the present pixel cursor position. Use LOCATE to position the pixel cursor. |

You may omit a parameter but you still must include the comma that would have followed the unspecified parameter. Omitted parameters take on the default values.

# DSAVE

Save a BASIC program file to disk

**DSAVE "filename" [,Ddrive number][<ON|,>Udevice number]**

**EXAMPLES:**

| | |
|---|---|
| DSAVE "BANKRECS" | Saves the program "BANKRECS" to disk. |
| DSAVE (A$) | Saves the disk program named in the variable A$. |
| DSAVE "PROG 3",D1,U9 | Saves the program "PROG3" to disk on unit number 9, drive 1. |

## DVERIFY

Verify the program in memory against the one on disk

**DVERIFY "filename"[,Ddrive number][<ON|,>Udevice number]**

To verify **Binary** data, see VERIFY "filename",8,1 format, under VERIFY command description.

### EXAMPLES:

| | |
|---|---|
| DVERIFY "C128" | Verifies program "C128" on drive 0, unit 8. |
| DVERIFY "SPRITES",D0,U9 | Verifies program "SPRITES" on drive 0, device 9. |

## END

Define the end of program execution

**END**

## ENVELOPE

Define a musical instrument envelope

**ENVELOPE n[,atk] [,dec] [,sus] [,rel][,wf] [,pw]**

where:

| | |
|---|---|
| **n** | Envelope number (0-9) |
| **atk** | Attack rate (0-15) |
| **dec** | Decay rate (0-15) |
| **sus** | Sustain (0-15) |
| **rel** | Release rate (0-15) |
| **wf** | Waveform: 0 = triangle<br>1 = sawtooth<br>2 = variable pulse (square)<br>3 = noise<br>4 = ring modulation |
| **pw** | Pulse width (0-4095) |

See the "T" option in the PLAY command to select an envelope in a PLAY string.

### EXAMPLE:

| | |
|---|---|
| ENVELOPE 1, 10, 5, 10, 0, 2, 2048 | This command sets envelope 1 to Attack = 10, Decay = 5, Sustain = 10, Release = 0, waveform = variable pulse (2), and the pulse width = 2048 |

# FAST

Sets the 8502 microprocessor at a speed of 2MHz.

**FAST**

This command initiates 2MHz mode, causing the VIC 40-column screen to be turned off. All operations are speeded up considerably. Graphics may be used, but will not be visible until a SLOW command is issued. The Commodore 128 powers up in 1MHz mode. The DMA operations (FETCH, SWAP, STASH) must be performed at 1MHz (slow) speed.

# FETCH

Get data from expansion (RAM module) memory

**FETCH #bytes, intsa, expsa, expb**

where **bytes** = Number of bytes to get from expansion memory (0-65535) where 0 = 64K (65535 bytes)
**intsa** = Starting address of host RAM (0-65535)
**expb** = 64K expansion RAM bank number (0-7) where expb = 0–1 for 128K and expb = 0–7 for up to 512K.
**expsa** = Starting address of expansion RAM (0-65535)

The host BANK for the ROM and I/O configuration is selected with the BANK command. The DMA(VIC) RAM bank is selected by bits 6 and 7 of the RAM configuration register within the MMU($D506).

# FILTER

Define sound (SID chip) filter parameters

**FILTER [freq][,lp] [,bp] [,hp] [,res]**

where:

| | |
|---|---|
| **freq** | Filter cut-off frequency (0-2047) |
| **lp** | Low-pass filter on (1), off (0) |
| **bp** | Band-pass filter on (1), off (0) |
| **hp** | High-pass filter on (1), off (0) |
| **res** | Resonance (0-15) |

Unspecified parameters result in no change to the current value.

**EXAMPLES:**

| | |
|---|---|
| FILTER 1024,0,1,0,2 | Set the cutoff frequency at 1024, select the band pass filter and a resonance level of 2. |
| FILTER 2000,1,0,1,10 | Set the cutoff frequency at 2000, select both the low pass and high pass filters (to form a notch reject) and set the resonance level at 10. |

# FOR / TO / STEP / NEXT

Define a repetitive program loop structure.

**FOR variable = start value TO end value [STEP increment] NEXT variable**

The logic of the FOR/NEXT statement is as follows. First, the loop variable is set to the start value. When the program reaches a program line containing the NEXT statement, it adds the STEP increment (default = 1) to the value of the loop variable and checks to see if it is higher than the end value of the loop. If the loop variable is less than or equal to the end value, the loop is executed again, starting with the statement immediately following the FOR statement. If the loop variable is greater than the end value, the loop terminates and the program resumes immediately following the NEXT statement. The opposite is true if the step size is negative. See also the NEXT statement.

**EXAMPLE:**

```
10 FOR L = 1 TO 10
20 PRINT L
30 NEXT L
40 PRINT "I'M DONE! L = "L
```

This program prints the numbers from one to 10 followed by the message I'M DONE! L = 11.

**EXAMPLE:**

```
10 FOR L = 1 TO 100
20 FOR A = 5 TO 11 STEP .5
30 NEXT A
40 NEXT L
```

The FOR . . . NEXT loop in lines 20 and 30 are nested inside the one in line 10 and 40. Using a STEP increment of .5 is used to illustrate the fact that floating point indices are valid. The inner rested loop must lie completely within the outer rested loop (lines 10 and 40).

# GET

Receive input data from the keyboard, one character at a time, without waiting for a key to be pressed.

**GET variable list**

**EXAMPLE:**

```
10 DO:GETA$:LOOP UNTIL A$="A"
```

This line waits for the A key to be pressed to continue.

20 GET B, C, D GET numeric variables B,C and D from the keyboard without waiting for a key to be pressed.

## GETKEY

Receive input data from the keyboard, one character at a time and wait for a key to be pressed.

**GETKEY variable list**

**EXAMPLE:**

10 GETKEY A$

This line waits for a key to be pressed. Typing any key continues the program.

10 GETKEY A$,B$,C$

This line waits for three alphanumeric characters to be entered from the keyboard.

## GET#

Receive input data from a tape, disk or RS232

**GET# logical file number, variable list**

**EXAMPLE:**

10 GET#1,A$ This example receives one character, which is stored in the variable A$, from logical file number 1. This example assumes that file 1 was previously opened. See the OPEN statement.

## GO64

Switch to C64 mode

**GO64**

To return to C128 mode, press the reset button, or turn off the computer power and turn it on again.

## GOSUB

Call a subroutine from the specified line number

**GOSUB line number**

**EXAMPLE:**

20 GOSUB 800 This example calls the subroutine beginning at line 800 and executes it. All subroutines must terminate with a RETURN statement.

800 PRINT "THE C128 WAS WORTH THE WAIT!": RETURN

## GOTO / GO TO

Transfer program execution to the specified line number

**GOTO line number**

**EXAMPLES:**

| | |
|---|---|
| 10 PRINT"COMMODORE"<br>20 GOTO 10 | The GOTO in line 20 makes line 10 repeat continuously until RUN/STOP is pressed. |
| GOTO 100 | Starts (RUNs) the program starting at line 100, without clearing the variable storage area. |

## GRAPHIC

Select a graphic mode

1) **GRAPHIC mode [,clear][,s]**
2) **GRAPHIC CLR**

This statement puts the Commodore 128 in one of the six graphic modes:

| MODE | DESCRIPTION |
|---|---|
| **0** | **40-column text (default)** |
| **1** | **standard bit-map graphics** |
| **2** | **standard bit-map graphics (split screen)** |
| **3** | **multi-color bit-map graphics** |
| **4** | **multi-color bit-map graphics (split screen)** |
| **5** | **80-column text** |

**EXAMPLES:**

| | |
|---|---|
| GRAPHIC 1,1 | Select standard bit map mode and clear the bit map. |
| GRAPHIC 4,0,10 | Select split screen multi-color bit map mode, do not clear the bit map and start the split screen at line 10. |
| GRAPHIC 0 | Select 40-column text. |
| GRAPHIC 5 | Select 80-column text. |
| GRAPHIC CLR | Clear and deallocate the bit map screen. |

## GSHAPE

**See SSHAPE.**

# HEADER

Format a diskette

**HEADER "diskname" [,I i.d.] [,Ddrive number]**
**[<ON|,>Udevice number]**

Before a new disk can be used for the first time, it must be formatted with the HEADER command. The HEADER command can also be used to erase a previously formatted disk, which can then be reused.

When you enter a HEADER command in direct mode, the prompt **ARE YOU SURE?** appears. In program mode, the prompt does not appear.

The HEADER command is analogous to the BASIC 2.0 command:

**OPEN 1,8,15,"N0:diskname,i.d."**

**EXAMPLES:**

| | |
|---|---|
| HEADER "MYDISK",I23, D0 | This headers "MYDISK" using i.d. 23 on drive 0, (default) device number 8. |
| HEADER "RECS", I45, D1 ON U9 | This headers "RECS" using i.d. 45, on drive 1, device number 9. |
| HEADER "C128 PROGRAMS", D0 | This is a quick header on drive 0, device number 8, assuming the disk in the drive was already formatted. The old i.d. is used. |
| HEADER (A$),I76,D0,U9 | This example headers the diskette with the name specified by the variable A$, and the i.d. 76 on drive 0, device number 9. |

# HELP

Highlight the line where the error occurred

**HELP**

The HELP command is used after an error has been reported in a program. When HELP is typed in 40-column format, the line where the error occurs is listed, with the portion containing the error displayed in reverse field. In 80-column format, the portion of the line where the error occurs is underlined.

# IF / THEN / ELSE

Evaluate a conditional expression and execute portions of a program depending on the outcome of the expression

**IF expression THEN statements [:ELSE else-clause]**

THE IF . . . THEN statement evaluates a BASIC expression and takes one of two possible courses of action depending upon the outcome of the expression. If the expression is true, the statement(s) following THEN is executed. This can be any BASIC statement or a line number. If the expression is false, the program resumes with the program line immediately following the program line containing the IF statement, unless an ELSE clause is present. The entire IF . . . THEN statement must be contained within 160 characters. Also see BEGIN/BEND.

The ELSE clause, if present, must be on the same line as the IF . . . THEN portion of the statement, and separated from the THEN clause by a colon. When an ELSE clause is present, it is executed only when the expression is false. The expression being evaluated may be a variable or formula, in which case it is considered true if nonzero, and false if zero. In most cases, there is an expression involving relational operators ( =, <,>, <=,>=, <>).

**EXAMPLE:**

```
50 IF X > 0 THEN PRINT "OK": ELSE END
```

This line checks the value of X. If X is greater than 0, the statement immediately following the keyword THEN (PRINT "OK") is executed and the ELSE clause is ignored. If X is less than or equal to 0, the ELSE clause is executed and the statement immediately following THEN is ignored.

```
10 IF X = 10 THEN 100
20 PRINT "X DOES NOT EQUAL 10"
:
99 STOP
100 PRINT "X EQUALS 10"
```

This example evaluates the value of X. IF X equals 10, the program control is transferred to line 100 and the message "X EQUALS 10" is printed. IF X does not equal 10, the program resumes with line 20, the C128 prints the prompt "X DOES NOT EQUAL 10" and the program stops.

# INPUT

Receive a data string or a number from the keyboard and wait for the user to press RETURN

**INPUT ["prompt string";] variable list**

**EXAMPLE:**

```
10 INPUT "PLEASE TYPE A NUMBER";A
20 INPUT "AND YOUR NAME";A$
30 PRINT A$ " YOU TYPED THE NUMBER";A
```

# INPUT #

Input data from an I/O channel into a string or numeric variable

**INPUT# file number, variable list**

**EXAMPLE:**

```
10 OPEN 2,8,2
20 INPUT#2, A$, C, D$
```

This statement INPUTs the data stored in variables A$, C and D$ from the disk file number 2, which was OPENed in line 10.

# KEY

Define or list function key assignments

**KEY [key number, string]**

The maximum length for all the definitions together is 241 characters. (p. 3-41)

**EXAMPLE:**

```
KEY 7, "GRAPHIC0" + CHR$(13) + "LIST" + CHR$(13)
```

This tells the computer to select the (VIC) text screen and list the program whenever the F7 key is pressed (in direct mode). CHR$(13) is the ASCII character for RETURN and performs the same action as pressing the RETURN key. Use CHR$(27) for ESCape. Use CHR$(34) to incorporate the double quote character into a KEY string. The keys may be redefined in a program. For example:

```
10 KEY 2,"PRINT DS$" + CHR$(13)
```

This tells the computer to check and display the disk drive error channel variables (PRINT DS$) each time the F2 function key is pressed.

```
10 FOR I=1 to 7 STEP 2
20 KEY I, CHR$(I + 132):NEXT
30 FOR I=2 to 8 STEP 2
40 KEY I, CHR$(I + 132):NEXT
```

This defines the function keys as they are defined on the Commodore 64.

# LET

Assigns a value to a variable

**[LET] variable = expression**

**EXAMPLE:**

```
10 LET A = 5        Assign the value 5 to numeric variable A.
```

| | |
|---|---|
| 20 B = 6 | Assign the value 6 to numeric variable B. |
| 30 C = A * B + 3 | Assign the numeric variable C, the value resulting from 5 times 6 plus 3. |
| 40 D$ = ''HELLO'' | Assign the string ''HELLO'' to string variable D$. |

## LIST

List the BASIC program currently in memory

**LIST [first line] [- last line]**

In C128 mode, LIST can be used within a program without terminating program execution.

### EXAMPLES:

| | |
|---|---|
| LIST | Shows entire program. |
| LIST 100– | Shows from line 100 until the end of the program. |
| LIST 10 | Shows only line 10. |
| LIST –100 | Shows all lines from the beginning through line 100. |
| LIST 10–200 | Shows lines from 10 to 200, inclusive. |

## LOAD

Load a program from a peripheral device such as the disk drive or Datassette

**LOAD ''filename'' [,device number] [,relocate flag]**

This is the command used to recall a program stored on disk or cassette tape. Here, the filename is a program name up to 16 characters long, in quotes. The name must be followed by a comma (outside the quotes) and a number which acts as a device number to determine where the program is stored (disk or tape). If no number is supplied, the Commodore 128 assumes device number 1 (the Datassette tape recorder).

### EXAMPLES:

| | |
|---|---|
| LOAD | Reads in the next program from tape. |
| LOAD ''HELLO'' | Searches tape for a program called HELLO, and LOADs it if found. |
| LOAD (A$),8 | LOADs the program from disk whose name is stored in the variable A$. |
| LOAD''HELLO'',8 | Looks for the program called HELLO on disk drive number 8, drive 0. (This is equivalent to DLOAD ''HELLO''). |
| LOAD''MACHLANG'',8,1 | LOADs the machine language program called ''MACHLANG'' into the location from which it was SAVEd. |

# LOCATE

Position the bit map pixel cursor on the screen

**LOCATE X,Y**

The LOCATE statement places the pixel cursor (PC) at any specified pixel coordinate on the screen.

The pixel cursor (PC) is the coordinate on the bit map screen where drawing of circles, boxes, lines and points and where PAINTing begins.

**EXAMPLE:**

| | |
|---|---|
| LOCATE 160,100 | Positions the PC in the center of the bit map screen. Nothing will be seen until something is drawn. |
| LOCATE +20,100 | Move the pixel cursor 20 pixels to the right of the last PC position and place it at Y coordinate 100. |
| LOCATE –30, +20 | Move the PC 30 pixels to the right and 20 down from the previous PC position. |

The PC can be found by using the RDOT(0) function to get the X-coordinate and RDOT(1) to get the Y-coordinate. The color source of the pixel at the PC can be found by PRINTing RDOT(2).

# MONITOR

Enter the Commodore 128 machine language monitor

**MONITOR**

See Chapter 6 for details on the Commodore 128 Machine Language Monitor.

# MOVSPR

Position or move sprite on the screen

| | |
|---|---|
| 1) **MOVSPR number,X,Y** | Place the specified sprite at absolute sprite coordinate X,Y. |
| 2) **MOVSPR number, +/–X, +/–Y** | Move sprite relative to the position of the sprite's current position. |
| 3) **MOVSPR number,X;Y** | Move sprite distance X at angle Y relative to the sprite's current position. |

4) **MOVSPR number, angle # speed** — Move sprite at an angle relative to its current coordinate, in the clockwise direction and at the specified speed.

where:

**number** is sprite's number (1 through 8)
**X,Y** is coordinate of the sprite location.

**angle** is the angle (0-360) of motion in the clockwise direction relative to the sprite's original coordinate.

**speed** is a speed (0-15) in which the sprite moves.

This statement moves a sprite to a specific location on the screen according to the SPRITE coordinate plane (not the bit map plane) or initiates sprite motion at a specified rate. See MOVSPR in Chapter 9 for a diagram of the sprite coordinate system.

**EXAMPLES:**

| | |
|---|---|
| MOVSPR 1,150,150 | Position sprite 1 near the center of the screen, x,y coordinate 150,150. |
| MOVSPR 1, +20,–30 | Move sprite 1 to the right 20 coordinates and up 30 coordinates. |
| MOVSPR 4, –50, +100 | Move sprite 4 to the left 50 coordinates and down 100 coordinates. |
| MOVSPR 5, 45 #15 | Move sprite 5 at a 45 degree angle in the clockwise direction, relative to its original x and y coordinates. The sprite moves at the fastest rate (15). |

**NOTE:** Once you specify an angle and a speed as in the fourth example of the MOVSPR statement, the sprite continues on its path (even if the sprite display is disabled) after the program stops, until you set the speed to 0 or press RUN/STOP and RESTORE. Also, keep in mind that the SCALE command affects the MOVSPR coordinates. If you add SCALing to your programs, you also must adjust the sprites' new coordinates so they appear correctly on the screen.

## NEW

Clear (erase) BASIC program and variable storage

**NEW**

# ON

Conditionally branch to a specified program line number according to the results of the specified expression

**ON expression <GOTO/GOSUB> line #1 [, line #2, . . .]**

**EXAMPLE:**

```
10 INPUT X:IF X<0 THEN 10
20 ON X GOTO 30, 40, 50, 60
25 STOP
30 PRINT "X = 1"
40 PRINT "X = 2"
50 PRINT "X = 3"
60 PRINT "X = 4"
```

When X = 1, ON sends control to the first line number in the list (30). When X = 2, ON sends control to the second line (40), etc.

# OPEN

Open files for input or output

**OPEN logical file number, device number [,secondary address] [<,"filename [,filetype[, [mode"]]|<,cmd string>]**

**EXAMPLES:**

| | |
|---|---|
| 10 OPEN 3,3 | OPEN the screen as file number 3. |
| 20 OPEN 1,0 | OPEN the keyboard as file number 1. |
| 30 OPEN 1,1,0,"DOT" | OPEN the cassette for reading, as file number 1, using "DOT" as the filename. |
| OPEN 4,4 | OPEN the printer as file number 4. |
| OPEN 15,8,15 | OPEN the command channel on the disk as file 15, with secondary address 15. Secondary address 15 is reserved for the disk drive error channel. |
| 5 OPEN 8,8,12,"TESTFILE,S,W" | OPEN a sequential disk file for writing called TESTFILE as file number 8, with secondary address 12. |

See also: CLOSE, CMD, GET#, INPUT#, and PRINT# statements and system variables ST, DS, and DS$.

# PAINT

Fill area with color

**PAINT [color source],X,Y[,mode]**

where:

**color source** 0 = bit map foreground
1 = bit map background (default)
2 = multi-color 1
3 = multi-color 2

**X,Y** starting coordinate, scaled (default at pixel cursor (PC))

**mode** 0 = paint an area defined by the color source selected
1 = paint an area defined by any nonbackground source

The PAINT command fills an area with color. It fills in the area around the specified point until a boundary of the same specified color source is encountered. For example, if you draw a circle in the foreground color source, start PAINTing the circle where the coordinate assumes the background source. The Commodore 128 will only PAINT where the specified source in the PAINT statement is different from the source of the x and y pixel coordinate. It cannot PAINT points where the sources are the same in the PAINT statement and the specified coordinate. The X and Y coordinate must lie completely within the boundary of the shape you intend to PAINT, and the source of the starting pixel coordinate and the specified color source must be different.

### EXAMPLE:

| | |
|---|---|
| 10 CIRCLE 1, 160,100,65,50 | Draws an outline of a circle. |
| 20 PAINT 1, 160,100 | Fills in the circle with color from source 1 (VIC foreground), assuming point 160,100 is colored in the background color (source 0). |
| 10 BOX 1, 10, 10, 20, 20 | Draws an outline of a box. |
| 20 PAINT 1, 15, 15 | Fills the box with color from source 1, assuming point 15,15 is colored in the background source (0). |
| 30 PAINT 1, + 10, + 10 | PAINT the screen in the foreground color source at the coordinate relative to the pixel cursor's previous position plus 10 in both the vertical and horizontal positions. |

| | |
|---|---|
| 100 PAINT 1, 100;90 | Paint the screen area 90° relative to and 100 pixels away from the current pixel cursor co-ordinate. |

## PLAY

Defines and plays musical notes and elements within a string or string variable.

**PLAY "Vn,On,Tn,Un,Xn,elements, notes"**

where the string or string variable is composed of the following

**Vn** = Voice (n = 1-3)
**On** = Octave (n = 0-6)
**Tn** = Tune Envelope Defaults (n = 0-9)
- 0 = piano
- 1 = accordion
- 2 = calliope
- 3 = drum
- 4 = flute
- 5 = guitar
- 6 = harpsichord
- 7 = organ
- 8 = trumpet
- 9 = xylophone

**Un** = Volume (n = 0-8)
**Xn** = Filter on (n = 1), off(n = 0)
**notes:** A,B,C,D,E,F,G
**elements:**

| | |
|---|---|
| **#** | Sharp |
| **$** | Flat |
| **W** | Whole note |
| **H** | Half note |
| **Q** | Quarter note |
| **I** | Eighth note |
| **S** | Sixteenth note |
| **.** | Dotted |
| **R** | Rest |
| **M** | Wait for all voices currently playing to end the current "measure" |

The PLAY statement gives you the power to select voice, octave and tune envelope (including ten predefined musical instrument envelopes), the volume, the filter, and the notes you want to PLAY. All these controls are enclosed in quotes. You may include spaces in a PLAY string for readability.

All elements except R and M precede the musical notes in a PLAY string.

**EXAMPLES:**

| | |
|---|---|
| PLAY "V1O4T0U5X0CDEFGAB" | Play the notes C,D,E,F,G,A and B in voice 1, octave 4, tune envelope 0 (piano), at volume 5, with the filter off. |
| PLAY "V3O5T6U7X1#B$AW.CHDQEIF" | Play the notes B-sharp, A-flat, a whole dotted-C note, a half D-note, a quarter E-note and an eighth F-note. |
| A$ = "V3O5T6U3ABCDE": PLAY A$ | PLAY the notes and elements within A$. |
| PLAY "V1CV2EV3G" | Plays a chord in the default setting. |

# POKE

Change the contents of a RAM memory location

**POKE address, value**

**EXAMPLE:**

10 POKE 53280,1 Changes VIC border color

# PRINT

Output to the text screen

**PRINT [print list]**

The word PRINT can be followed by any of the following:

| | |
|---|---|
| **Characters inside quotes** | ("text") |
| **Variable names** | (A, B, A$, X$) |
| **Functions** | (SIN(23), ABS(33)) |
| **Expressions** | (2+2),A+3,A=B) |
| **Punctuation marks** | (;,) |

**EXAMPLES:**

| | **RESULTS** |
|---|---|
| 10 PRINT "HELLO" | HELLO |
| 20 A$ ="THERE":PRINT "HELLO";A$ | HELLO THERE |
| 30 A = 4:B = 2:?A + B | 6 |
| 40 J = 41:PRINT J;:PRINT J – 1 | 41 40 |
| 50 PRINT A;B;:D = A + B:PRINT D;A-B | 4 2 6 2 |

See also POS, SPC, TAB and CHAR functions.

# PRINT#

Output data to files

**PRINT# file number[, print list]**

PRINT# is followed by a number which refers to the data file previously OPENed.

**EXAMPLE:**

| | |
|---|---|
| 10 OPEN 4,4<br>20 PRINT#4,"HELLO THERE!",A$,B$ | Outputs the data "HELLO THERE" and the variables A$ and B$ to the printer. |
| 10 OPEN 2,8,2<br>20 PRINT#2,A,B$,C,D | Outputs the data variables A, B$, C and D to the disk file number 2. |

**NOTE:** The PRINT# command is used by itself to close the channel to the printer before closing the file, as follows:

```
10 OPEN 4,4
30 PRINT#4, "PRINT WORDS"
40 PRINT#4
50 CLOSE 4
```

# PRINT USING

Output using format

**PRINT [#file number,] USING"format list"; print list**

This statement defines the format of string and numeric items for printing to the text screen, printer or other device.

**EXAMPLE:**

```
5 X = 32: Y = 100.23: A$ = "CAT"
10 PRINT USING "$##.### ";13.25,X,Y
20 PRINT USING "###>#";"CBM",A$
```

When this is RUN, line 10 prints:

| | |
|---|---|
| $13.25 $32.00 $***** | Five asterisks (*****) are printed instead of a Y value because Y has five digits, and this condition does not conform to format list (as explained below). |

Line 20 prints this:

| | |
|---|---|
| CBM CAT | Leaves two spaces before printing "CBM" as defined in format list. |

The pound sign (#) reserves room for a single character in the output field. If the data item contains more characters than there are # signs in the format field, the entire field is filled with asterisks (*): no characters are printed.

**EXAMPLE:**

```
10 PRINT USING ''####'';X
```

For these values of X, this format displays:

```
A = 12.34      12
A = 567.89     568
A = 123456     ****
```

For a STRING item, the string data is truncated at the bounds of the field. Only as many characters are printed as there are pound signs (#) in the format item. Truncation occurs on the right.

**EXAMPLES:**

| FIELD | EXPRESSION | RESULT | COMMENT |
|---|---|---|---|
| ##.# | –.1 | –0.1 | **Leading zero added.** |
| ##.# | 1 | 1.0 | **Trailing zero added.** |
| #### | –100.5 | –101 | **Rounded to no decimal places.** |
| #### | –1000 | **** | **Overflow because four digits and a minus sign cannot fit in field.** |
| ###. | 10 | 10. | **Decimal point added.** |
| #$## | 1 | $1 | **Floating dollar sign.** |

# PUDEF

Redefine symbols in PRINT USING statement

**PUDEF ''nnnn''**

Where ''nnnn'' is any combination of characters, up to four in all. PUDEF allows you to redefine any of the following four symbols in the PRINT USING statement: blanks, commas, decimal points and dollar signs. These four symbols can be changed into some other character by placing the new character in the correct position in the PUDEF control string.

Position 1 is the filler character. The default is a blank. Place a new character here for another character to appear in place of blanks.

Position 2 is the comma character. Default is a comma.

Position 3 is the decimal point. Default is a decimal point.

Position 4 is the dollar sign. Default is a dollar sign.

**EXAMPLES:**

```
10 PUDEF"*"        PRINT * in the place of blanks.
20 PUDEF"<"        PRINT < in the place of commas.
```

# READ

Read data from DATA statements and input it into a numeric or string variable)

**READ variable list**

This statement inputs information from DATA statements and stores it in variables, where the data can be used by the RUNning program.

In a program, you can READ the data and then re-read it by issuing the RESTORE statement. The RESTORE sets the sequential data pointer back to the beginning, where the data can be read again. See the RESTORE and DATA statements.

**EXAMPLES:**

```
10 READ A, B, C                  READ the first three numeric variables from
20 DATA 3, 4, 5                  the closest data statement.

10 READ A$, B$, C$               READ the first three string variables from
20 DATA JOHN, PAUL, GEORGE       the nearest data statement.

10 READ A, B$, C                 READ (and input into the C128 memory) a
20 DATA 1200, NANCY, 345         numeric variable, a string variable and an-
                                 other numeric variable.
```

# RECORD

Position relative file pointers

**RECORD# logical file number, record number [,byte number]**

This statement positions a relative file pointer to select any byte (character) of any record in the relative file.

When the record number value is set higher than the last record number in the file, the following occurs:

For a write (PRINT#) operation, additional records are created to expand the file to the desired record number.

For a read (INPUT#) operation, a null record is returned and a "RECORD NOT PRESENT ERROR occurs". See your disk drive manual for details about relative files.

**EXAMPLES:**

```
10 DOPEN#2,"FILE"
20 RECORD#2,10,1
30 PRINT#2,A$
40 DCLOSE#2
```

This example opens an existing relative file called "FILE" as file number 2 in line 10. Line 20 positions the relative file pointer at the first byte in record number 10. Line 30 actually writes the data, A$, to file number 2.

## REM

Comments or remarks about the operation of a program line

**REM message**

### EXAMPLE:

10 NEXT X:REM THIS LINE INCREMENTS X.

## RENAME

Change the name of a file on disk

**RENAME "old filename" TO "new filename" [,Ddrive number][<ON|,> Udevice number]**

### EXAMPLES:

| | |
|---|---|
| RENAME "TEST" TO "FINALTEST",D0 | Change the name of the file "TEST" to "FINAL TEST". |
| RENAME (A$) TO (B$),D0,U9 | Change the filename specified in A$ to the filename specified in B$ on drive 0, device number 9. Remember, whenever a variable name is used as a filename, it must be enclosed in parentheses. |

## RENUMBER

Renumber lines of a BASIC program

**RENUMBER [new starting line number][,increment][,old starting line number]**

### EXAMPLES:

| | |
|---|---|
| RENUMBER | Renumbers the program starting at 10, and increments each additional line by 10. |
| RENUMBER 20, 20, 1 | Starting at line 1, renumbers the program. Line 1 becomes line 20, and other lines are numbered in increments of 20. |

RENUMBER,, 65 Starting at line 65, renumbers in increments of 10. Line 65 becomes line 10. If you omit a parameter, you must still enter a comma in its place.

## RESTORE

Reset READ pointer so the DATA can be reREAD

**RESTORE [line#]**

If a line number follows the RESTORE statement, the READ pointer is set to the first data item in the specified program line. Otherwise the pointer is reset to the beginning of the first DATA statement in the BASIC program.

### EXAMPLES:

```
10 FOR I = 1 TO 3
20 READ X
30 ALL = X + ALL
40 NEXT
50 RESTORE
60 GOTO 10
70 DATA 10,20,30
```

This example READs the data in line 70 and stores it in numeric variable X. It adds the total of all the numeric data items. Once all the data has been READ, three cycles through the loop, the READ pointer is RESTOREd to the beginning of the program and it returns to line 10 and performs repetitively.

```
10 READ A,B,C
20 DATA 100,500,750
30 READ X,Y,Z
40 DATA 36,24,38
50 RESTORE 40
60 READ S,P,Q
```

Line 50 of this example RESTORES the DATA pointer to the beginning data item in line 40. When line 60 is executed, it will READ the DATA 36,24,38 from line 40, and store it in numeric variables S, P, and Q, respectively.

## RESUME

Define where the program will continue (RESUME) after an error has been trapped

**RESUME [line number | NEXT]**

This statement is used to restart program execution after TRAPping an error. With no parameters, RESUME attempts to re-execute the statement in which the error occurred. RESUME NEXT resumes execution at the statement immediately following the one indicating the error. RESUME followed by a line number will GOTO the specific line and resume execution from that line number. RESUME can only be used in program mode.

### EXAMPLE:

```
10 TRAP 100
15 INPUT " ENTER A NUMBER";A
20 B = 100/A
40 PRINT"THE RESULT =";B
```

```
50 INPUT ''DO YOU WANT TO RUN IT AGAIN (Y/N)'';Z$:IF Z$ = ''Y''
THEN 10
60 STOP
100 INPUT''ENTER ANOTHER NUMBER (NOT ZERO)'';A
110 RESUME 20
```

This example traps a ''DIVISION BY ZERO ERROR'' in line 20 if 0 is entered in line 15. If zero is entered, the program goes to line 100, where you are asked to input another number besides 0. Line 110 returns to line 20 to complete the calculation. Line 50 asks if you want to repeat the program again. If you do, press the Y key.

## RETURN

Return from subroutine

**RETURN**

### EXAMPLE:

```
10 PRINT ''ENTER MAIN PROGRAM''
20 GOSUB 100
30 PRINT ''END OF PROGRAM''
.
.
.
90 STOP
100 PRINT ''SUBROUTINE 1''
110 RETURN
```

This example calls the subroutine at line 100 which prints the message ''SUBROUTINE 1'' and RETURNs to line 30, the rest of the program.

## RUN

Execute BASIC program

1) **RUN [line number]**
2) **RUN ''filename'' [,Ddrive number][<ON|,>Udevice number]**

### EXAMPLES:

| | |
|---|---|
| RUN | Starts execution from the beginning of the program. |
| RUN 100 | Starts program execution at line 100. |
| RUN''PRG1'' | DLOADs ''PRG1'' from disk drive 8, and runs it from the starting line number. |
| RUN(A$) | DLOADs the program named in the variable A$. |

# SAVE

Store the program in memory to disk or tape

**SAVE ["filename"][,device number][,EOT flag]**

**EXAMPLES:**

| | |
|---|---|
| SAVE | Stores program on tape, without a name. |
| SAVE "HELLO" | Stores a program on tape, under the name HELLO. |
| SAVE A$,8 | Stores on disk, with the name stored in variable A$. |
| SAVE "HELLO",8 | Stores on disk, with name HELLO (equivalent to DSAVE "HELLO"). |
| SAVE "HELLO", 1, 2 | Stores on tape, with name HELLO, and places an END OF TAPE marker after the program. |

# SCALE

Alter scaling in graphics mode

**SCALE n [,Xmax,Ymax]**

where:

**n** = 1 (on) or 0 (off)

Coordinates may be scaled from 0 to 32767 (default = 1023) in both X and Y (in either standard or multicolor bit map mode), rather than the normal scale values, which are:

| | | |
|---|---|---|
| **multi-color mode** | X = 0 to 159 | Y = 0 to 199 |
| **bit map mode** | X = 0 to 319 | Y = 0 to 199 |

**EXAMPLES:**

```
10 GRAPHIC 1,1
20 SCALE 1:CIRCLE 1,180,100,100,100
```

Enter standard bit map, turn scaling on to default size (1023, 1023) and draw a circle.

```
10 GRAPHIC 1,3
20 SCALE 1,1000,5000
30 CIRCLE 1,180,100,100,100
```

Enter multi-color mode, turn scaling on to size (1000, 5000) and draw a circle.

The SCALE command affects the sprite coordinates in the MOVSPR command. If you add scaling to a program that contains sprites, adjust the MOVSPR coordinates accordingly.

## SCNCLR

Clear screen

**SCNCLR mode number**

The modes are as follows:

| MODE NUMBER | MODE |
|---|---|
| **0** | **40 column (VIC) text** |
| **1** | **bit map** |
| **2** | **split screen bit map** |
| **3** | **multi-color bit map** |
| **4** | **split screen multi-color bit map** |
| **5** | **80 column (8563) text** |

This statement with no argument clears the graphic screen, if it is present, otherwise the current text screen is cleared.

### EXAMPLES:

SCNCLR 5 Clears 80 column text screen.
SCNCLR 1 Clears the (VIC) bit map screen.
SCNCLR 4 Clears the (VIC) multicolor bit map and 40-column text split screen.

## SCRATCH

Delete file from the disk directory

**SCRATCH "filename" [,Ddrive number][<ON|,>Udevice number]**

### EXAMPLE:

SCRATCH "MY BACK", D0

This erases the file MY BACK from the disk in drive 0.

## SLEEP

Delay program for a specific period of time

**SLEEP N**

where N is seconds 0< N < = 65535.

# SLOW

Return the Commodore 128 to 1MHz operation

**SLOW**

# SOUND

Output sound effects and musical notes

**SOUND v,f,d[,dir][,m][,s][,w][,p]**

where:
- **v** = voice (1..3)
- **f** = frequency value (0..65535)
- **d** = duration (0..32767)
- **dir** = step direction (0(up), 1(down) or 2(oscillate)) default = 0
- **m** = minimum frequency (if sweep is used) (0..65535) default = 0
- **s** = step value for sweep (0..32767) default = 0
- **w** = waveform (0 = triangle, 1 = sawtooth, 2 = variable, 3 = noise) default = 2
- **p** = pulse width (0..4095) default = 2048

**EXAMPLES:**

| | |
|---|---|
| SOUND 1,40960,60 | Play a SOUND at frequency 40960 in voice 1 for 1 second. |
| SOUND 2,20000,50,0,2000,100 | Output a sound by sweeping through frequencies starting at 2000 and incrementing upward in units of 100 up to 20,000. Each frequency is played for 50 *jiffies*. |
| SOUND 3,5000,90,2,3000,500,1 | This example outputs a range of sounds starting at a minimum frequency of 3000, through 5000, in increments of 500. The direction of the sweep is back and forth (oscillating). The selected waveform is sawtooth and the voice selected is 3. |

# SPRCOLOR

Set multi-color 1 and/or multi-color 2 colors for all sprites

**SPRCOLOR [smcr-1][,smcr-2]**

where:

- **smcr-1** Sets multi-color 1 for all sprites.
- **smcr-2** Sets multi-color 2 for all sprites.

Either of these parameters may be any color from 1 through 16.

**EXAMPLES:**

SPRCOLOR 3,7 Sets sprite multi-color 1 to red and multi-color 2 to blue.

SPRCOLOR 1,2 Sets sprite multi-color 1 to black and multi-color 2 to white.

## SPRDEF

Enter the SPRite DEFinition mode to create and edit sprite images.

**SPRDEF**

The SPRDEF command defines sprites interactively

Entering the SPRDEF command displays a sprite work area on the screen which is 24 characters wide by 21 characters tall. Each character position in the grid corresponds to a sprite pixel in the sprite displayed to the right of the work area. Here is a summary of the SPRite DEFinition mode operations and the keys that perform them:

| USER INPUT | DESCRIPTION |
|---|---|
| **1-8** | **Selects a sprite number at the SPRITE NUMBER? prompt only.** |
| **A** | **Turns on and off automatic cursor movement.** |
| **CRSR keys** | **Moves cursor in work/area.** |
| **RETURN KEY** | **Moves cursor to start of next line.** |
| **RETURN key** | **Exits sprite designer mode at the SPRITE NUMBER? prompt only.** |
| **HOME key** | **Moves cursor to top left corner of sprite work area.** |
| **CLR key** | **Erases entire grid.** |
| **1-4** | **Selects color source (enables/disables pixels).** |
| **CTRL key, 1-8** | **Selects sprite foreground color (1-8).** |
| **Commodore key, 1-8** | **Selects sprite foreground color (9-16).** |
| **STOP key** | **Cancels changes and returns to prompt.** |
| **SHIFT RETURN** | **Saves sprite and returns to SPRITE NUMBER? prompt.** |
| **X** | **Expands sprite in X (horizontal) direction.** |
| **Y** | **Expands sprite in Y (vertical) direction.** |
| **M** | **Multi-color sprite mode.** |
| **C** | **Copies sprite data from one sprite to another.** |

## SPRITE

Turn on and off, color, expand and set screen priorities for a sprite

**SPRITE <number> [,on/off][,fgnd][,priority][,x-exp] [,y-exp][,mode]**

The SPRITE statement controls most of the characteristics of a sprite.

| PARAMETER | DESCRIPTION |
|---|---|
| **number** | **Sprite number (1-8)** |
| **on/off** | **Turn sprite on (1) or off (0)** |
| **foreground** | **Sprite foreground color (1-16) (default = sprite number)** |
| **priority** | **Priority is 0 if sprites appear in front of objects on the screen. Priority is 1 if sprites appear in back of objects on the screen.** |
| **x-exp** | **Horizontal EXPansion on (1) or off (0)** |
| **y-exp** | **Vertical EXPansion on (1) or off (0)** |
| **mode** | **Select standard sprite (0) or multi-color sprite (1)** |

Unspecified parameters in subsequent sprite statements take on the characteristics of the previous SPRITE statement. You may check the characteristics of a SPRITE with the RSPRITE function.

### EXAMPLES:

SPRITE 1,1,3 Turn on sprite number 1 and color it red.

SPRITE 2,1,7,1,1,1 Turn on sprite number 2, color it blue, make it pass behind objects on the screen and expand it in the vertical and horizontal directions.

SPRITE 6,1,1,0,0,1,1 Turn on SPRITE number 6, color it black. The first 0 tells the computer to display the sprites in front of objects on the screen. The second 0 and the 1 following tell the C128 to expand the sprite vertically only. The last 1 specifies multi-color mode. Use the SPRCOLOR command to select the sprite's multi-colors.

## SPRSAV

Copy sprite data from a text string variable into a sprite or vice versa, or copy data from one sprite to another.

**SPRSAV <origin>,<destination>**

Either the origin or the destination can be a sprite number or a string variable but they both cannot be string variables. They can both be sprite numbers. If you are storing a string into a sprite, only the first 63 bytes of data are used. The rest are ignored since a sprite can only hold 63 data bytes.

### EXAMPLES:

SPRSAV 1,A$ Transfers the image (data) from sprite 1 to the string named A$.

SPRSAV B$,2 Transfers the data from string variable B$ into sprite 2.

SPRSAV 2,3 Transfers the data from sprite 2 to sprite 3.

# SSHAPE / GSHAPE

Save/retrieve shapes to/from string variables

SSHAPE and GSHAPE are used to save and load rectangular areas of bit map screens to/from BASIC string variables. The command to save an area of the bit map screen into a string variable is:

**SSHAPE string variable, X1, Y1 [,X2,Y2]**

where:

**string variable** String name to save data in
**X1,Y1** Corner coordinate (0,0 through 319,199) (scaled)
**X2,Y2** Corner coordinate opposite (X1,Y1) (default is the PC)

The command to retrieve (load) the data from a string variable and display it on specified screen coordinates is:

**GSHAPE string variable [X,Y][,mode]**

where:

**string** Contains shape to be drawn
**X,Y** Top left coordinate (0,0 through 319,199) telling where to draw the shape (scaled—the default is the pixel cursor)
**mode** Replacement mode:
0 = place shape as is (default)
1 = invert shape
2 = OR shape with area
3 = AND shape with area
4 = XOR shape with area

The replacement mode allows you to change the data in the string variable so you can invert it, perform a logical OR, exclusive OR (turn off bytes that are on) or AND operation on the image.

## EXAMPLES:

| | |
|---|---|
| SSHAPE A$,10,10 | Saves a rectangular area from the coordinates 10,10 to the location of the pixel cursor, into string variable A$. |
| SSHAPE B$,20,30,43,50 | Saves a rectangular area from top left coordinate (20,30) through bottom right coordinate (43,50) into string variable B$. |
| SSHAPE D$, + 10, + 10 | Saves a rectangular area 10 pixels to the right and 10 pixels down from the current position of the pixel cursor. |

| | |
|---|---|
| GSHAPE A$,120,20 | Retrieves shape contained in string variable A$ and displays it at top left coordinate (120,20). |
| GSHAPE B$,30,30,1 | Retrieves shape contained in string variable B$ and displays it at top left coordinate 30,30. The shape is inverted due to the replacement mode being selected by the 1. |
| GSHAPE C$, + 20, + 30 | Retrieves shape from string variable C$ and displays it 20 pixels to the right and 30 pixels down from the current position of the pixel cursor. |

**NOTE:** Beware using modes 1-4 with multi-color shapes. You may obtain unpredictable results.

## STASH

Move contents of host memory to expansion RAM

**STASH #bytes, intsa, expsa, expb**

Refer to FETCH command for description of parameters.

## STOP

Halt program execution

**STOP**

## SWAP

Swap contents of host RAM with contents of expansion RAM

**SWAP #bytes, intsa, expsa, expb**

Refer to FETCH command for description of parameters.

## SYS

Call and execute a machine language subroutine at the specified address

**SYS address [,a][,x][,y][,s]**

This statement calls a subroutine at a given address in a memory configuration previously set up according to the BANK command. Optionally, arguments a,x,y and s are loaded into the accumulator, x, y and status registers, respectively, before the subroutine is called.

The address range is 0 to 65535. The 8502 microprocessor begins executing the machine-language program starting at the specified memory location. Also see the BANK command.

**EXAMPLES:**

| | |
|---|---|
| SYS 32768 | Calls and executes the machine-language routine at location 32768 ($8000). |
| SYS 6144,0 | Calls and executes the machine-language routine at location 6144 ($1800) and loads zero into the accumulator. |

## TEMPO

Define the speed of the song being played

**TEMPO n**

where n is a relative duration between (1 and 255)
The default value is 8, and note duration increases with n.

**EXAMPLES:**

| | |
|---|---|
| TEMPO 16 | Defines the Tempo at 16. |
| TEMPO 1 | Defines the TEMPO at the slowest speed. |
| TEMPO 250 | Defines the TEMPO at 250. |

## TRAP

Detect and correct program errors while a BASIC program is RUNning

**TRAP [line number]**

The RESUME statement can be used to resume program execution. TRAP with no line number turns off error trapping. An error in a TRAP routine cannot be trapped. Also see system variables ST, EL, DS and DS$.

**EXAMPLES:**

| | |
|---|---|
| .100 TRAP 1000 | If an error occurs, GOTO line 1000. |
| ⋮ | |
| .1000 ?ERR$(ER);EL | Print the error message, and the error number. |
| .1010 RESUME | Resume with program execution. |

## TROFF

Turn off error tracing mode

**TROFF**

# TRON

Turn on error tracing

**TRON**

TRON is used in program debugging. This statement begins trace mode. When you RUN the program, the line numbers of the program appear in brackets before any action for that line occurs.

# VERIFY

Verify program in memory against one saved to disk or tape

**VERIFY "filename" [,device number][,relocate flag]**

Issue the VERIFY command immediately after you SAVE a program.

**EXAMPLES:**

| | |
|---|---|
| VERIFY | Checks the next program on the tape. |
| VERIFY "HELLO" | Searches for HELLO on tape, checks it against memory. |
| VERIFY "HELLO",8,1 | Searches for HELLO on disk, then checks it against memory. |

# VOL

Define output level of sound for SOUND and PLAY statements

**VOL volume level**

**EXAMPLES:**

| | |
|---|---|
| VOL 0 | Sets volume to its lowest level. |
| VOL 15 | Sets volume for SOUND and PLAY statements to its highest output. |

# WAIT

Pause program execution until a data condition is satisfied

**WAIT <location>, <mask-1> [,mask-2>]**

where:

**location** = 0–65535
**masks** = 0–255

The WAIT statement causes program execution to be suspended until a given memory address recognizes a specified bit pattern or value.

The first example below WAITs until a key is pressed on the tape unit to continue with the program. The second example will WAIT until a sprite collides with the screen background.

**EXAMPLES:**

```
WAIT 1, 32, 32
WAIT 53273, 2
WAIT 36868, 144, 16
```

## WIDTH

Set the width of drawn lines

**WIDTH n**

**EXAMPLES:**

```
WIDTH 1   Set single width for graphic commands
WIDTH 2   Set double width for drawn lines
```

## WINDOW

Define a screen window

**WINDOW top left col,top left row,bot right col,bot right row[,clear]**

This command defines a logical window within the 40 or 80 column text screen. The coordinates must be in the range 0-39/79 for 40- and 80-column values respectively and 0-24 for row values. The clear flag, if provided (1), causes a screen-clear to be performed (but only within the limits of the newly described window).

**EXAMPLES:**

```
WINDOW 5,5,35,20     Defines a window with top left corner coordinate as
                     5,5 and bottom right corner coordinate as 35,20.

WINDOW 10,2,33,24,1  Defines a window with upper left corner coordinate
                     10,2 and lower right corner coordinate 33,24. Also
                     clears the portion of the screen within the window as
                     specified by the 1.
```

## BASIC FUNCTIONS

The format of the function description is:

**FUNCTION (argument)**

where the argument can be a numeric value, variable or string.

Each function description is followed by an EXAMPLE. The first line appearing below the word "**EXAMPLE**" is the function you type. The second line without bold is the computer's response.

## ABS

Return absolute value of argument X

**ABS (X)**

**EXAMPLE:**

PRINT ABS (7*(–5))

35

## ASC

Return CBM ASCII code for the first character in X$

**ASC(X$)**

This function returns the CBM ASCII code of the first character of X$.

**EXAMPLE:**

X$ = "C128":PRINT ASC (X$)

67

## ATN

Return the arctangent of X in radians

**ATN (X)**

The value returned is in the range $-\pi/2$ through $\pi/2$.

**EXAMPLE:**

PRINT ATN (3)

1.24904577

## BUMP

Return sprite collision information

**BUMP (N)**

To determine which sprites have collided since the last check, use the BUMP function. BUMP(1) records which sprites have collided with each other, and BUMP(2) records which sprites have collided with other objects on the screen. COLLISION need not be active to use BUMP. The bit positions (0-7) in the BUMP value correspond to sprites 1 through 8 respectively. BUMP(n) is reset to zero after each call.

Here's how the sprite numbers and BUMP values that are returned correspond:

| BUMP Value: | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|
| Sprite Number: | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |

**EXAMPLES:**

PRINT BUMP (1) 12 indicates that sprites 3 and 4 have collided.

PRINT BUMP (2) 32 indicates the sprite 6 has collided with an object on the screen.

## CHR$

Return character for specified CBM ASCII code X

CHR$(X)

The argument (X) must be in the range 0–255. This is the opposite of ASC and returns the string character whose CBM ASCII code is X. Refer to Appendix E for a table of CHR$ codes.

**EXAMPLES:**

PRINT CHR$ (65) Prints the A character.
A
PRINT CHR$ (147) Clears the text screen.

## COS

Return cosine for angle of X in radians

**COS(X)**

**EXAMPLE:**

PRINT COS (π/3)
.5

## FNxx

Return value from user defined function xx

**FNxx(X)**

This function returns the value from the user defined function xx created in a DEF FNxx statement

**EXAMPLE:**

```
10 DEF FNAA(X) = (X-32)*5/9
20 INPUT X
30 PRINT FNAA(X)
RUN
?40 (? is input prompt)
4.44444445
```

**NOTE:** If GRAPHIC is used in a program that defines a function, issue the GRAPHIC command before defining the function or the function definition is destroyed.

## FRE

Return number of available bytes in memory

**FRE (X)**

where X is the RAM bank number. X = 0 for BASIC program storage and X = 1 to check for available BASIC variable storage.

### EXAMPLES:

PRINT FRE (0) Returns the number of free bytes for BASIC programs.
58109

PRINT FRE (1) Returns the number of free bytes for BASIC variable storage.
64256

## HEX$

Return hexadecimal string equivalent to decimal number X

**HEX$(X)**

### EXAMPLE:

PRINT HEX$(53280)
D020

## INSTR

Return starting position of string 2 within string 1

**INSTR (string 1, string 2 [,starting position])**

### EXAMPLE:

PRINT INSTR (''COMMODORE 128'',''128'')
11

## INT

Return integer form (truncated) of a floating point value

**INT(X)**

This function returns the integer value of the expression. If the expression is positive, the fractional part is left out. If the expression is negative, any fraction causes the next lower integer to be returned.

**EXAMPLES:**

```
PRINT INT(3.14)
3

PRINT INT(–3.14)
–4
```

## JOY

Return position of joystick and the status of the fire button

**JOY(N)**

when N equals:

**1** JOY returns position of joystick 1.
**2** JOY returns postion of joystick 2.

Any value of 128 or more means that the fire button is also pressed. To find the joystick position if the fire button is pressed subtract 128 from the JOY value. The direction is indicated as follows.

```
          1
     8         2
7         0         3
     6         4
          5
```

**EXAMPLES:**

| | |
|---|---|
| PRINT JOY (2)<br>135 | Joystick 2 fires to the left. |
| IF (JOY (1) > 127) THEN PRINT ''FIRE'' | Determines whether the fire button is pressed. |
| DIR = JOY(1) AND 15 | Returns direction (only) of joystick 1. |

## LEFT$

Return the leftmost characters of string

**LEFT$ (string,integer)**

**EXAMPLE:**

```
PRINT LEFT$(''COMMODORE'',5)
COMMO
```

## LEN

Return the length of a string

**LEN (string)**

The returned integer value is in the range 0–255.

**EXAMPLE:**

```
PRINT LEN (''COMMODORE128'')
12
```

## LOG

Return natural log of X

**LOG(X)**

The argument X must be greater than 0.

**EXAMPLE:**

```
PRINT LOG (37/5)
2.00148
```

## MID$

Return a substring from a larger string

**MID$ (string,starting position[,length])**

This function extracts the number of characters specified by length (0–255), from string, starting with the character specified by starting position (1–255).

**EXAMPLE:**

```
PRINT MID$(''COMMODORE 128'',3,5)
MMODO
```

## PEEK

Return contents of a specified memory location

**PEEK(X)**

The data will be returned from the bank selected by the most recent BANK command. See the BANK command.

**EXAMPLE:**

```
10 BANK 15:VIC = DEC(''D000'')
20 FOR I = 1 TO 47
30 PRINT PEEK(VIC + I)
40 NEXT
```

This example displays the contents of the registers of the VIC chip (some of which are ever-changing).

## PEN

Return X and Y coordinates of the light pen

**PEN(n)**

where **n** = 0 PEN returns the X coordinate of light pen position on any VIC screen.
**n** = 1 PEN returns the Y coordinate of light pen position on any VIC screen.
**n** = 2 PEN returns the character column position of the 80 column display.
**n** = 3 PEN returns the character row position of the 80 column display.
**n** = 4 PEN returns the (80-column) light pen trigger value.

The VIC PEN values are not sealed and are taken from the same coordinate plane as sprites use. Unlike the 40 column (VIC) screen, the 80 column (8563) coordinates are character row and column positions and not pixel coordinates like the VIC screen. Both the 40 and 80 column screen coordinate values are approximate and vary, due to the nature of light pens. The 80-column read values are not valid until PEN(4) is true.

Light pens are always plugged in to control port 1.

### EXAMPLES:

10 PRINT PEN(0);PEN(1) — Displays the X and Y coordinates of the light pen (for the 40 column screen).

10 DO UNTIL PEN(4):LOOP — Ensures that the read values are valid (for the 80 column screen).

```
20 X = PEN(2)
30 Y = PEN(3)
40 REM:REST OF PROGRAM
```

## π

Return the value of pi (3.14159265)

π

### EXAMPLE:

PRINT π  This returns the result 3.14159265.

## POINTER

Return the address of a variable

**POINTER (variable name)**

This function returns a zero if the variable is not defined.

### EXAMPLE:

A = POINTER (Z)  This example returns the address of variable Z.
NOTE: Address returned is in RAM BANK 1.

# POS

Return the current cursor column position within the current screen window

**POS(X)**

The POS function indicates where the cursor is within the defined screen window. X is a dummy argument, which must be specified, but the value is ignored. The values returned range from 0–39 on the VIC screen and 0–79 on the 80-column screen.

**EXAMPLE:**

```
FOR I = 1 to 10 : ?SPC(I); POS(0): NEXT
```

This displays the current cursor position within the defined text window.

# POT

Returns the value of the game-paddle potentiometer

**POT (n)**

when:

**n** = 1, POT returns the position of paddle #1 (control port 1)
**n** = 2, POT returns the position of paddle #2 (control port 1)
**n** = 3, POT returns the position of paddle #3 (control port 2)
**n** = 4, POT returns the position of paddle #4 (control port 2)

The values for POT range from 0 to 255. Any value of 256 or more means that the fire button is also depressed.

**EXAMPLE:**

```
10 PRINT POT(1)
20 IF POT(1) > 256 THEN PRINT "FIRE"
```

This example displays the value of game paddle 1.

# RCLR

Return color of color source

**RCLR(N)**

This function returns the color (1 through 16) assigned to the color source N (0< = N = < 6), where the following N values apply:

| SOURCE | DESCRIPTION |
|---|---|
| **0** | 40-column background |
| **1** | bit map foreground |
| **2** | multi-color 1 |
| **3** | multi-color 2 |
| **4** | 40-column border |
| **5** | 40- or 80-column character color |
| **6** | 80-column background color |

The counterpart to the RCLR function is the COLOR command.

**EXAMPLE:**

```
10 FOR I = 0 TO 6
20 PRINT ''SOURCE'';I;''IS COLOR CODE'';RCLR(I)
30 NEXT
```

This example prints the color codes for all six color sources.

## RDOT

Return current position or color source of pixel cursor

**RDOT (N)**

where:

**N** = 0 returns the X coordinate of the pixel cursor
**N** = 1 returns the Y coordinate of the pixel cursor
**N** = 2 returns the color source (0–3) of the pixel cursor

This function returns the location of the current position of the pixel cursor or the current color source of the pixel cursor.

**EXAMPLES:**

| | |
|---|---|
| PRINT RDOT(0) | Returns X position of pixel cursor |
| PRINT RDOT(1) | Returns Y position of pixel cursor |
| PRINT RDOT(2) | Returns color source of pixel cursor |

## RGR

Return current graphic mode

**RGR(X)**

This function returns the current graphic mode. X is a dummy argument, which must be specified. The counterpart of the RGR function is the GRAPHIC command. The value returned by RGR(X) pertains to the following modes:

| VALUE | GRAPHIC MODE |
|---|---|
| **0** | **40 column (VIC) text** |
| **1** | **Standard bit map** |
| **2** | **Split screen bit map** |
| **3** | **Multi-color bit map** |
| **4** | **Split screen Multi-color bit map** |
| **5** | **80 column (8563) text** |

**EXAMPLE:**

```
PRINT RGR(0)
1
```
Displays the current graphic mode; in this case, standard bit map mode.

```
PRINT RGR(0)
8
```
Both multi-color bit map and 80-column text modes are enabled.

## RIGHT$

Return sub-string from rightmost end of string

**RIGHT$(string, numeric)**

**EXAMPLE:**

```
PRINT RIGHT$(''BASEBALL'',5)
EBALL
```

## RND

Return a random number

**RND (X)**

If X = 0 RND returns a random number based on the hardware clock.
If X > 0 RND generates a reproducible random number based on the seed value below.
If X < 0 produces a random number which is used as a base called a seed.

**EXAMPLES:**

```
PRINT RND(0)
.507824123
```
Displays a random number between 0 and 1.

```
PRINT INT(RND(1)*100 + 1)
89
```
Displays a random number between 1 and 100.

## RSPCOLOR

Return sprite multicolor values

**RSPCOLOR (X)**

When:

**X = 1** RSPCOLOR returns the sprite multi-color 1.
**X = 2** RSPCOLOR returns the sprite multi-color 2.

The returned color value is a value between 1 and 16. The counterpart of the RSPCOLOR function is the SPRCOLOR statement. Also see the SPRCOLOR statement.

### EXAMPLE:

```
10 SPRITE 1,1,2,0,1,1,1
20 SPRCOLOR 5,7
30 PRINT ''SPRITE MULTI-COLOR 1 IS'';RSPCOLOR(1)
40 PRINT ''SPRITE MULTI-COLOR 2 IS'';RSPCOLOR(2)
RUN

SPRITE MULTI-COLOR 1 IS 5
SPRITE MULTI-COLOR 2 IS 7
```

In this example line 10 turns on sprite 1, colors it white, expands it in both the X and Y directions and displays it in multi-color mode. Line 20 selects sprite multi-colors I and 2 (5 and 7 respectively). Lines 30 and 40 print the RSPCOLOR values for multi-color I and 2.

## RSPPOS

Return the speed and position values of a sprite

**RSPPOS (sprite number,position|speed)**

where sprite number identifies which sprite is being checked, and position and speed specifies X and Y coordinates or the sprite's speed.

When position equals:

**0** RSPPOS returns the current X position of the specified sprite.
**1** RSPPOS returns the current Y position of the specified sprite.

When speed equals:

**2** RSPPOS returns the speed (0-15) of the specified sprite.

### EXAMPLE:

```
10 SPRITE 1,1,2
20 MOVSPR 1,45#13
30 PRINT RSPPOS (1,0);RSPPOS (1,1);RSPPOS (1,2)
```

This example returns the current X and Y sprite coordinates and the speed (13).

# RSPRITE

Return sprite characteristics

**RSPRITE (sprite number,characteristic)**

RSPRITE returns sprite characteristics that were specified in the SPRITE command. Sprite number specifies the sprite (1–8) you are checking and the characteristic specifies the sprite's display qualities as follows:

| CHARACTERISTIC | RSPRITE RETURNS THESE VALUES: | |
|---|---|---|
| **0** | **Enabled(1) / disabled(0)** | |
| **1** | **Sprite color (1-16)** | |
| **2** | **Sprites are displayed in front of (0) or behind (1) objects on the screen** | |
| **3** | **Expand in X direction** | **yes = 1, no = 0** |
| **4** | **Expand in Y direction** | **yes = 1, no = 0** |
| **5** | **Multi-color** | **yes = 1, no = 0** |

**EXAMPLE:**

```
10 FOR I = 0 TO 5
20 PRINT RSPRITE (1,I)
30 NEXT
```

This example prints all 6 characteristics of sprite 1.

# RWINDOW

Returns the size of the current window or the number of columns of the current screen

**RWINDOW (n)**

When **n** equals:

**0** RWINDOW returns the number of lines in the current window.
**1** RWINDOW returns the number of rows in the current window.
**2** RWINDOW returns either of the values 40 or 80, depending on the current screen output format you are using.

The counterpart of the RWINDOW function is the WINDOW command.

**EXAMPLE:**

```
10 WINDOW 1,1,10,10
20 PRINT RWINDOW(0);RWINDOW(1);RWINDOW(2)
RUN
9 9 40
```

This example returns the lines (10) and columns (10) in the current window. This example assumes you are displaying the window in 40 column format.

## SGN

Return sign of argument X

**SGN(X)**

**EXAMPLE:**

```
PRINT SGN(4.5);SGN(0);SGN(–2.3)
1   0 – 1
```

## SIN

Return sine of argument

**SIN(X)**

**EXAMPLE:**

```
PRINT SIN (π/3)
.866025404
```

## SPC

Skip spaces on printed output

**SPC (X)**

**EXAMPLE:**

```
PRINT ''COMMODORE'';SPC(3);''128''
COMMODORE       128
```

## SQR

Return square root of argument

**SQR (X)**

**EXAMPLE:**

```
PRINT SQR(25)
5
```

## STR$

Return string representation of number

**STR$ (X)**

**EXAMPLE:**

```
PRINT STR$(123.45)
123.45

PRINT STR$(–89.03)
–89.03

PRINT STR$(1E20)
1E + 20
```

## TAB

Moves cursor to tab position in present statement

**TAB (X)**

**EXAMPLE:**

```
10 PRINT''COMMODORE''TAB(25)''128''
COMMODORE                 128
```

## TAN

Return tangent of argument in radians

**TAN(X)**

This function returns the tangent of X, where X is an angle in radians

**EXAMPLE:**

```
PRINT TAN(.785398163)
1
```

## USR

Call user-defined subprogram

**USR(X)**

When this function is used, the BASIC program jumps to a machine language program whose starting point is contained in memory locations 4633($1219) and 4634($121A), (or 785($0311) and 786($0312) in C64 mode). The parameter X is passed to the machine-language program in the floating-point accumulator ($63–$68 in C128 mode). A value is returned to the BASIC program through the calling variable. You must direct the value into a variable in your program in order to receive the value back from the floating-point accumulator. An ILLEGAL QUANTITY ERROR results if you don't specify this variable. This allows the user to exchange a variable between machine code and BASIC.

**EXAMPLE:**

```
10 POKE 4633,0
20 POKE 4634,48
30 A = USR(X)
40 PRINT A
```

Place starting location ($3000 = 12288:$00 = 0:$30) = 48 of machine language routine in location 4633 and 4634. Line 30 stores the returning value from the floating-point accumulator. The USER vector is assumed to be in BANK 15. Your machine language routine *MUST* be in RAM bank 0 below address $4000.

## VAL

Return the numeric value of a number string

**VAL(X$)**

**EXAMPLE:**

```
10 A$ = "120"
20 B$ = "365"
30 PRINT VAL (A$ + B$)
RUN
485
```

## XOR

Return exclusive OR value

**XOR (n1,n2)**

This function returns the exclusive OR of the numeric argument values n1 and n2.

**X = XOR (n1,n2)**

where n1, n2, are 2 unsigned values (0-65535)

**EXAMPLE:**

```
PRINT XOR(128,64)
192
```

# RESERVED SYSTEM WORDS (KEYWORDS)

This section lists the words used to make up the BASIC 7.0 language. These words cannot be used within a program as other than a component of the BASIC language. The only exception is that they may be used within quotes (in a PRINT statement, for example).

| | | | | |
|---|---|---|---|---|
| ABS | DELETE | HELP | POT | SPRCOLOR |
| AND | DIM | HEX$ | PRINT | SPRDEF |
| APPEND | DIRECTORY | IF | PRINT# | SPRITE |
| ASC | DLOAD | INPUT | PUDEF | SPRSAV |
| ATN | DO | INPUT# | (QUIT) | SQR |
| AUTO | DOPEN | INSTR | RCLR | SSHAPE |
| BACKUP | DRAW | INT | RDOT | ST |
| BANK | DS | JOY | READ | STASH |
| BEGIN | DS$ | KEY | RECORD | STEP |
| BEND | DSAVE | LEFT$ | REM | STOP |
| BLOAD | DVERIFY | LEN | RENAME | STR$ |
| BOOT | EL | LET | RENUMBER | SWAP |
| BOX | ELSE | LIST | RESTORE | SYS |
| BSAVE | END | LOAD | RESUME | TAB |
| BUMP | ENVELOPE | LOCATE | RETURN | TAN |
| CATALOG | ER | LOG | RGR | TEMPO |
| CHAR | ERR$ | LOOP | RIGHT$ | THEN |
| CHR$ | EXIT | MID$ | RND | TI |
| CIRCLE | EXP | MONITOR | RREG | TI$ |
| CLOSE | FAST | MOVSPR | RSPCOLOR | TO |
| CLR | FETCH | NEW | RSPPOS | TRAP |
| CMD | FILTER | NEXT | RSPRITE | TROFF |
| COLLECT | FN | NOT | RUN | TRON |
| COLLISION | FOR | (OFF) | RWINDOW | UNTIL |
| COLOR | FRE | ON | SAVE | USING |
| CONCAT | GET | OPEN | SCALE | USR |
| CONT | GET# | OR | SCNCLR | VAL |
| COPY | G064 | PAINT | SCRATCH | VERIFY |
| COS | GOSUB | PEEK | SGN | VOL |
| DATA | GOTO | PEN | SIN | WAIT |
| DCLEAR | GO TO | PLAY | SLEEP | WHILE |
| DCLOSE | GRAPHIC | POINTER | SLOW | WIDTH |
| DEC | GSHAPE | POKE | SOUND | WINDOW |
| DEF FN | HEADER | POS | SPC | XOR |

**NOTE:** Keywords shown in parentheses are not implemented in C128 BASIC 7.0.

Reserved variable names are names reserved for the variables DS, DS$, ER, EL, ST, TI and TI$, and the function ERR$. Keywords such as TO and IF or any other names that contain keywords, such as RUN, NEW or LOAD cannot be used.

ST is a status variable for input and output (except normal screen/keyboard operations). The value of ST depends on the results of the last I/O operation. In general, if the value of ST is 0, then the operation was successful.

TI and TI$ are variables that relate to the real time clock built into the Commodore 128. The system clock is updated every 1/60th of a second. It starts at 0 when the Commodore 128 is turned on, and is reset only by changing the value of TI$. The variable TI gives the current value of the clock in 1/60th of a second. TI$ is a string that reads the value of the real time clock as a 24-hour clock. The first two characters of TI$ contain the hour, the third and fourth characters are minutes and the fifth and sixth characters are seconds. This variable can be set to any value (so long as all characters are numbers) and will be updated automatically as a 24-hour clock.

### EXAMPLE:

TI$ = "101530" Sets the clock to 10:15 and 30 seconds (AM).

The value of the clock is lost when the Commodore 128 is turned off. It starts at zero when the Commodore 128 is turned on, and is reset to zero when the value of the clock exceeds 235959 (23 hours, 59 minutes and 59 seconds).

The variable DS reads the disk drive command channel and returns the current status of the drive. To get this information in words, PRINT DS$. These status variables are used after a disk operation, like DLOAD or DSAVE, to find out why the error light on the disk drive is blinking.

ER, EL and the ERR$ function are variables used in error trapping routines. They are usually only useful within a program. ER returns the last error number encountered since the program was RUN. EL is the line where the error occurred. ERR$ is a function that allows the program to print one of the BASIC error messages. PRINT ERR$(ER) prints out the proper error message.

## RESERVED SYSTEM SYMBOLS

The following characters are reserved system symbols.

| | SYMBOL | USE(S) |
|---|---|---|
| + | **Plus sign** | **Arithmetic addition; string concatenation; relative pixel cursor/sprite movement; declare decimal number in machine language monitor** |
| - | **Minus sign** | **Arithmetic subtraction; negative number; unary minus; relative pixel cursor/ sprite movement** |
| * | **Asterisk** | **Arithmetic multiplication** |
| / | **Slash** | **Arithmetic division** |
| ↑ | **Up arrow** | **Arithmetic exponentiation** |
| | **Blank space** | **Separate keywords and variable names** |
| = | **Equal sign** | **Value assignment; relationship testing** |
| < | **Less than** | **Relationship testing** |
| > | **Greater than** | **Relationship testing** |
| , | **Comma** | **Format output in variable lists; command/statement function parameters** |

| | SYMBOL | USE(S) |
|---|---|---|
| . | Period | Decimal point in floating-point constants |
| ; | Semicolon | Format output in variable lists; delimiter |
| : | Colon | Separate multiple BASIC statements on a program line |
| " " | Quotation mark | Enclose string constants |
| ? | Question mark | Abbreviation for the keyword PRINT |
| ( | Left parenthesis | Expression evaluation and functions |
| ) | Right parenthesis | Expression evaluation and functions |
| % | Percent | Declare a variable name as integer; declare binary number in machine language monitor |
| # | Number | Precede the logical file number in input/output statements |
| $ | Dollar sign | Declare a variable name as a string and declare hexadecimal number in machine language monitor |
| & | And sign | Declare octal number in machine language monitor |
| π | Pi | Declare the numeric constant 3.141592654 |

# 3

# ONE STEP BEYOND SIMPLE BASIC

This chapter takes you one step beyond simple BASIC and presents a collection of useful routines. You can incorporate these routines into your own programs as needed. In most cases the routines will require only line number changes to be fitted into your programs.

## CREATING A MENU

A *menu* is a list of choices you select to perform a specific operation within an application program. A menu directs the computer to a particular part of a program. Here is a general example of a menu program:

```
5 REM MENU SKELETON
10 SCNCLR 0
20 PRINT"1. FIRST ITEM"
30 PRINT"2. SECOND ITEM"
40 PRINT"3. THIRD ITEM"
50 PRINT"4. FOURTH ITEM"
100 PRINT:PRINT"SELECT AN ITEM FROM ABOVE"
110 GETKEY A$
120 A=VAL (A$): IF A>4 THEN 10
130 ON A GOSUB 1000,2000,3000,4000
140 GOTO 10:REM RETURN TO MENU
999 STOP
1000 REM START FIRST ROUTINE FOR ITEM ONE HERE
1999 RETURN
2000 REM START SECOND ROUTINE HERE
2999 RETURN
3000 REM START THIRD ROUTINE HERE
3999 RETURN
4000 REM START FOURTH ROUTINE HERE
4999 RETURN
```

**Program 3–1. Menu Skeleton**

The SCNCLR 0 command in line 10 clears the 40-column screen. (Use SCNCLR 5 if you are using the 80-column screen. The easiest selection is by a number. You may use as many selections as can fit on the screen. Line 100 displays a message to the user. The GETKEY command in line 110 forces the computer to wait for a key to be pressed. Since a key represents a character symbol, A$ is a string variable. So that it can be interpreted as a numeric value in an ON GOTO statement, the string variable is converted to a number with the VAL function in line 120. The IF . . . THEN statement in line 120 screens user errors by preventing the user from selecting a number that is not in the range of numbers used for choices (4). Line 130 directs control to the appropriate section (i.e., line number) in your program. Since four selections are offered in this example, you must include at least four line numbers. Line 1999 returns to the menu at the end of each subroutine that you add at lines 1000, 2000, 3000 and 4000 in the menu skeleton.

## BUFFER ROUTINE

The C128 keyboard buffer can hold and dispense up to ten characters from the keyboard. This is useful in a word processing program where it is possible at certain moments to type faster than the software can actually process. The characters that haven't been displayed yet are temporarily stored in the keyboard buffer. The computer can hold the next instruction in the buffer for use when the program is ready. This buffer allows a maximum of ten characters in queue. To see the buffer in action, enter the command SLEEP 5 and immediately press ten different letter keys. After five seconds, all ten characters are displayed on the screen.

Here is a buffer routine that allows you to put items in the keyboard buffer from within a program so they are dispensed automatically as the computer is able to act upon them.

In line 10, memory location 208 (198 in C64 mode) is filled with a number between 0 and 10—the number of keyboard characters in the keyboard buffer. In line 20, memory locations 842 through 851 (631–640 in C64 mode) are filled with any ten characters you want placed there. In this example, seven characters are in the buffer, each a carriage RETURN character. CHR$(13) is the character string code for the carriage return character.

Line 40 places the text "?CHR$(156)" on the screen, but does not execute the instruction. Line 50 displays the word "LIST" on the screen. Neither command is executed until the program ends. In the C128, the keyboard buffer automatically empties when a program ends. In this case, the characters in the buffer (carriage return) are emptied and act as though you are pressing the RETURN key manually. When this occurs on a line where the commands from lines 40 and 50 are displayed, they are executed as though you typed them in direct mode and pressed the RETURN key yourself. When this program ends, the character color is changed to purple and the program is LISTED to the screen. This technique is handy in restarting programs (with RUN or GOTO).

The next section gives a practical example of using the buffer routine.

```
10 POKE 208,7:REM SPECIFY # OF CHARS IN BUFFER
20 FOR I=842 TO 849:POKE I,13:NEXT:REM PLACE CHARS IN BUFFER
30 SLEEP 2 :REM DELAY
40 SCNCLR:PRINT:PRINT:PRINT:PRINT:PRINT:PRINT:PRINT"? CHR$(156)"
50 PRINT:PRINT:PRINT:PRINT"LIST":REM PLACE LIST ON SCREEN
60 PRINT CHR$(19):PRINT:PRINT:REM GO HOME AND CURSOR DOWN TWICE
70 REM WHEN PROGRAM ENDS, BUFFER EMPTIES AND EXECUTES 7 RETURNS.
80 REM THIS CHANGES CHAR COLOR TO PURPLE AND LISTS THE PROGRAM AUTOMATICALLY
90 REM AS IF YOU PRESSED THE RETURN KEY MANUALLY
```

**Program 3–2. Buffer Return**

## LOADING ROUTINE

The buffer can be used in automatic loader routines. Many programs often involve the loading of several machine code routines as well as a BASIC program. The results of the following loader are similar to many found on commercial software packages.

```
2 COLOR 4,1:COLOR 0,1:COLOR 5,1
5 A$="PICTURE"
10 SCNCLR:PRINT:PRINT:PRINT:PRINT"LOAD"CHR$(34)A$CHR$(34)",8,1"
15 PRINT:PRINT:PRINT"NEW"
25 B$="FILE3.BIN"
30 PRINT:PRINT:PRINT"LOAD"CHR$(34)B$CHR$(34)",8,1"
45 PRINT:PRINT:PRINT:PRINT:PRINT:PRINT"SYS12*256"
90 PRINT CHR$(5):PRINT"          GREETINGS FROM COMMODORE"
100 PRINT"     PLEASE STAND BY - LOADING":PRINT CHR$(144)
200 PRINT CHR$(19)
300 POKE208,7:FORI=842TO851:POKEI,13:NEXT
```

**Program 3–3. Loading Routine**

Line 2 colors the border, screen and characters black. Line 5 assigns A$ the filename "PICTURE", which in this example assumes that it is an 8K binary file of a bit-mapped screen. Line 10 places the LOAD instruction for the picture file on the screen, but does not execute it. A carriage return from the keyboard buffer executes the load instruction once the program ends and the keyboard buffer empties. Line 15 prints the word "NEW" on the screen. Again, this operation is not carried out until a carriage return is executed on the same line once the keyboard buffer empties. After loading a machine language program, a NEW is performed to set pointers to their original positions and clear the variable storage area. Line 30 displays the second load instruction for the machine language program "FILE3.BIN". This hypothetical program enables the bit mapped PICTURE, and anything else you want to supply in the program. Line 45 initiates (SYS12*256), the "FILE3.BIN" program starting at 3072 ($0C00) once the keyboard buffer empties. This is only a template sample for you to follow. "PICTURE" and "FILE3.BIN" are programs you supply and are only used to illustrate one technique of automatic loading. Since the previous character color was black, all the loading instructions are displayed in black on a black background, so they can't be seen. The CHR$(5) in line 90 changes the character color to white, so the only visible messages are the ones in white in lines 90 and 100, while the disk drive is loading "PICTURE" and "FILE3.BIN". Line 300 is the buffer routine.

If you were to do each step manually it would require seven "RETURNS". This program places seven carriage return characters in the keyboard buffer, and they are dispensed automatically when the program ends. As each RETURN is accepted, the corresponding screen instruction is enacted automatically as if you had pressed the RETURN key manually.

## PROGRAMMING THE C128 FUNCTION KEYS

As each of the function keys (F1 through F8) is pressed, the computer displays a BASIC command on the screen and in some cases acts immediately. These are known as the default values of the function keys. Enter a KEY command to get a list of function key values at any time.

## CHANGING FUNCTION KEYS

You can change the value assigned to any function key by entering the KEY command followed by the number (1 through 8) of the key you want changed, a comma, and the new key instruction in a string format. For example:

```
KEY1, "DLOAD" + CHR$(34) + "PROGRAM NAME"
+ CHR$(34) + CHR$(13) + "LIST" + CHR$(13)
```

This tells the computer to automatically load the BASIC program called "program name" and list it immediately (whenever F1 is pressed). The character string code value for the quote character is 34. It is necessary for LOAD and SAVE operations. The character string code value for RETURN is 13 and provides immediate execution. Otherwise, the instruction is only displayed on the screen and requires you to supply the additional response and press the RETURN key.

The following example uses the ASCII value for the ESCape key to assign the F3 key to cause a downward scroll:

```
KEY 3,CHR$(27) + "W"
```

> **NOTE:** All eight KEY definitions in total must not exceed *246* characters.

## USING C64 FUNCTION KEY VALUES IN C128 MODE

Programs previously written for the C64 which incorporate the function keys may still be used in C128 mode by first assigning the C64 ASCII values to them with this instruction:

```
10J = 132:FORA = 1TO2:FORK = ATO8STEP2:J = J + 1:KEYK,CHR$(J):NEXT:
NEXT
```

# HOW TO CRUNCH BASIC PROGRAMS

Several techniques known collectively as *memory crunching* allow you to get the most out of your computer's memory. These techniques include eliminating spaces, using multiple instructions, having syntax relief, removing remark statements, using variables, and in general using BASIC intelligently.

## ELIMINATING SPACES

In most BASIC commands, spacing is unnecessary, except inside quotes when you want the spaces to appear on the screen. Although spaces improve readability, the extra space consumes additional memory. Here is an instructional line presented both ways:

```
10INPUT"FIRST NAME";N$:FOR T = A TO M:PRINT "OK":

10INPUT"FIRST NAME";N$:FORT = ATOM:PRINT"OK":
```

## USING MULTIPLE INSTRUCTIONS

Colons allow you to place several instructions within a single program line. Each program line consumes additional memory. Be careful, however, crunching IF statements. Any instruction after the IF statement with the same line number can be bypassed along with the IF . . . THEN condition. The following line is the equivalent of five lines:

```
    (A)
10 PRINTX:INPUTY:PRINTY:SCNCLR0:?J

    (B)
10 PRINTX
20 INPUTY
30 PRINTY
40 SCNCLR0
50 PRINTJ
```

Example A requires less space in memory and on disk. Example B requires 16 additional bytes; 2 bytes for each additional line number and 2 bytes for the link to the next line number.

## SYNTAX RELIEF

Some BASIC syntax is very flexible and this flexibility can be used to your advantage. The LET statement, for example, can be written without LET. LET Y= 2 is the same as Y=2. Although it is good practice to initialize all variables to zero, it is not necessary since the computer automatically sets all variables to zero, including subscripted variables. DIMension all arrays (subscripted variables) to have twelve or more elements. The C128 automatically dimensions each variable to have eleven subscripted elements if no dimension is specified following DIM and the variable names. Often semicolons are not required in PRINT statements. Both of these perform the same results:

```
10 PRINT"A";Z$;"WORD";CHR$(65);"NOW $"
20 PRINT"A"Z$"WORD"CHR$(65)"NOW $"
```

## REMOVING REM STATEMENTS

Although REM statements are useful to the programmer, removing them makes a considerable amount of memory available again. It might be a good idea to create a separate listing with REM statements.

## USING VARIABLES

Replace repeated numbers with a variable. This is especially important with large numbers such as memory addresses. POKEing several numbers in sequence conserves memory if a variable is used, such as POKE 54273+V, etc. Of course, single-letter variable names require the least memory. Reuse old variables such as those used in FOR . . . NEXT loops. Whenever possible, make use of integer variables since they consume far less memory than floating-point variables.

## USING BASIC INTELLIGENTLY

If information is used repeatedly, store the data in integer arrays, if possible. Use DATA statements where feasible. Where a similar line is used repeatedly, create a single line with variables and access it with GOSUBs. Use TAB and SPC functions in place of extensive cursor controls.

## MASKING BITS

Any of the bits within a byte can be controlled individually, using the Boolean operators AND and OR. Calculations with AND and OR are based on a truth table (Table 3–1) showing the results given all possible true and false combinations of the arguments X and Y.

| X | Y | X AND Y | X OR Y |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 |

**Table 3–1. AND and OR Truth Table**

With "0" representing False and "1" Truth, Table 3–1 shows how the operators AND and OR work. The result of an AND operation between two bits is only true if both bits are true (1). Otherwise the combination is false. Any bit combination with a zero yields a zero in an AND operation. The result of an AND operation is only true (equal to 1) if both bits are true (equal to 1).

The result of an OR operation is only false if each bit is false. Otherwise the result is true. Any bit combination with a one yields a one in an OR operation. ONLY two zeros result in a zero.

Observe the following example with the numbers 5 and 6 in binary form. When you type the command PRINT 5 AND 6, the result is 4. Here's why:

| | | | |
|---|---|---|---|
| | 5= | 0000 | 0101 |
| | 6= | 0000 | 0110 |
| ANDed | 4= | 0000 | 0100 |

Instead of adding, ANDing performs a bit-by-bit comparison in accordance with the rules of the AND truth table. Compare column-for-column from the right: 1 AND 0=0, 0 AND 1=0, 1 AND 1=1, 0 AND 0=0. The result "0100" converted to decimal is the number 4.

What is the effect of ORing 5 and 6? Again comparing bit-by-bit, using the rules from the OR truth table:

| | | | |
|---|---|---|---|
| | **5 =** | **0000** | **0101** |
| | **6 =** | **0000** | **0110** |
| **ORing** | **7 =** | **0000** | **0111** |

The result 0111 is decimal 7. Notice from the right that 1 OR 0 = 1, 0 OR 1 = 1, 1 OR 1 = 1 and 0 OR 0 = 0.

Understanding how these OR and AND combinations work gives you the power to control individual bits within your computer's memory. Many of the 8-bit bytes utilize each bit for separate control parameters.

## USING OR AND AND TO MODIFY THE BIT VALUES IN A BYTE

A byte is a group of eight binary digits labeled, from right to left, 0 to 7. Each binary digit position represents a decimal value equal to two raised to the power of the position number. For example, the decimal value of position 6 is 2**6 or 64. From left to right the positions are:

7 6 5 4 3 2 1 0

and the corresponding values in decimal are:

128 64 32 16 8 4 2 1

To turn on a bit, place a "1" in its position. To turn it off, enter a "0". Hence the binary 10010000 has bits 4 and 7 on. Their values are 128 and 16. So if a particular byte is POKED with 144 (128 + 16), these two bits are turned on. To turn bits on, store (POKE) a new value to that byte—a value equal to the sum of all the decimal equivalents of all the bits that are enabled (on). Of course, you do not always know which bits are already on. You may only want to turn on specific bits without affecting the others. That's the purpose of the logical operations AND and OR.

First, obtain the decimal value of the byte by PEEKing. Then add the decimal value of the bit you wish to turn on. The following command turns on bit 2 of memory address "V":

```
POKEV, PEEK(V) + 4
```

This assumes bit 2 (third bit from the right) had a value of 0. Had it already been "on," it would have no effect. To prevent such confusion, the C128 uses the power of Boolean Logic.

Ideally you want to read (PEEK) each bit. The proper approach is to OR the byte with an operand byte which will yield the desired binary value. Suppose we want to turn on bit 5; the operand byte becomes 00100000. By ORing this with any byte it will affect *only* bit 5, because any combination involving 1 in an OR operation results in 1. Thus no bit already ON can be inadvertently turned off.

```
POKEV,PEEK(V) OR 32
```

Just as OR turns a switch on, AND can turn a switch off—with a slight difference. AND results in a "1" *only* if both bits compared are "1." The trick is to compare the

byte in question with an operand byte of all ON bits except the bit you want turned off. Bits to remain on will not be affected. To turn off bit 5, AND the byte in question with the mirror image of 00100000 or the operand byte 11011111. In decimal this value is always 255 minus the value of the bit(s) you want to turn off. Thus:

POKEV,PEEK(V) AND (255–32)

turns off bit 5.

Use OR to turn bits ON
Use AND to turn bits OFF

**EXAMPLES:**

| | |
|---|---|
| POKEW,PEEK(W) OR 129 | Turns ON bits 0 and 7 of memory address W. |
| POKES,PEEK(S) AND 126 | Turns OFF bits 0 and 7 of memory register S (Remember 255–129 = 126) |
| POKEC,PEEK(C)AND254 | Turns OFF bit 0 |
| POKEC,PEEK(V)OR63 | Turns ON all bits except 6 and 7 |

# DEBUGGING PROGRAMS

No program when first written is free of "bugs" or errors. The process of finding errors and removing them, *debugging*, combines editing with problem solving.

## SYNTAX ERRORS

Syntax errors result from misspelling or misusing the guidelines and formats of BASIC commands. An error message is displayed on the screen defining the line where the error occurs. Typing HELP <RETURN> or pressing the HELP key also highlights the line with the error. Common syntax errors include misspelled BASIC terms, misplaced punctuation, unpaired parentheses, reserved variable names such as TI$, use of line numbers that do not exist, etc.

## LOGIC ERRORS

Sometimes errors exist in the program logic, with the result that the program doesn't do exactly what you think it is supposed to do. Some logic errors are caused by the order of instructions. One common fault occurs when you forget that anything on a line after an IF statement is affected by the IF condition.

Some errors in logic require a trial-and-error investigation. This is best initiated by asking the computer for help in BASIC.

## USING A DELAY

Where the computer responds rapidly, it often helps to see a response by inserting a SLEEP command for a temporary time delay. This gives you a chance to see exactly what is happening in the program.

## USING PRINT AND STOP

Insert STOP statements within your program prior to the suspect instruction line. Good locations are at the end of specific tasks. Run the program. After the STOP statement puts you into direct mode, use the PRINT command to identify clues to the problem by determining the values of the various variables, especially those within loops. Check these with what you expect. Continue the program with CONT to the next STOP statement until you modify your program.

## TRAPPING AN ERROR

Debugging is the art of detecting the source of problem. The following program is perfectly valid; however, it produces an error when B equals zero.

```
10 INPUT A,B
20 PRINT A/B
30 GOTO 10
```

Although in this case the computer defines the error as a DIVISION BY ZERO error, it is not always obvious how the variable B became a zero. It could have been derived from a complex formula embedded in your program, or directly inputting the value zero into a variable.

The BASIC TRAP command has a technique of trapping such an error in a program without crashing. Since you can't always foresee all the possible values of the variable B, you can screen the probable error of division of zero by including a TRAP at the beginning of the program.

```
 5  TRAP 50
10  INPUT A,B
20  PRINTA/B
30  GOTO10
50  PRINT''DIVISION BY ZERO IS NOT POSSIBLE''
60  PRINT''ENTER ANOTHER NUMBER FOR B BESIDES ZERO''
70  RESUME
```

RESUME is required after the TRAP response in order to reactivate the TRAP. If you include the option to enter a replacement for B, RESUME without a line number returns to the cause of the error—line 20—and executes it as follows:

```
65  INPUT B
```

The use of RESUME NEXT proceeds with the next line after the TRAP command, i.e., line 10.

TRAP tells the computer to go to a specific line number whenever an error occurs. Do NOT use TRAP until you have removed all syntax errors first. TRAP can only catch the error condition it is looking for. An error in the syntax or the logic of your TRAP routine may cause an error, or may not catch the error that you are looking for. In other words, TRAP routines are sensitive to errors, too.

## ERROR FUNCTIONS

Several reserved variables inherent in the system store information about program errors. ER stores the error number. EL stores the relevant program line number. ERR$(N) returns the string representing ER or EL. In the example of division by zero, ERR$(ER) returns "DIVISION BY ZERO" and ERR$(EL) returns "BREAK". Add this to the program in the previous section. See Appendix A for a complete listing of errors.

## D O S ERRORS

Information on disk errors is determined from the variables DS and DS$ where DS is the error number (See Appendix B) and DS$ provides the error number, error message, and track and sector of the error. DS$ reads the disk error channel and is used during a disk operation to determine why the disk drive error light is blinking.

Trying to read a directory without a disk in place results in the following error when the PRINT DS$ command is issued:

74, DRIVE NOT READY, 00, 00

Appendix B highlights specific causes of errors. To convert a function key to read the disk-drive error channel automatically, use:

KEY 1, "PRINT DS$+CHR$(13)

## TRACING AN ERROR

Some programs have many complex loops that are tedious to follow. A methodical step-by-step trace is useful. The BASIC TRON and TROFF commands can be used within a program as a debugging tool to trace specific routines.

Some errors can only be found by acting like the computer and methodically following each instruction step-by-step, and then doing all the calculations until you discover something wrong. Fortunately the Commodore 128 can trace errors for you. Enter the direct command TRON prior to running a program. The program displays each line number as they occur in brackets, followed by each result. (To slow down the display, hold the Commodore ( C= ) key down.)

Try it with this double loop:

```
10  FOR A=1T05
20  FOR B=2T06
30  C=B*A:K=K+C:PRINTK
40  NEXTB:NEXTA
50  PRINTK
```

The results will start off like this:

```
[10] [20] [30] [30] [30]2
[40] [30] [30] [30]5
```

meaning the first printed result is the number 2 after operations in lines 10, 20, 30 are performed. Then lines 40 and 30 result in 5, etc. Notice three activities were performed in line 30. The Trace function is turned off with the direct command TROFF.

# WINDOWING

The standard screen display size is 40- or 80-columns by 25 lines. It is often convenient to have a portion of the screen available for other work. The process of producing and isolating small segments of your screen is called "windowing."

## DEFINING A WINDOW

There are two ways to create a window—either directly or within a program using the WINDOW command. Using the ESCape key followed by a T or B is all that is necessary to describe and set a window.

Here's how to define a window in direct mode:

1. Move the cursor to the upper-left corner position of the proposed window. Press the (ESC) escape key, then press the letter T key.
2. Move the cursor to the bottom right corner and press the escape key (ESC) then press the letter B key.

Now that your window is in effect, all commands and listings remain in the window until you exit by pressing the HOME key twice. This is useful if you have a listing on the main screen and wish to keep it while you display something else in a window. See Chapter 13, the Commodore 128 Operating System, under the screen editor for special ESCape controls within a window.

Although it is possible to define several windows simultaneously on the screen, only one window can be used at a time. The other windows remain on the display, but they are inactive. To re-enter a window you have exited, define the top and bottom corners of the window with the ESC T and ESC B commands, respectively, as you did originally.

The second way to define a window is with the BASIC window command. The command:

```
WINDOW 20,12,39,24,1
```

establishes a window with the top-left corner at column 20, row 12, and the bottom-right corner at column 39, row 24. The 1 signifies the area to be cleared. Once this command is specified, all activities are restricted to this window.

Use the window command within a program whenever you want to perform an activity in an isolated area on the screen.

# ADVANCED BASIC PROGRAMMING TECHNIQUES FOR COMMODORE MODEMS

The following information tells you how to:

1. Generate Touch Tone™ frequencies
2. Detect telephone ringing
3. Program the telephone to be on or off the hook
4. Detect carrier

The programming procedures operate in C128 or C64 modes with the Modem/300. In C128 mode, select a bank configuration which contains BASIC, I/O, and the Kernal.

## GENERATING TOUCH TONE (DTMF) FREQUENCIES

Each button on the face of a Touch Tone telephone generates a different pair of tones (frequencies). You can simulate these tones with your Commodore 128 computer. Each button has a row and column value in which you must store the appropriate memory location in order to output the correct frequency. Here are the row and column frequency values that apply to each button on the face of your Touch Tone telephone:

TOUCH TONE FREQUENCY TABLE

| | COLUMN 1 (1029 HZ) | COLUMN 2 (1336 HZ) | COLUMN 3 (1477 HZ) |
|---|---|---|---|
| **Row 1 (697 Hz)** | **1** | **2** | **3** |
| **Row 2 (770 Hz)** | **4** | **5** | **6** |
| **Row 3 (852 Hz)** | **7** | **8** | **9** |
| **Row 4 (941 Hz)** | * | **0** | # |

To generate these tones in BASIC with your Commodore 128, follow this procedure:

1. Initialize the sound (SID) chip with the following BASIC statements:

```
SID = 54272
POKE SID + 24,15:POKE SID + 4,16
POKE SID + 11,16:POKE SID + 5,0:POKE SID + 12,0
POKE SID + 6,15*16:POKE SID + 13,15*16:POKE SID + 23,0
```

2. Next, select one row and one column value for each digit in the telephone number. The POKE statement for each row and column are as follows:

```
Column 1: POKE SID, 117:POKE SID + I,77
Column 2: POKE SID,152:POKE SID + 1,85
Column 3: POKE SID,I61, POKE SID + 1,94
Row 1: POKE SID + 7,168:POKE SID + 8,44
Row 2: POKE SID + 7,85,:POKE SID + 8,49
Row 3: POKE SID + 7,150:POKE SID + 8,54
Row 4: POKE SID + 7,74 :POKE SID + 8,60
```

For example, to generate a tone for the number 1, POKE the values for row 1, column 1 as follows

```
POKE SID + 7,168:POKE SID + 8,44:REM ROW I
POKE SID,117:POKE SID + 1,77:REM COLUMN 1
```

3. Turn on the tones and add a time delay with these statements:

```
POKE SID + 4,17:POKE SID + 11,17:REM ENABLE TONES
FOR I=1 TO 50:NEXT:REM TIME DELAY
```

4. Turn off the tones and add a time delay with the following statements:

```
POKE SID + 4,16:POKE SID + 11,I6:REM DISABLE TONES
FOR I=1 TO 50:NEXT:REM TIME DELAY
```

5. Now repeat steps 2 through 4 for each digit in the telephone number you are dialing.
6. Finally, disable the sound chip with this statement:

```
POKE SID + 24,0
```

## DETECTING TELEPHONE RINGING

To detect whether your telephone is ringing using a Commodore 128, use the following statement:

```
IF (PEEK(56577) AND 8) = 0 THEN PRINT "RINGING"
```

If bit 3 of location 56577 contains a value other than 0, the phone is not ringing.

## PROGRAMMING THE TELEPHONE TO BE ON OR OFF THE HOOK

To program the phone to be off the hook using a Commodore 128, enter the following statements in a program:

```
OH = 56577:HI = 32:LO = 255 - 32
POKE (OH + 2),(PEEK(OH + 2) OR HI)
POKE OH,(PEEK(OH) AND LO)
```

To hang up the phone with a Commodore 128, enter this statement in a program:

```
POKE OH,(PEEK(OH) OR HI)
```

Here is the procedure to dial and originate a communication link:

1. Set the modem's answer/originate switch to the "O" for originate.
2. Program the telephone to be OFF the hook.
3. Wait 2 seconds (FOR I = 1 to 500:NEXT:REM 2-SECOND DELAY)
4. Dial each digit and follow it with a delay (FOR I = 1 TO 50:NEXT)
5. When a carrier (high pitched tone) is detected, the Modem/300 automatically goes on-line with the computer you are connecting with.
6. Program the phone to hang up when you are finished.

Here is the procedure to answer a call:

1. Set the modem's answer/originate switch to "A" for answer.
2. To manually answer, program the telephone to be OFF the hook.
3. To automatically answer, detect if the phone is ringing then program the phone to be OFF the hook.
4. The Modem/300 automatically answers the call.
5. Program the phone to hang up when you are finished.

## DETECTING CARRIER

Your Commodore Modem/1200 and Modem/300 are shipped from the factory with the ability to detect a carrier on the Commodore 128.

That ability is useful in an unattended auto-answer mode. By monitoring the carrier detect line, the computer can be programmed to hang up after loss of carrier. Since a caller may forget to hang up, your program should monitor the transmit and receive data lines. If there is no activity for five minutes or so, the modem itself should hang up.

To detect carrier on the Commodore 128, the following statement can be used in a BASIC program:

```
OH = 56577:
IF ((PEEK (OH) AND 16) = 0) THEN PRINT "CARRIER DETECTED"
```

If bit 4 of location 56577 contains a value other than 0, then no carrier is detected.

## ROTARY (PULSE) DIALING

In order to dial a number with a modem, the software in the computer must generate pulses at a prescribed rate. In the United States and Canada, the rate is between 8 and 10 pulses per second with a 58% to 64% break duty cycle. Most people, however, use 10 pulses per second with a 60% break duty cycle.

So to make a call, your software must first take the phone "off-hook" (the equivalent of you picking up the receiver). Then to dial the first digit, a 3 for instance,

the software must put the phone on-hook for 60 milliseconds and off-hook for 40 milliseconds. Repeat this process three times to dial a 3.

The same method is used to dial other digits, except 0, which is pulsed ten times. Pause at least 600 milliseconds between each digit.

## USING ESCAPE CODES

To perform any of the escape capabilities within a program, use a line such as:

```
10150 PRINT CHR$(27)+"U"
```

to create an underline cursor (in 80-column only). For example, to clear from the cursor to the end of a window:

```
10160 PRINT CHR$(27)+"@"
```

(See the Screen Editor section of Chapter 13 for all the escape and control codes available on the Commodore 128.)

# RELOCATING BASIC

To relocate the beginning or ending of BASIC (in C128 mode) for additional memory or to protect machine-language programs from being overwritten by BASIC text, it is necessary to redefine the starting and ending pointers in required memory addresses.

The Start of BASIC pointer is located at address 45($2D) and 46($2E). The Top of BASIC pointer is at addresses 4626($1212) and 4627($1213). The following instruction displays the default locations of the beginning and end of BASIC text, respectively (when a VIC bit-mapped screen is not allocated):

```
PRINT PEEK(45),PEEK(46),PEEK(4626),PEEK(4627)

1  28  0  255
```

Since the second number in each case is the high byte value, the default start of basic is 28*256 plus 1 or 7169 while the top is 255*256 or 65280.

The following command reduces the size of BASIC text (program) area by 4K by lowering the top of BASIC to address 61184 (239*256):

```
POKE4626,239:POKE4627,0:NEW
```

To move the beginning of BASIC up in memory by 1K, from 7168 to 8192, use this command line:

```
POKE 46,32:POKE45,1:NEW
```

This is the case only when a bit-mapped graphics screen is not allocated. Remember, the beginning of BASIC starts at 16384($4000) when a bit-mapped screen is allocated, and other parts of memory are shifted around.

## MERGING PROGRAM AND FILES

Files can be merged (combined) by opening an existing file and locating the pointer to the end of the file so subsequent data can be written to the disk file. C128 BASIC has included the APPEND command to accomplish this:

APPEND#5, "FILE NAME"

opens channel 5 to a previously stored file named "FILE NAME." Subsequent write (PRINT#5) statements will add further information to the file. APPEND is primarily used for data files.

The command CONCAT allows the concatenation (combine in sequence) of two files or programs while maintaining the name of the first.

CONCAT"PART2B" TO "PART2"

creates a new file called Part 2, consisting of the old Part 2, plus the new Part 2b in sequence. Concatenated BASIC program files must be renumbered before they can work. Other corrections may also be necessary.

The BASIC routines described in this chapter can greatly enchance the capabilities of your programs. So far, BASIC has been discussed in detail. The machine language programming introduced in Chapter 5 can extend program capabilities even further. And, as shown in Chapter 7, for still greater flexibility and power, you can combine BASIC and machine language in your programs.

# 4

# COMMODORE 128 GRAPHICS PROGRAMMING

## HOW TO USE THE GRAPHICS SYSTEM

# COMMODORE 128 VIDEO FEATURES

In C128 Mode, Commodore BASIC 7.0 offers fourteen high-level graphics commands that make difficult programming jobs easy. You can now draw circles, boxes, lines, points and other geometric shapes, with ten high level commands such as DRAW, BOX and CIRCLE, and with four sprite commands. (The sprite commands are described in Chapter 9.) You no longer have to be a machine language programmer, or purchase additional graphics software packages to display intricate and visually pleasing graphics displays—the Commodore 128 BASIC graphics capabilities take care of this for you. Of course, if you *are* a machine language programmer or a software developer, the exceptional C128 video hardware features offer high price/performance value for any microcomputer application.

The C128 graphics features include:

- Specialized graphics and sprite commands
- 16 colors
- 6 display modes, including:
  - Standard character mode
  - Multi-color character mode
  - Extended background color mode
  - Standard bit map mode
  - Multi-color bit map mode
  - Combined bit map and character modes (split-screen)
- 8 programmable, movable graphic objects called SPRITES which make animation possible
- Custom programmable characters
- Vertical and horizontal scrolling

The Commodore 128 is capable of producing two types of video signals: 40-column composite video, and 80-column RGBI video. The composite video signal, channeled through a VIC II (Video Interface Controller) chip (8564)—similar to that used in the Commodore 64—mixes all of the colors of the spectrum in a single signal to the video monitor. The 8563 separates the colors red, green and blue to drive separate cathode ray guns within the video monitor for a cleaner, crisper and sharper image than composite video.

The VIC II chip supports all of the Commodore BASIC 7.0 graphics commands, SPRITES, sixteen colors, and the graphic display modes mentioned before. The 80-column chip, primarily designed for business applications, also supports sixteen colors (a few of which are different from those of the VIC chip), standard text mode, and bit map mode. Sprites are not available in 80-column output. Bit map mode is not supported by the Commodore BASIC 7.0 language in 80-column output. The 80-column screen can be bit mapped through programming the 8563 video chip with machine language programs. See Chapter 10, Programming the 80-Column (8563) Chip, for information on bit mapping the 80-column screen.

This chapter discusses how to use the Commodore 128 graphics features through BASIC using the VIC (40-column) screen. Except for the sprite commands, each graphic command is listed in alphabetical order. The sprite commands are covered in Chapter 9. Following the format of each command are example programs that illustrate the features of that command. Wherever possible, machine language routines are included to show how the machine language equivalent of a BASIC graphics command operates.

Chapter 8, The Power Behind Commodore 128 Graphics, is a description of the inner workings of the Commodore 128 graphics capabilities. It explains how screen, color and character memory are used and how these memory components store and address data in each display mode. Chapter 9 then explains how to use sprites with the new BASIC commands. Chapter 9 also discusses the inner workings of sprites, their storage and addressing requirements, color assignments, and describes how to control sprites through machine language.

# TYPES OF SCREEN DISPLAY

Your C128 displays information several different ways on the screen; the parameter "source" in the command pertains to three different modes of screen display.

## TEXT DISPLAY

Text display shows only text or characters, such as letters, numbers, special symbols and the graphics characters on the front faces of most C128 keys. The C128 can display text in both 40-column and 80-column screen formats. Text display includes standard character mode, multi-color character mode and extended background color mode.

The Commodore 128 normally operates in standard character mode. When you first turn on the Commodore 128, you are automatically in standard character mode. In addition, when you write programs, the C128 is in standard character mode. Standard character mode displays characters in one of sixteen colors on a background of one of sixteen colors.

Multi-color character mode gives you more control over color than the standard graphics modes. Each screen dot, a pixel, within an 8-by-8 character grid can have one of four colors, compared with the standard mode which has only one of two colors. Multi-color mode uses two additional background color registers. The three background color registers and the character color register together give you a choice of four colors for each dot within an 8-by-8 dot character grid.

Each pixel in multi-color mode is twice as wide as a pixel in standard character mode and standard bit map mode. As a result, multi-color mode has only half the horizontal resolution (160 × 200) of the standard graphics modes. However, the increased control of color more than compensates for the reduced horizontal resolution.

Extended background color mode allows you to control the background color and foreground color of each character. Extended background color mode uses all four background color registers. In extended color mode, however, you can only use the first sixty-four characters of the screen code character set. The second set of sixty-four characters is the same as the first, but they are displayed in the color assigned to

background color register two. The same holds true for the third set of sixty-four characters and background color register three, and the fourth set of sixty-four characters and background color register four. The character color is controlled by color memory. For example, in extended color mode, you can display a purple character with a yellow background on a black screen.

Each of the character display modes receives character information from one of two places in the Commodore 128 memory. Normally, character information is taken from character memory stored in a separate chip called ROM (Read Only Memory). However, the Commodore 128 gives you the option of designing your own characters and replacing the original characters with your own. Your own programmable characters are stored in RAM.

## BIT MAP DISPLAY

Bit map mode allows you to display highly detailed graphics, pictures and intricate drawings. This type of display mode includes standard bit map mode and multi-color bit map mode. Bit map modes allow you to control each individual screen dot or pixel (picture element) which provides for considerable detail in drawing pictures and other computer art. These graphic displays are only supported in BASIC by the VIC chip.

The 80-column chip is designed primarily for character display, but you can bit map it through your own programs. See Chapter 10, Programming the 80-Column (8563) Chip, for detailed information.

The difference between text and bit map modes lies in the way in which each screen addresses and stores information. The text screen can only manipulate entire characters, each of which covers an area of 8 by 8 pixels on the screen. The more powerful bit map mode exercises control over each pixel on your screen.

Standard bit map mode allows you to assign each screen dot one of two colors. Bit mapping is a technique that stores a bit in memory for each dot on the screen. In standard bit map mode, if the bit in memory is turned off, the corresponding dot on the screen becomes the color of the background. If the bit in memory is turned on, the corresponding dot on the screen becomes the color of the foreground image. The series of 64,000 dots on the screen and 64,000 corresponding bits in memory control the image you see on the screen. Most of the finely detailed computer graphics you see in demonstrations and video games are bit mapped high-resolution graphics.

Multi-color bit map mode is a combination of standard bit map mode and multi-color character mode. You can display each screen dot in one of four colors within an 8 × 8 character grid. Again, as in multi-color character mode, there is a tradeoff between the horizontal resolution and color control.

## SPLIT SCREEN DISPLAY

The third type of screen display, split screen, is a combination of the first two types. The split-screen display outputs part of the screen as text and part in bit map mode (either standard or multi-color). The C128 is capable of this since it depends on two parts of its memory to store the two screens: one part for text, and the other for graphics.

# COMMAND SUMMARY

Following is a brief explanation of each graphics command available in BASIC 7.0:

**BOX:** Draws rectangles on the bit-map screen
**CHAR:** Displays characters on the bit-map screen
**CIRCLE:** Draws circles, ellipses and other geometric shapes
**COLOR:** Selects colors for screen border, foreground, background and characters
**DRAW:** Displays lines and points on the bit-map screen
**GRAPHIC:** Selects a screen display (text, bit map or split-screen bit map)
**GSHAPE:** Gets data from a string variable and places it at a specified position on the bit-map screen
**LOCATE:** Positions the bit-map pixel cursor on the screen
**PAINT:** Fills area on the bit-map screen with color
**SCALE:** Sets the relative size of the images on the bit-map screen
**SSHAPE:** Stores the image of a portion of the bit-map screen into a text-string variable
**WIDTH:** Sets the width of lines drawn

The following paragraphs give the format and examples for each of the non-sprite BASIC 7.0 graphic commands. For a full explanation of each of these commands, see the BASIC 7.0 Encyclopedia in Chapter 2.

## BOX

Draw a box at a specified position on the screen.

**BOX [color source], X1, Y1[,X2,Y2][,angle][,paint]**

where:

| | |
|---|---|
| **color source** | 0 = Background color<br>1 = Foreground color<br>2 = Multi-color 1<br>3 = Multi-color 2 |
| **X1, Y1** | Top left corner coordinate (scaled) |
| **X2, Y2** | Bottom right corner opposite X1, Y1, is the pixel cursor location (scaled) |
| **angle** | Rotation in clockwise degrees; default is 0 degrees |
| **paint** | Paint shape with color<br>0 = Do not paint<br>1 = Paint |

## EXAMPLES:

```
10 COLOR 0,1:COLOR 1,6:COLOR 4,1
20 GRAPHIC 1,1:REM SELECT BMM
30 BOX 1,10,10,70,70,90,1:REM DRAW FILLED GREEN BOX
40 FOR I=20 TO 140 STEP 3
50 BOX 1,I,I,I+60,I+60,I+80:REM DRAW AND ROTATE BOXES
60 NEXT
70 BOX 1,140,140,200,200,220,1:REM DRAW 2ND FILLED GREEN BOX
80 COLOR 1,3:REM SWITCH TO RED
90 BOX 1,150,20,210,80,90,1:REM DRAW FILLED RED BOX
100 FOR I=20 TO 140 STEP 3
110 BOX 1,I+130,I,I+190,I+60,I+70:REM DRAW AND ROTATE RED BOXES
120 NEXT
130 BOX 1,270,140,330,200,210,1:REM DRAW 2ND FILLED RED BOX
140 SLEEP 5  :REM DELAY
150 GRAPHIC 0,1:REM SWITCH TO TEXT MODE
```

```
10 COLOR 0,1:COLOR 4,1:COLOR 1,6
20 GRAPHIC 1,1
30 BOX 1,0,0,319,199
40 FOR X=10 TO 160 STEP 10
50 C=X/10
60 COLOR 1,C
70 BOX 1,X,X,320-X,320-X
80 NEXT
90 SLEEP 5
100 GRAPHIC 0,1
```

```
10 COLOR 0,1:COLOR 4,1:COLOR 1,6
20 GRAPHIC 1,1
30 BOX 1,50,50,150,120
40 BOX 1,70,70,170,140
50 DRAW 1,50,50 TO 70,70
60 DRAW 1,150,120 TO 170,140
70 DRAW 1,50,120 TO 70,140
80 DRAW 1,150,50 TO 170,70
90 CHAR 1,20,20,"CUBE EXAMPLE"
100 SLEEP 5
110 GRAPHIC 0,1
```

```
10 COLOR 1,6:COLOR 4,1:COLOR 0,1
20 GRAPHIC 1,1:REM SELECT BIT MAP MODE
30 DO  :REM CALCULATE RANDOM POINTS
40 X1=INT(RND(1)*319+1)
50 X2=INT(RND(1)*319+1)
60 X3=INT(RND(1)*319+1)
70 X4=INT(RND(1)*319+1)
80 Y1=INT(RND(1)*199+1)
90 Y2=INT(RND(1)*199+1)
100 Y3=INT(RND(1)*199+1)
110 Y4=INT(RND(1)*199+1)
120 BOX 1,X1,Y1,X2,Y2:REM DRAW THE RANDOM BOXES
130 BOX 1,X3,Y3,X4,Y4
140 DRAW 1,X1,Y1 TO X3,Y3:REM CONNECT THE POINTS
150 DRAW 1,X2,Y2 TO X4,Y4
160 DRAW 1,X1,Y2 TO X3,Y4
170 DRAW 1,X2,Y1 TO X4,Y3
180 SLEEP2:REM DELAY
190 SCNCLR
200 LOOP:REM LOOP CONTINUOUSLY
```

## CHAR

Display characters at the specified position on the screen.

**CHAR [color source],X,Y[,string][,RVS]**

This is primarily designed to display characters on a bit mapped screen, but it can also be used on a text screen. Here's what the parameters mean:

| | |
|---|---|
| color source | 0 = Background<br>1 = Foreground |
| X | Character column (0–79) (wraps around to the next line in 40-column mode) |
| Y | Character row (0–24) |
| string | String to print |
| RVS | Reverse field flag (0 = off, 1 = on) |

### EXAMPLE:

```
10 COLOR 2,3: REM multi-color 1 = Red
20 COLOR 3,7: REM multi-color 2 = Blue
30 GRAPHIC 3,1
40 CHAR 0,10,10, "TEXT",0
```

## CIRCLE

Draw circles, ellipses, arcs, etc. at specified positions on the screen.

**CIRCLE [color source],X,Y[,Xr][,Yr]**
**[,sa][,ea][,angle][,inc]**

where:

| | |
|---|---|
| color source | 0 = background color<br>1 = foreground color<br>2 = multi-color 1<br>3 = multi-color 2 |
| X,Y | Center coordinate of the CIRCLE |
| Xr | X radius (scaled) |
| Yr | Y radius (default is xr) |
| sa | Starting arc angle (default 0 degrees) |
| ea | Ending arc angle (default 360 degrees) |
| angle | Rotation in clockwise degrees (default is 0 degrees) |
| inc | Degrees between segments (default is 2 degrees) |

### EXAMPLES:

| | |
|---|---|
| CIRCLE1, 160,100,65,10 | Draws an ellipse. |
| CIRCLE1, 160,100,65,50 | Draws a circle. |
| CIRCLE1, 60,40,20,18,,,,45 | Draws an octagon. |

| | |
|---|---|
| CIRCLE1, 260,40,20,,,,,90 | Draws a diamond. |
| CIRCLE1, 60,140,20,18,,,,120 | Draws a triangle. |
| CIRCLE1, +2, +2,50,50 | Draws a circle (two pixels down and two to the right) relative to the original coordinates of the pixel cursor. |

### SAMPLE PROGRAMS:

```
10 REM SUBMARINE TRACKING SYSTEM
20 COLOR 0,1:COLOR 4,1:COLOR 1,2:REM SELECT BKGRND, BRDR,SCREEN COLORS
30 GRAPHIC 1,1:REM ENTER BIT MAP MODE
40 BOX 1,0,0,319,199
50 CHAR 1,7,24,"SUBMARINE TRACKING SYSTEM" :REM DISPLAY CHARS ON BIT MAP
60 COLOR 1,3:REM SELECT RED
70 XR=0:YR=0:REM INIT X AND Y RADIUS
80 DO
90 CIRCLE 1,160,100,XR,YR,0,360,0,2:REM DRAW CIRCLES
100 XR=XR+10:YR=YR+10:REM UPDATE RADIUS
110 LOOP UNTIL XR=90
120 DO
130 XR= 0:YR= 0
140 DO
150 CIRCLE 0,160,100,XR,YR,0,360,0,2 :REM ERASE CIRCLE
160 COLOR 1,2 :REM SWITCH TO WHITE
170 DRAW 1,160,100+XR:DRAW 0,160,100+XR:REM DRAW SUBMARINE BLIP
180 COLOR 1,3:REM SWITCH BACK TO RED
190 SOUND 1,16000,15:REM BEEP
200 CIRCLE 1,160,100,XR,YR,0,360,0,2 :REM DRAW CIRCLE
210 XR=XR+10:YR=YR+10 :REM UPDATE RADIUS
220 LOOP UNTIL XR=90 :REM LOOP
230 LOOP
```

```
10 COLOR 0,1:COLOR 4,1:COLOR 1,7
20 GRAPHIC 1,1:REM SELECT BMM
30 X=150:Y= 150:XR=150:YR=150
40 DO
50 CIRCLE 1,X,Y,XR,YR
60 X=X+7 :Y=Y-5:REM INCREMENT X AND Y COORDINATES
70 XR=XR-5 :YR=YR-5:REM DECREMENT X AND Y RADII
80 LOOP UNTIL XR=0
90 GRAPHIC 0,1:REM SELECT TEXT MODE
```

## COLOR

Define colors for each screen area.

**COLOR source number, color number**

This statement assigns a color to one of the seven color areas:

| AREA | SOURCE |
|---|---|
| **0** | **40-column (VIC) background** |
| **1** | **40-column (VIC) foreground** |
| **2** | **multi-color 1** |
| **3** | **multi-color 2** |
| **4** | **40-column (VIC) border** |
| **5** | **character color (40- or 80-column screen)** |
| **6** | **80-column background color** |

**Colors codes are in the range 1-16.**

| COLOR CODE | COLOR | COLOR CODE | COLOR |
|---|---|---|---|
| 1 | Black | 9 | Orange |
| 2 | White | 10 | Brown |
| 3 | Red | 11 | Light Red |
| 4 | Cyan | 12 | Dark Gray |
| 5 | Purple | 13 | Medium Gray |
| 6 | Green | 14 | Light Green |
| 7 | Blue | 15 | Light Blue |
| 8 | Yellow | 16 | Light Gray |

**Color Codes in 40-Column (VIC) Output**

### EXAMPLE:

COLOR 0, 1: Changes background color of 40-column screen to black.
COLOR 5, 8: Changes character color to yellow.

### SAMPLE PROGRAM:

```
10 REM CHANGE FOREGROUND BIT MAP COLOR
20 GRAPHIC 1,1
30 I=1
40 DO
50 COLOR 1,I
60 BOX 1,100,100,219,159
70 I=I+1:SLEEP 1
80 LOOP UNTIL I=17
90 GRAPHIC 0,1
100 REM CHANGE BORDER COLOR
110 I=1
120 DO
130 COLOR 4,I
140 I=I+1:SLEEP 1
150 LOOP UNTIL I=17
160 REM CHANGE CHARACTER COLOR
170 I=1
180 DO
190 COLOR 5,I
200 PRINT"COLOR CODE";I
210 I=I+1:SLEEP 1
220 LOOP UNTIL I=17
230 REM CHANGE BACKGROUND COLOR
240 I=1
250 DO
260 COLOR 0,I
270 I=I+1:SLEEP 1
280 LOOP UNTIL I=17
290 COLOR 0,1:COLOR 4,1:COLOR 5,2
```

## DRAW

Draw dots, lines and shapes at specified positions on screen.

**DRAW [color source], [X1, Y1][TO X2, Y2] . . .**

Here are the parameter values:

where:

| | |
|---|---|
| **color source** | 0 Bit map background |
| | 1 Bit map foreground |
| | 2 Multi-color 1 |
| | 3 Multi-color 2 |
| **X1,Y1** | Starting coordinate (0,0 through 319,199) (scaled) |
| **X2,Y2** | Ending coordinate (0,0 through 319,199) (scaled) |

## EXAMPLES:

| | |
|---|---|
| DRAW 1, 100, 50 | Draw a dot. |
| DRAW , 10, 10 TO 100,60 | Draw a line. |
| DRAW , 10, 10 TO 10,60 TO 100,60 TO 10,10 | Draw a triangle. |

## SAMPLE PROGRAMS:

```
10 REM DRAW EXAMPLES
20 COLOR 0,1:COLOR 4,1:COLOR 1,6
30 GRAPHIC 1,1
40 CHAR 1,10,1,"THE DRAW COMMAND"
50 X=10
60 DO
70 DRAW 1,X,50:REM DRAW POINTS
80 X=X+10
90 LOOP UNTIL X=320
100 CHAR 1,12,7 ,"DRAWS POINTS"
110 Y=70
120 DO
130 Y=Y+5
140 DRAW 1,1,Y TO Y,Y :REM DRAW LINES
150 LOOP UNTIL Y=130
160 CHAR 1,18,11,"LINES"
170 DRAW 1,10,140 TO 10,199 TO 90,165 TO 40,160 TO 10,140:REM DRAW SHAPE 1
180 DRAW 1,120,145 TO 140,195 TO 195,195 TO 225,145 TO 120,145:REM DRAW SHAPE
190 DRAW 1,250,199 TO 319,199 TO 319,60 TO 250,199:REM DRAW SHAPE 3
200 CHAR 1,22,15,"AND SHAPES"
210 SLEEP 5:GRAPHIC 0,1

10 COLOR 0,1:COLOR 4,1:COLOR 1,7
20 GRAPHIC 1,1:REM SELECT BMM
30 Y=1
40 DO
50 DRAW 1,1,Y TO 320,Y:REM DRAW HORIZONTAL LINES
60 Y=Y+10
70 LOOP WHILE Y<200
75 X=1
80 DO
90 DRAW 1,X,1 TO X,200:REM DRAW VERTICAL LINES
95 X=X+10
97 LOOP WHILE X<320
100 COLOR 1,3:REM SWITCH TO RED
110 DRAW 1,160,0 TO 160,200:REM DRAW X AXIS IN RED
120 DRAW 1,0,100 TO 320,100:REM DRAW Y AXIS IN RED
130 COLOR 1,6:REM SWITCH TO GREEN
140 DRAW 1,0,199 TO 50,100 TO 90,50 TO 110,30 TO 150,20 TO 180,30
150 DRAW 1,180,30 TO 220,10 TO 260,80 TO 320,0:REM DRAW GROWTH CURVE
160 CHAR 1,7,23,"PROJECTED SALES THROUGH 1990"
170 CHAR 1,1,21,"1970     1975     1980     1985     1990"
180 SLEEP 10:GRAPHIC 0,1:REM DELAY AND SWITCH TO TEXT MODE
```

## GRAPHIC

Select a graphic mode.

1) **GRAPHIC mode [,clear][,s]**
2) **GRAPHIC CLR**

This statement puts the Commodore 128 in one of the six graphic modes:

| MODE | DESCRIPTION |
|---|---|
| **0** | **40-column text** |
| **1** | **standard bit map graphics** |
| **2** | **standard bit map graphics (split screen)** |
| **3** | **multi-color bit map graphics** |
| **4** | **multi-color bit map graphics (split screen)** |
| **5** | **80-column text** |

### EXAMPLES:

| | |
|---|---|
| GRAPHIC 1,1 | Select standard bit map mode and clear the bit map. |
| GRAPHIC 4,0,10 | Select split screen multi-color bit map mode, do not clear the bit map and start the split screen at line 10. |
| GRAPHIC 0 | Select 40-column text. |
| GRAPHIC 5 | Select 80-column text. |
| GRAPHIC CLR | Clear and deallocate the bit map screen. |

### SAMPLE PROGRAM:

```
10 REM GRAPHIC MODES EXAMPLE
20 COLOR 0,1:COLOR 4,1:COLOR 1,7
30 GRAPHIC 1,1:REM ENTER STND BIT MAP
40 CIRCLE 1,160,100,60,60
50 CIRCLE 1,160,100,30,30
60 CHAR 1,9,24,"STANDARD BIT MAP MODE"
70 SLEEP 4
80 GRAPHIC 0,1:REM ENTER STND CHAR MODE
90 COLOR 1,6:REM SWITCH TO GREEN
100 FOR I=1TO 25
110 PRINT"STANDARD CHARACTER MODE"
120 NEXT
130 SLEEP 4
140 GRAPHIC 2,1:REM SELECT SPLIT SCREEN
150 CIRCLE 1,160,70,50,50
160 CHAR 1,14,1,"SPLIT SCREEN"
170 CHAR 1,8,16,"STANDARD BIT MAP MODE ON TOP"
180 FOR I=1 TO 25
190 PRINT" STANDARD CHARACTER MODE ON THE BOTTOM"
200 NEXT
210 SLEEP 3:REM DELAY
220 SCNCLR:REM CLEAR SCREEN
230 GRAPHIC CLR:REM DE-ALLOCATE BIT MAP
```

## GSHAPE

Retrieve (load) the data from a string variable and display it on a specified coordinate.

**GSHAPE string variable [X,Y][,mode]**

where:

| | |
|---|---|
| **string** | Contains shape to be drawn |
| **X,Y** | Top left coordinate (0,0 through 319,199) telling where to draw the shape (scaled—the default is the pixel cursor) |
| **mode** | Replacement mode:<br>0 = place shape as is (default)<br>1 = invert shape<br>2 = OR shape with area<br>3 = AND shape with area<br>4 = XOR shape with area |

### SAMPLE PROGRAM:

```
10 REM DRAW, SAVE AND GET THE COMMODORE SYMBOL
20 COLOR 0,1:COLOR 4,1:COLOR 1,7
30 GRAPHIC 1,1:REM SELECT BMM
40 CIRCLE 1,160,100,20,15:REM OUTER CIRCLE
50 CIRCLE 1,160,100,10,9:REM INNER CIRCLE
60 BOX 1,165,85,185,115:REM ISOLATE AREA TO BE ERASED
70 SSHAPE A$,166,85,185,115:REM SAVE THE AREA INTO A$
80 GSHAPE A$,166,85,4:REM EXCLUSIVE OR THE AREA-THIS (ERASES) TURNS OFF PIXELS
90 DRAW 0,165,94 TO 165,106:REM TURN OFF (DRAW IN BKGRND COLOR) PIXELS IN "C="
100 DRAW 1,166,94 TO 166,99 TO 180,99  TO 185,94 TO 166,94:REM UPPER FLAG
110 DRAW 1,166,106 TO 166,101 TO 180,101 TO 185,106 TO 166,106:REM LOWER FLAG
120 PAINT 1,160,110:REM PAINT "C"
130 PAINT 1,168,98 :REM UPPER FLAG
140 SLEEP 5:REM DELAY
150 SSHAPE B$,137,84,187,116:REM SAVE SHAPE INTO B$
160 DO
170 SCNCLR
180 Y=10
190 DO
200 X=10
210 DO
220 GSHAPE B$,X,Y:REM GET AND DISPLAY SHAPE
230 X=X+50:REM UPDATE X
240 LOOP WHILE X<280
250 Y=Y+40:REM UPDATE Y
260 LOOP WHILE Y<160
270 SLEEP 3
280 LOOP
```

## LOCATE

Position the bit map pixel cursor (PC) on the screen.

**LOCATE X, Y**

### EXAMPLE:

| | |
|---|---|
| LOCATE 160,100 | Position the PC in the center of the bit map screen. Nothing will be seen until something is drawn. |
| LOCATE +20,100 | Move the PC 20 pixels to the right of the last PC position and place it at Y coordinate 100. |
| LOCATE +30,+20 | Move the PC 30 pixels to the right and 20 down from the previous PC position. |

## PAINT

Fill area with color.

**PAINT [color source],X,Y[,mode]**

where:

| | |
|---|---|
| **color source** | 0 Bit map background |
| | 1 Bit map foreground (default) |
| | 2 Multi-color 1 |
| | 3 Multi-color 2 |
| **X,Y** | starting coordinate, scaled (default at pixel cursor (PC)) |
| **mode** | 0 = paint an area defined by the color source selected |
| | 1 = paint an area defined by any non-background source |

**EXAMPLE:**

| | |
|---|---|
| 10 CIRCLE 1, 160,100,65,50 | Draws an outline of a circle. |
| 20 PAINT 1, 160,100 | Fills in the circle with color from source 1 (VIC foreground), assuming point 160,100 is colored in the background color (source 0). |
| 10 BOX 1, 10, 10, 20, 20 | Draws an outline of a box. |
| 20 PAINT 1, 15, 15 | Fills the box with color from source 1, assuming point 15,15 is colored in the background source (0). |
| 30 PAINT 1, +10, +10 | PAINT the screen in the foreground color source at the coordinate relative to the pixel cursor's previous position plus 10 in both the vertical and horizontal positions. |

## SCALE

Alter scaling in graphics mode.

**SCALE n [,Xmax,Ymax]**

where:

**n** = 1 (on) or 0 (off)
**X max** = 320–32767
(default = 1023)
**Y max** = 200–32767
(default = 1023)

The default scale values are:

| | |
|---|---|
| Multi-color mode | X = 0 to 159 Y = 0 to 199 |
| Bit map mode | X = 0 to 319 Y = 0 to 199 |

**EXAMPLES:**

| | |
|---|---|
| 10 GRAPHIC 1,1 | Enter standard bit map, turn scaling on to default size (1023, 1023) and draw a circle. |
| 20 SCALE 1:CIRCLE 1,180,100,100,100 | |

| | |
|---|---|
| 10 GRAPHIC 1,3 | Enter multi-color mode, turn scaling |
| 20 SCALE 1,1000,5000 | on to size (1000,5000) and draw a |
| 30 CIRCLE 1,180,100,100,100 | circle. |

## SSHAPE

Save shapes to string variables.

**SSHAPE** and **GSHAPE** are used to save and load rectangular areas of multi-color or bit mapped screens to/from BASIC string variables. The command to save an area of the screen into a string variable is:

**SSHAPE string variable, X1, Y1 [,X2,Y2]**

where:

| | |
|---|---|
| **string variable** | String name to save data in |
| **X1,Y1** | Corner coordinate (0,0 through 319,199) (scaled) |
| **X2,Y2** | Corner coordinate opposite (X1,Y1) (default is the PC) |

### EXAMPLES:

| | |
|---|---|
| SSHAPE A$,10,10 | Saves a rectangular area from the coordinate 10,10 to the location of the pixel cursor, into string variable A$. |
| SSHAPE B$, 20,30,47,51 | Saves a rectangular area from top left coordinate (20,30) through bottom right coordinate (47,51) into string variable B$. |
| SSHAPE D$, +10, +10 | Saves a rectangular area 10 pixels to the right and 10 pixels down from the current position of the pixel cursor. |

Also, see the example program under GSHAPE for another example.

## WIDTH

Set the width of drawn lines.

**WIDTH n**

This command sets the width of lines drawn using BASIC's graphic commands to either single or double width. Giving n a value of 1 defines a single width line; a value of 2 defines a double width line.

### EXAMPLES:

WIDTH 1 Set Single width for graphic commands
WIDTH 2 Set double width for drawn lines

# 5

# MACHINE LANGUAGE ON THE COMMODORE 128

This chapter introduces you to 6502-based *machine language programming*. Read this section if you are a beginner or novice machine language programmer. This section explains the elementary principles behind programming your Commodore 128 in machine language. It also introduces you to the 8502 machine language instruction set and how to use each instruction. If you are already an experienced machine language programmer, skip this section and continue to the 8502 Instruction and Addressing Table at the end of the chapter for reference material on machine language instructions. *The 8502 instruction set is exactly the same as the 6502 microprocessor instruction set.*

## WHAT IS MACHINE LANGUAGE?

Every computer has its own machine language. The type of machine language depends on which processor is built into the computer. Your Commodore 128 understands 8502 machine language, which is based on 6502 machine language, to carry out its operations. Think of the microprocessor as the brain of the computer and the instructions as the thoughts of the brain.

Machine language is the most elementary level of code that the computer actually interprets. True machine language is composed of binary strings of zeroes and ones. These zeroes and ones act as switches to the hardware, and tell the circuit where to apply voltage levels.

The machine language discussed in this chapter is symbolic 6502 Assembly language as it appears in the C128 Machine Language Monitor. This is not the full-blown symbolic assembly language as it appears in an Assembler package, since symbolic addresses or other higher level utilities that an Assembler software package would provide are not implemented.

Machine language is the lowest level language in which you can instruct your computer. BASIC is considered a high-level language. Although your Commodore 128 has BASIC built in, the computer must first interpret and translate it to a lower level that it can understand, before the computer can act upon BASIC instructions.

With each microinstruction, you give the computer a specific detail to perform. The computer takes nothing for granted in machine language, unlike BASIC, where many unnoticed machine-level functions are performed by one statement. One BASIC statement requires several machine language instructions to perform the same operation. Actually, when you issue a BASIC command, you are really calling a machine language subroutine that performs a computer operation.

## WHY USE MACHINE LANGUAGE?

If machine language is more intricate and complicated than BASIC, why use it? Certain applications, such as graphics and telecommunications, require machine language because of its speed. Since the computer does not have to translate from a higher-level language, it runs many times faster than BASIC.

Programs such as those used in arcade games cannot operate in the relatively slow speed of BASIC, so they are written in machine language. Other instances dictate the use of machine language simply because those programming operations are handled better than in a high-level language like BASIC. But some programming functions such as string operations are easier in BASIC than in machine language. In these cases, BASIC and machine language can be used together. You can find information on how to mix machine language with BASIC in Chapter 7.

Inside your computer is a perpetually running program called the operating system. The operating system program controls every function of your computer. It performs functions at lightning speeds you are not even aware of.

The operating system program is written entirely in machine language and is stored in a portion of the computer called the Kernal ROM. (Chapter 13 describes how to take advantage of the machine language programs within the Kernal, and how to use parts of the operating system in your own machine-language programs.)

Though machine language programming may seem more complicated and difficult than BASIC at first, think back to when you didn't know BASIC or your first programming language. That seemed difficult at first, too. If you learned BASIC or another programming language, you can learn machine language. Although it's a good idea to learn a higher-level language such as BASIC before you start machine language, it's not absolutely necessary.

# WHAT DOES MACHINE LANGUAGE LOOK LIKE?

Chapter 2 describes the C128 BASIC 7.0 language. Most statements in BASIC start with a BASIC verb or keyword, followed by an operand. The BASIC *keywords* resemble English verbs. The *operands* are variables, or constants, that are part of an expression. For example, A + B = 2, is an expression where A, B, and 2 are operands in the expression. Machine-language instructions are similar, though they have a uniform format. Here's the format for an 8502 symbolic machine language instruction as it appears in the C128 Machine Language Monitor:

OP-CODE FIELD OPERAND FIELD

## OPERATION CODE (OP-CODE) FIELD

The first part of a machine-language instruction is called the *operation code* or op-code. The op-code is comparable to a BASIC verb, in that it is the part of the instruction that performs an action. A machine language op-code is also referred to in an assembly language as a mnemonic. All 8502 (6502) machine language assembler mnemonics are three-letter abbreviations for the functions they perform. For example, the first and most common instruction you will learn is LDA, which stands for LoaD the Accumulator. This chapter defines all of the mnemonics.

## OPERAND FIELD

The second portion of a machine-language instruction is the OPERAND field. In the C128 Machine Language Monitor, the operand is separated from the op-code with at least one space and preceded by a ($) dollar sign, (+) plus sign (decimal), (&) ampersand (octal), or a (%) percent (binary) sign to signify that the operand is a hexadecimal, decimal, octal or binary number. An ADDRESS is the name of or reference to a specific memory location within the computer.

The number of a memory location is its address, just like houses on your street are numbered. Addresses in your computer are necessary so they can receive, store and send (LOAD) data back and forth to the microprocessor.

When you use the Commodore 128's built-in machine-language monitor, all numbers and addresses default to hexadecimal numbers, but they can be represented in decimal, octal or binary. The address is the hexadecimal number of the specified memory location. When you use an ASSEMBLER, the addresses are referred to as symbolic addresses. Symbolic addresses allow you to use variable names, instead of absolute addresses that specify the actual memory location. You declare the symbolic address to be the numeric address in the beginning of your machine language program or allow the assembler to assign the address.

When you refer to that address later in the program, you can refer to the symbolic address rather than to the absolute address as does the Machine Language Monitor. Using an assembler and symbolic addresses make programming in machine language easier than using the machine-language monitor and absolute addresses. You will learn about the eleven addressing modes later in this chapter.

As you know, the second part of a machine-language instruction is the OPERAND. A machine language operand can be a constant; it does not necessarily have to be an address reference. When a constant in machine language appears in place of an address as the second part of an instruction, an operation is performed on a data value rather than a memory location.

A pound sign (#) in front of the operand signifies immediate addressing, which you will learn more about later in the chapter. The pound sign is only used as an aid for the symbolic language programmer. The pound sign tells the computer to perform machine-language instruction on a constant, and not an address. In the case of the Machine Language Monitor, variable names are not allowed. To represent variables in the monitor, you must reference a memory location where your variable data value is stored.

## EXAMPLES OF MACHINE-LANGUAGE INSTRUCTIONS

```
LDA $100      ; Absolute addressing
LDA $10       ; Zero page absolute addressing
LDA ($FA),Y   ; Indirect indexed addressing
LDA $2000,X   ; Indexed addressing (absolute)
LDA #$10      ; Immediate addressing (constants)
```

## THE SIMILARITIES AND DIFFERENCES BETWEEN AN ASSEMBLER AND A MACHINE LANGUAGE MONITOR

An *assembler* and *machine-language monitor* both provide for symbolic op-codes. Assemblers typically allow symbolic operands as well, whereas the C128 machine-language monitor refers to addresses and operands literally (*absolutely*).

An assembler typically has two forms of a file: source code and object code. Source code is the file you create when you are writing the program including symbolic start addresses and comments.

The source code file is not executable. It must be assembled (in an intermediate process) into object code. which is executable code.

The machine language monitor start address is determined by where you place the actual instructions in memory. The monitor does not provide for comments. The resulting program, once it is input, is executed immediately as a binary file. No intermediate assembly step is needed.

# THE 8502 MICROPROCESSOR REGISTERS

You have learned that an address is a reference to a specific memory location among the 2 banks of RAM within the Commodore 128. Separate and independent of those RAM locations are special purpose work and storage areas within the microprocessor chip itself, called registers. These registers are where the values are manipulated. The manipulation of the microprocessor registers and their communication with the computer's memory (RAM and ROM) accomplishes all the functions of machine language and your computer's operating system.

Figure 5–1 shows a block diagram of the 8502 microprocessor. As shown in the figure, the 8502 microprocessor registers are:

Accumulator
X index register
Y index register
Status register
Program counter
Stack pointer

Following are descriptions of these registers.

## THE ACCUMULATOR

The accumulator is one of the most important registers within the 8502 microprocessor. As the name implies, it accumulates the results of specific operations. Think of the accumulator as the doorway to your microprocessor. All information that enters your computer must first pass through the accumulator (or the X or Y register).

For example, if you want to store a value within one of the RAM locations, you must first load the value into the accumulator (or the X or Y register) and then store it into the specified RAM location. You cannot store a value directly into RAM, without placing it into the accumulator or the index registers first. (The index registers are described in the following section.)

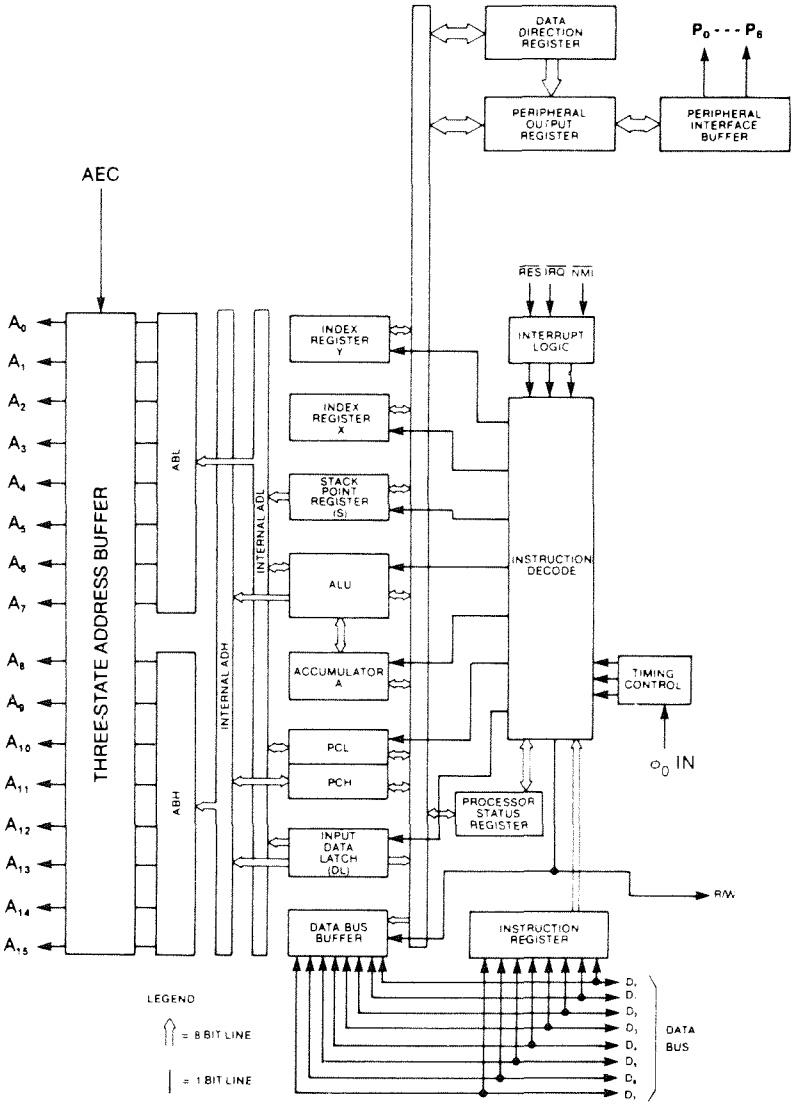



**Figure 5–1. 8502 Block Diagram**

All mathematical operations are performed within the arithmetic logic unit (ALU) and stored in the accumulator. It is considered a temporary mathematical work area. For example, you want to add two numbers, 2 + 3. First, load the accumulator with the 2. Next add 3 with the ADC mnemonic. Now, you want to perform another operation. You must store the answer from the accumulator into a RAM location before you perform the next math operation. If you don't, your original answer is erased.

The accumulator is so important that it has an addressing mode of its own. All the instructions using this mode pertain specifically to the accumulator. The following three sample instructions pertain solely to the accumulator in its own addressing mode:

LDA - LOAD accumulator with memory
STA - STORE the accumulator in memory
ADC - ADD contents of memory to the accumulator

Details on all of the accumulator addressing commands are given later in this chapter.

## THE X AND Y INDEX REGISTERS

The second most used registers are the *X* and *Y index registers*. These index registers are used primarily to modify an address by adding an index within a machine-language instruction. They also can be used as temporary storage locations or to load values and store them in RAM like the accumulator.

When modifying an address, the contents of the index registers are added to an original address, called the base address, to find an address relative to the base address. The resulting address yields the effective address—i.e., the location where a data value is stored or retrieved. The effective address is acted upon by machine-language instructions. For example, you want to place the value 0 in locations 1024 through 1034. In BASIC, here's how you do it:

```
10 FOR I = 1024 to 1034
20 POKE I,0
30 NEXT
```

Here's how you do it in symbolic machine language by using the X or Y index register. NOTE: Don't worry if you don't understand all of the following instructions. They are discussed fully in the TYPES OF INSTRUCTIONS section, later in this chapter.

```
        LDA #$00      Load the Accumulator with 0
        TAX           Transfer the contents of Accumulator (0) to X
                      Register.
START   STA $0400,X   Store contents of Accumulator in address $0400 + X
        INX           Increment the X register
        CPX #$0B      Compare the X register with $0B (11 decimal)
        BNE START     If X register does not equal 11 branch to START.
        BRK           Stop
```

* = In the machine-language monitor the symbolic label START is not allowed, so it would appear as an absolute address reference (eg; $183B).

The BASIC example above places a 0 in locations (addresses) 1024 through 1034. Line 10 sets up a loop from memory locations 1024 to 1034. Line 20 POKEs the value 0 into the location specified by I. The first time through the loop, I equals 1024. The second time through the loop, I equals 1025 and so on. Line 30 increments the index variable I by I each time it is encountered.

The previous machine-language example accomplishes the same task as the BASIC example. LDA #$00 loads a 0 into the accumulator. TAX transfers the contents of the accumulator into the X-index register. The following machine-language instructions form a loop:

```
START   STA $0400,X
        INX
        CPX #$0B
        BNE START
```

Here's what happens within the loop. STA $0400,X stores a 0 in location $0400 (hex) the first time through the loop. Location $0400 is location 1024 decimal. INX increments the X register by 1, each cycle through the loop. CPX #$0B compares the contents of the X register with the constant 11 ($0B). If the contents of the X register do not equal 11, the program branches back to START STA $0400,X and the loop is repeated.

The second time through the loop, 0 is stored in address $0401 (1025 decimal) and the X register is incremented again. The program continues to branch until the contents of the X register equal 11.

The effective address for the first cycle through the loop is $0400 which is 1024 decimal. For the second cycle through the loop the effective address is $0400 + 1, and so on. Now you can see how the index registers modify the address within machine-language instruction.

## THE STATUS REGISTER

The microprocessor's *status register* indicates the status of certain conditions within the 8502. The status register is controlled by seven programming states of the microprocessor, and indicates the conditions with flags. The status register is one byte, so each flag is represented by a single bit. Bit 5 is not implemented.

Branching instructions check (4 of the 7 bits in) the status register to determine whether a condition has occurred. The conditions for branching pertain to the value of the bits in the status register. If a condition is true, meaning the FLAG bit corresponding to one of the four conditions is high (equal to a 1), the computer branches. If the condition you are testing is not true, the computer does not branch and the program resumes with the instruction immediately following the branch.

Figure 5–2 shows the layout of the 8502 status register and lists the conditions the status register flags.

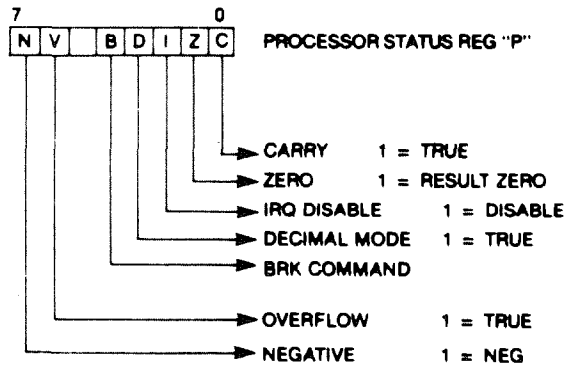



**Figure 5–2. 8502 Status Register**

The Carry bit (0) is set if an addition operation carries a bit into the next position to the left of the leftmost bit. The Carry bit is set by other conditions, of which this is one. The SEC instruction sets the Carry bit. CLear the Carry bit with the CLC instruction.

The Zero bit (1) is set if the result of an operation equals zero. The command BEQ stands for Branch on result EQual to Zero. The command BNE stands for Branch on Result Not Equal to zero. If the zero bit in the status register is set, the program branches to the address relative to the current program counter value (for a BEQ instruction). Otherwise, the BEQ command is skipped and the program resumes with the instruction immediately following the BEQ statement.

The IRQ Disabled bit (2) is set if your program requests interrupts to be disabled with the SEI command (Set Interrupt Disable Status). The Disable Interrupt Status bit is cleared with the CLI command (CLear Interrupt Disable bit) to permit interrupts to occur. You will learn more about programming interrupts in the section entitled TYPES OF INSTRUCTIONS and in the Raster Interrupt program explanation in Chapter 8.

The microprocessor sets the Decimal Mode bit (3) if you instruct the microprocessor to SEt Decimal Mode with the SED instruction. CLear the Decimal Mode bit with the CLD instruction, CLear Decimal Mode.

The BRK flag (bit 4) operates similar to the IRQ disable flag. If a BRK instruction occurs, it is set to 1. Like an IRQ interrupt, the BRK causes the contents of the program counter to be pushed onto the stack. The contents of the status register is pushed on top of the stack and evaluated. If the BRK flag is set, the operating system or your application program must evaluate whether or not a BRK or interrupt has occurred.

If the BRK flag is cleared once the status register is pushed onto the stack, the processor handles this as an interrupt and services it. Unlike an interrupt, the BRK flag causes the address of the program counter plus two to be saved. The microprocessor expects this to be the address of the next instruction to be executed. You may have to

adjust this address since it may not be the actual address of the next instruction within your program.

The Overflow flag (bit 6) is set by a signed operation overflowing into the sign bit (bit 7) of the status register. You can clear the Overflow bit in the status register with the CLV instruction (CLear Overflow flag). You can conditionally branch if the Overflow bit is set with the BVS (Branch Overflow Set) instruction. Similarly, you can conditionally branch if the overflow bit is clear with the BVC (Branch Overflow Clear) instruction. The BIT instruction can be used to intentionally set the overflow flag.

The microprocessor sets the negative flag (bit 7) if the result of an arithmetic operation is less than 0. You can conditionally branch if the result of an arithmetic operation is negative, using the BMI instruction, (Branch on result MInus) or positive using the BPL instruction, (Branch on Result Positive).

The status register indicates seven important conditions within the microprocessor while your machine language program is executing. Your program can test for certain conditions, and act upon the results. It gives you a way to conditionally control certain machine level functions depending on the value of the status flags.

## THE PROGRAM COUNTER

So far all of the registers within the 8502 are 8 bits, or one byte. The *program counter* is twice as wide (16 bits) as the accumulator, X or Y registers or the status register. The program counter is a 16-bit register because it holds the current address of the next instruction to be executed. The addresses used in an 8502-based microprocessor are all 16 bits wide. They have to be in order to address all locations within each 64K RAM bank.

The program counter holds the address of the next instruction to be executed. It fetches the addresses of the instructions sequentially (usually) and places them on the 16-bit address bus. The processor obtains the data or instructions at the specified 16-bit address from the data bus. Then they are decoded and executed.

## THE STACK POINTER

Within the RAM of the Commodore 128 is a temporary work area called the *stack*. It starts at location decimal 256 and ends at location 511 (hex $100 to $1FF). This area of computer RAM is referred to as page 1. Paging is explained in the next section.

The stack is used for three purposes in your computer: temporary storage, control of subroutines, and interrupts. The stack is a LIFO (Last In, First Out) structure which means the last value placed on the stack is the first one taken off. When you place a value on the stack, it is referred to as pushing. When you take a value off the stack, it is considered pulling or popping.

Think of the structure as a stack of lunch trays in a cafeteria. The first tray used is the one that is pulled off the top. The last one used is the one on the bottom, and it is used only if all the others are pulled off before it.

The stack pointer is the address of the top stack value (plus 1). When a value is pulled from the stack, the stack pointer then indicates the new address of the next item on the stack. When a subroutine is called or an interrupt occurs, the

address where the interrupt or subroutine occurs is pushed on top of the stack. Once the interrupt or subroutine is serviced, the address where it occurred is popped off the stack and the computer continues where it left off when the interrupt or subroutine occurred.

# 16-BIT ADDRESSING: THE CONCEPT OF PAGING

The Commodore 128 contains 128K of Random Access Memory (RAM). This means you have two banks of 65536 (64K) RAM memory locations (minus two for locations 0 and 1, which are always present in a RAM bank). Since the 8502 is an 8-bit microprocessor, it needs two 8-bit bytes to represent any number between 0 and 65535. One eight-bit byte can only represent numbers between 0 and 255. Your computer needs a way to represent numbers as large as 65535 in order to address all the memory locations.

Here's how your computer represents the largest number in one 8-bit byte. The computer stores it as a binary number. You usually represent it as a hexadecimal number in your machine-language programs. Figure 5-3 shows the relationship between binary, hexadecimal and decimal numbers.

| BINARY | HEXADECIMAL | DECIMAL |
|---|---|---|
| 1 eight-bit Byte = 1 1 1 1 1 1 1 1 | $FF | 255 |

**Figure 5–3. Comparison of Number Systems**

A byte contains eight binary digits (bits). Each bit can have a value of 0 or 1. The largest number your computer can represent in eight binary digits is 1 1 1 1 1 1 1 1, which equals 255 in decimal. This means all eight bits are set, or equal to 1. A bit is considered off if it is equal to 0. In converting binary to decimal, all the binary digits that are set are equal to 2 raised to the power of the bit position. The bit positions are labeled 0 through 7 from right to left. Figure 5–4 provides a visual representation of converting binary to decimal.

| | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ | |
|---|---|---|---|---|---|---|---|---|---|
| One binary byte = | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| The byte in decimal = | 128 + | 64 + | 32 + | 16 + | 8 + | 4 + | 2 + | 1 | = 255 |

**Figure 5–4. Binary/Decimal Conversion**

The top of each column represents the value of 2 raised to the power of the bit position. Since each bit is turned on when you represent the largest number in one byte, add all the values at the bottom of each to obtain the decimal equivalent. Figure 5-5 shows another example that converts the binary number 1 1 0 0 1 0 1 0 to decimal.

| | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ | |
|---|---|---|---|---|---|---|---|---|---|
| **One binary byte** | **1** | **1** | **0** | **0** | **1** | **0** | **1** | **0** | |
| **The byte in decimal** | **128** | **+ 64** | **+ 0** | **+ 0** | **+ 8** | **+ 0** | **+ 2** | **+ 0** | **= 202** |

**Figure 5–5. Binary/Decimal Conversion**

Remember, only add the values of two raised to the bit position if the bit is set. If a bit is off, it equals zero.

Now that you can convert one byte from binary to decimal, you are probably wondering what this has to do with 16-bit addressing. We mentioned before that the program counter—the register responsible for storing the address of the next instruction to be executed—is 16 bits wide. This means it uses two bytes side-by-side to calculate the address.

You just learned about the low byte, the lower half of the 16 bits used to represent an address. The upper half of the 16-bit address is called the high byte. The high byte calculates the upper half of the address the same way as the low byte, except the bit position numbers are labeled from 8 on the right to 15 on the left. When you raise 2 to the power of these bit positions, and add the resulting values to the low byte of the address, you arrive at addresses that go up to 65535. This allows your computer to represent any number between 0 and 65535, and address any memory location within each 64K RAM bank. Figure 5–6 is an illustration of a 16-bit address in decimal:

| **High Byte** | $2^{15}$ | $2^{14}$ | $2^{13}$ | $2^{12}$ | $2^{11}$ | $2^{10}$ | $2^9$ | $2^8$ | |
|---|---|---|---|---|---|---|---|---|---|
| **One binary byte** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | |
| **The byte in decimal** | **= 32768** | **+ 16384** | **+ 8192** | **+ 4096** | **+ 2048** | **+ 1024** | **+ 512** | **+ 256** | **= 65280** |
| **Low Byte** | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ | |
| **One binary byte** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | |
| **The byte in decimal** | **= 128** | **+ 64** | **+ 32** | **+ 16** | **+ 8** | **+ 4** | **+ 2** | **+ 1** | **= 255** |
| | | | | | | | | | **65535** |

**Figure 5–6. Example of 16-Bit Address in Decimal**

You can see that the highest number of the high byte of the 16-bit address is 65280. And you know that the highest number of the low byte of the 16-bit address equals 255. Add the highest high-byte and the highest low-byte number (65280 + 255), to arrive at 65535, the highest address within each of the two 64K RAM banks.

When the microprocessor calculates the address of the next instruction, it looks at the high byte of the 16-bit program counter. Try to think of the high byte of the address as just another 8-bit byte. If this was the case, the bit positions would be labeled from 0 on the right through 7 on the left, just like the low byte of the address. Therefore, the largest number this 8-bit byte can represent again is 255 decimal.

The value in the high byte determines which 256-byte block is accessed. These 256-byte blocks are referred to as pages. The high byte determines the page boundary of the address, so the high byte is calculated in increments of 256 bytes. The high byte of the program counter determines which of the possible 256 pages is being addressed. If you multiply the number of possible pages, 255 by 256 bytes, you realize the highest page starts at location 65280, decimal, the same number as in the high byte in Figure 5–6. Location 65280 is the highest page boundary addressable.

What if you want to address a memory location that does not lie on a page boundary? That's where the low byte of the 16-bit address comes in.

The high byte of the program counter represents the 256-byte page boundary. All addresses between boundaries are represented by the low byte. For example, to address location 65380 decimal represent the high byte as 255, since 255 times 256 equals 65280. You still have to move 100 addresses higher in memory to location 65380.

The low byte contains the value 100 decimal. The low byte value is added to the high byte to find the actual, or effective address.

When you look at the memory map of your Commodore 128, you will see references to the low byte and high byte pointers or vectors to certain machine-language routines within the operating system or to important system memory locations, like the start of BASIC.

You can find out the contents of these addresses and where the routines reside in your Commodore 128's memory by using the PEEK command in BASIC, or the Memory command in the Machine Language Monitor. To find the effective address using BASIC, look in the memory map for the reference to a specific routine or system function, sometimes called a vector. PEEK the high byte, the page number of the routine. Multiply by 256 to find the page boundary. Then PEEK the low byte and add it to the page boundary to arrive at the effective decimal address.

Keep in mind that all the address calculations are performed in binary. They are explained in decimal so they're easier to understand. In your machine language programs, you will usually reference memory in hexadecimal notation, explained in the next section.

# HEXADECIMAL NOTATION

Your 8502 microprocessor only understands the binary digits 0 and 1. Although machine language usually requires hexadecimal notation and BASIC processes decimal numbers, those numbers are translated and processed as binary numbers. Your computer uses three different number systems, binary (base 2), hexadecimal (base 16) and decimal (base 10). The machine-language monitor also uses the octal number base. A number base is also referred to as a radix; therefore, the C128 uses four radices, but the microprocessor only understands binary at machine level.

BASIC understands decimal numbers because they are easiest for people to use. Although BASIC doesn't process as fast as machine language, the ease of use makes up for the loss of speed.

Machine language uses hexadecimal notation because it is closer to the binary number system and easier to translate than decimal. Hexadecimal representation is also used usually by machine-level programmers because it is easier for people to think of a group of eight binary digits (a whole byte) than it is to think of them as separate digits by themselves. How do you find it easier to represent this value:

3A (hexadecimal), or as 00111010 (binary)?

Once values are translated from the higher level language into a form that the microprocessor can understand (binary digits or bits), they are interpreted as electronic switches by the internal circuitry. The switches determine if an electronic impulse will be transmitted by the integrated circuit (I.C.) to perform a specific function, such as addressing a memory location. If the bit equals 1, the switch is interpreted as on, which sends a voltage level (approximately 3 to 5 volts) through the I.C. If the binary digit is equal to 0, no voltage is transmitted. Though this is a simplified illustration, you get an idea of how the microcomputer system can translate, process and perform the instructions you give to your computer. The hardware and software merge here, at machine level.

## UNDERSTANDING HEXADECIMAL (HEX) NOTATION

The key behind understanding hexadecimal (base 16) numbers is to forget about decimal (base 10). Hexadecimal digits are labeled from 0 through 9 and continuing with A through F, where F equals 15 in decimal. By convention, hexadecimal numbers are written with a dollar sign preceding the value so that they can be distinguished from decimal values. Figure 5–7 provides a table of the hexadecimal digits and their decimal and binary equivalents:

| HEXADECIMAL | DECIMAL | BINARY |
|---|---|---|
| $0 | 0 | 0000 |
| $1 | 1 | 0001 |
| $2 | 2 | 0010 |
| $3 | 3 | 0011 |
| $4 | 4 | 0100 |
| $5 | 5 | 0101 |
| $6 | 6 | 0110 |
| $7 | 7 | 0111 |
| $8 | 8 | 1000 |
| $9 | 9 | 1001 |
| $A | 10 | 1010 |
| $B | 11 | 1011 |
| $C | 12 | 1100 |
| $D | 13 | 1101 |
| $E | 14 | 1110 |
| $F | 15 | 1111 |

**Figure 5–7. Hexadecimal/Decimal/Binary Conversion**

Each hex digit represents four bits. The highest number you can represent with four bits is 15 decimal. In machine language, you usually represent operands and addresses as two or four hex digits. Since each hex digit of a four-digit hexadecimal address takes up four bits, four of them represent 16 bits for addressing.

At first you'll find yourself converting decimal addresses and operands into hexadecimal. Then you'll want to convert the other way. See the HEX$ and DEC functions for quick and easy decimal to HEX conversions. In the machine language monitor, use the (+) plus sign to represent decimal numbers. Use the conversions for now, but eventually you should find yourself thinking hexadecimal notation instead of always converting from decimal to hexadecimal.

# ADDRESSING MODES IN THE COMMODORE 128

*Addressing* is the process by which the microprocessor references memory. The 8502 microprocessor has many ways to address the internal locations in memory. The different addressing modes require either one, two or three bytes of storage depending on the instruction. Each instruction has a different version and op-code. For example, LDA (LoaD the Accumulator) has eight versions, each with a different op-code to specify the various addressing modes. See the 8502 Instruction and Addressing Table section for the different versions of all the 8502 machine-language instructions.

## ACCUMULATOR ADDRESSING

*Accumulator addressing* implies that the specified operation code operates on the accumulator. The operand field is omitted since the instruction can only perform the operation on the accumulator. Accumulator instructions require only one byte of storage. Here are some examples of accumulator addressing instructions:

| INSTRUCTION | HEX OPCODE | MEANING |
|---|---|---|
| **ASL** | **$0A** | **Shift one bit left** |
| **LSR** | **$4A** | **Shift one bit right** |
| **ROR** | **$6A** | **Rotate one bit right** |

## IMMEDIATE ADDRESSING

*Immediate addressing* specifies that the operand be a constant value rather than the contents of a particular address. The operand is the data, not a pointer to the data. At machine level, the microprocessor actually interprets an operand field constant and an address in the operand field as two different op-codes, so the pound sign gives the programmer a way to distinguish between the data and a pointer to the data. Immediate addressing instructions require two bytes of storage. Here are some immediate addressing instruction examples:

| INSTRUCTION | HEX OPCODE | MEANING |
|---|---|---|
| **LDA #$0F** | **$A9** | **Load the accumulator with 15 ($0F)** |
| **CMP #$FF** | **$C9** | **Compare the accumulator with 255 ($FF)** |
| **SBC #$E0** | **$E9** | **Subtract 224 ($E0) from accumulator** |

## ABSOLUTE ADDRESSING

*Absolute addressing* allows you to access any of the memory locations within either 64K RAM bank. Absolute addressing requires three bytes of storage; the first byte for the op-code, the second for the low byte of the address and the third for the high byte. Here are some examples of absolute addressing instructions:

| INSTRUCTION | HEX OPCODE | MEANING |
|---|---|---|
| **INC $4FFC** | **$EE** | **Increment the contents of address $4FFC by 1** |
| **LDX $200C** | **$AE** | **Load the X register with the contents of address $200C** |
| **JSR $FFC3** | **$20** | **Jump to location $FFC3 and save the return address** |

## ZERO-PAGE ADDRESSING

*Zero-page addressing* requires two bytes of storage; the first byte is used for the opcode and the second for the zero-page address. Since zero page ranges from addresses 0 through 255, the computer only needs the low byte to represent the actual address. The high byte is assumed to be 0; therefore, it is not specified. When addressing a zero-page location, you can still use absolute addressing; however, the execution time is not as fast as zero-page addressing. Here are some examples:

| INSTRUCTION | HEX OPCODE | MEANING |
|---|---|---|
| **LDA $FF** | **$A5** | **Load the accumulator with the contents of zero-page location $FF (255)** |
| **ORA $E4** | **$05** | **OR the accumulator with the contents of location $E4** |
| **ROR $0F** | **$66** | **Rotate the contents of location $0F one bit to the right** |

## IMPLIED ADDRESSING

In *implied addressing mode*, no operand is specified because the op-code suggests the action to be taken. Since no address or operand is specified, an implied instruction requires only one byte for the op-code. Some examples are:

| INSTRUCTION | HEX OPCODE | MEANING |
|---|---|---|
| **DEX** | **$CA** | **Decrement the contents of the X register** |
| **INY** | **$C8** | **Increment the contents of the Y register** |
| **RTS** | **$60** | **Return from Subroutine** |

## RELATIVE ADDRESSING

*Relative addressing* is used exclusively with branch instructions. The branch instructions (BEQ, BNE, BCC, etc.) allow you to alter the execution path depending on a particular condition. Branch instructions are similar to IF . . . THEN statements in BASIC since they both conditionally perform a specified set of instructions.

The operand in the branch instruction determines the destination of the conditional branch. For example, the op-code BEQ stands for Branch on result EQual to zero. If the zero flag in the status register is equal to 1 add the operand to the program counter and continue execution at this new address. Figure 5–8 provides an example in symbolic assembly language.

| | | | |
|---|---|---|---|
| | **LDA #$01** | **.01800 A9 01** | **LDA #$01** |
| | **STA TEMP** | **.01802 85 FA** | **STA $FA** |
| | **DEC TEMP** | **.01804 C6 FA** | **DEC $FA** |
| **START** | **BEQ START** | **.01806 F0 FC** | **BEQ $1804** |
| | **LDX #$01** | **.01808 A2 01** | **LDX #$01** |
| | **STA COUNT** | **.0180A 85 FB** | **STA $FB** |
| | **(A)** | **(B)** | **(C)** |

*NOTE: The machine language monitor does not provide for symbolic addresses and labels like TEMP and START.

**Figure 5–8. Relative Addressing**

Figure 5–8 lists the (A) code on the left as it appears in symbolic assembly language. The code (B) in the middle is the actual machine-level machine code as it appears in the machine language monitor. The (C) code to the right is the symbolic machine language as it appears in the monitor as executable code.

In this program segment, the first instruction LoaDs the Accumulator with 1. STA is the op-code for STore the contents of the Accumulator in the variable TEMP. The third instruction, DEC, decrements the contents of the variable TEMP. In the third instruction, START is a label which marks the beginning of the conditional loop. The branch instruction (BEQ) checks to see if the value stored in TEMP equals 0 as a result of the DECrement instruction. The instruction marks the end of the loop.

The first time through this loop, the result in TEMP equals 0 so program control branches back to the instruction specified by the label START.

The second time through the loop, TEMP is less than zero; therefore, the zero flag in the status register is cleared, the program does not branch to START and continues with the statement directly following the branch instruction (LDX #$01).

Because of the way this program segment is written, a branch can occur only once, the first time through the loop.

Under relative addressing, the first byte of the instruction is the op-code and the second is the operand, representing an offset of a number of memory locations. The location to branch back to is not interpreted as an absolute address but an offset relative to the location of the branch instruction in memory.

The offset ranges from –128 through 127. If the condition of the branch is met, the offset is added to the program counter and the program branches to the address in memory.

In the example in Figure 5–8, notice that the operand in the branch instruction is only one instruction past the label START. The operand START is interpreted by the computer as an offset of three bytes backward in memory since the DEC instruction use 2 bytes and the BEQ op-code uses one byte. The 8502 can only branch forward 127 bytes and branch backward by 128 bytes.

If you enter the machine-language monitor and disassemble the machine-language code, you'll see how the computer represents a branch instruction operand as in part (B) of Figure 5–8. The symbolic code in part (C) operand field represents the operands as absolute addresses but the assembled hexadecimal code to the left in part (B) of the op code stores the operand using one byte, a number plus or minus the address of the branch instruction. The largest number for a forward branch is $7F. A backward branch is represented by hex numbers greater than $80. When you are within the machine-language monitor, subtract the operand offset from 255 ($FF) to find the actual value of the negative offset. In this case $FF minus 3 equals $FC, which is the operand in the branch instruction in part (B) of Figure 5–8.

Here are some examples of relative addressing branch instructions:

| INSTRUCTION | HEX OP-CODE | MEANING |
|---|---|---|
| **BEQ** | **$F0** | **Branch on result Equal to 0** |
| **BNE** | **$D0** | **Branch on result Not Equal to 0** |
| **BCC** | **$90** | **Branch on Carry Clear** |

## INDEXED ADDRESSING MODES

The Commodore 128 has two special-purpose registers: the X and Y index registers. In indexing addressing modes, index registers modify an address by adding their contents to a base address to arrive at the actual or effective address. For example, here's a program segment that illustrates the importance of address modification, using the X and Y index registers:

```
        LDA #$0F
        LDX #$00
LOOP    STA $2000,X
        INX
        BNE LOOP
```

The first instruction in this program loads the accumulator with $0F(15 decimal). The second instruction loads the X register with 0. The third instruction stores the contents of the accumulator into the address $2000 added to the contents of the X index register. The first time the loop cycles, $0F is stored in address $2000 ($2000 + 0 = $2000). The next instruction (INX) increments the contents of the X register. The last instruction in the loop branches to the statement specified by the label LOOP, which is the STA $2000, X instruction. The second time through the loop, $0F is stored in location $2001 ($2000 + 1). The third cycle of the loop stores $0F in location $2002, etc.

The loop continues to cycle and stores $0F in consecutive locations until the X register equals 0. In other words, the loop circulates 256 times until the X register equals 0, since 255 plus 1 is represented as 0. This is because the extra bit is carried over to the ninth bit position, which doesn't exist in an eight-bit number, so the register is reset to zero. This is similar to when your car odometer is set at 99,999 miles. When you travel another mile the dial resets to 00,000.

This example shows just one way to modify addresses with the index registers. The Commodore 128 has four indexed addressing modes: (1) indexed absolute addressing (illustrated in the example just shown), (2) indexed zero-page addressing, (3) indexed indirect addressing, and (4) indirect indexed addressing.

## INDEXED ZERO-PAGE ADDRESSING

This type of addressing is similar to zero-page addressing except that the index registers (X or Y) are used to modify addresses within page zero ($00 to $FF) of memory. Since zero-page addressing requires no high byte to represent the page number, this type of instruction requires only two bytes of memory. The effective (actual) address is calculated by adding the contents of the index register to the low byte of the address in the program counter. This addressing mode is faster and more efficient than using indexed absolute addressing in zero page.

Here are some examples of indexed zero-page addressing instructions:

| INSTRUCTION | HEX OP-CODE | MEANING |
|---|---|---|
| **INC operand, X** | **$F6** | **Increment the contents of memory by 1. The base address (the operand) is added to the contents of the index register (X)).** |
| **CMP operand,X** | **$D5** | **Compare the contents of the accumulator with memory. The memory base address (the operand) is added to the contents of the index register (X)).** |

## INDEXED ABSOLUTE ADDRESSING

Indexed absolute addressing allows you to access and modify any of the memory locations in each of the two 64K banks. The effective address is calculated by adding the contents of the index register (X or Y) to the high and low byte base address determined by the operand. Since absolute addressing can access any of the available memory locations, high and low bytes are required to form the 16-bit address. Therefore, this type of addressing requires three bytes.

Here are some examples of indexed absolute addressing instructions:

| INSTRUCTION | HEX OP-CODE | MEANING |
|---|---|---|
| **AND operand,Y** | **$39** | **Perform the logical AND operation on the accumulator and the contents of memory base address plus the contents of the register (Y).** |
| **ASL operand,X** | **$1E** | **Shift the contents of the memory (the memory is the base address (the operand) added to the contents of the index register (X)) one bit to the left.** |

# THE INDIRECT ADDRESSING CONCEPT

So far you've learned that the computer calculates the effective address as the base address (in the program counter) plus the offset from the contents of the index registers if indexed addressing is used. Indirect addressing calculates the effective address differently.

Think of indirect addressing as the address of an address. Here's an illustration using absolute indirect addressing:

```
JMP ($0326)
```

The above JuMP instruction is an example of absolute indirect addressing. This type of instruction requires three bytes: one for the op-code, one for the low byte and one byte for the high byte of the 16-bit address. The parentheses indicate that indirect addressing is used. The second and third bytes of the JMP instruction specify the low and high byte of the address. The address in the operand field is only the low byte of the effective address. The contents of the byte immediately following the address specified in the JMP instruction is automatically placed into the program counter as the high byte of the effective address. In this example, the contents of location $0326 and $0327 represent the address of the actual instructions to be executed. For example, location $0326, the low byte of the effective address, contains the value $65 and location $0327, the high byte of the effective address, contains the value $F2. The high- and low-byte values are placed in

the program counter as the address $F265, the actual address of the next instruction the computer executes then is $F265.

If the parentheses were not present, the assembler interprets the instruction as an absolute addressing instruction. The computer would understand the low byte to be $26 and the high byte to be $03 and would JuMP to the instruction located at $0326 instead of the intended address of $F265. Since this is not the case, the high byte is automatically presumed to be the low byte address plus 1 (the contents of $0327).

The last two addressing modes, indirect indexed and indexed indirect, use the same principle as absolute indirect addressing. Here's an explanation of each.

## INDEXED INDIRECT ADDRESSING

Indexed indirect addressing is similar to absolute indirect, although it uses index registers to modify an address. This type of addressing, sometimes called indirect X addressing, requires two bytes of storage: the first byte is for the op-code and the second is for the operand which is used in the effective address calculation. The address specified in the second byte is added to the contents of the X register and the carry, if any, is ignored. The results point to an address in page zero in which its contents contain the low byte of the effective address. The zero page address plus 1 indicates the high byte of the effective address. Both locations in which the low and high bytes of the effective address are contained must be located in page zero, locations $00 through $FF. Here's an example:

```
LDX #$04
LDA #$00
STA ($DF,X)
```

The first line loads the X register with $04. Next, the accumulator is loaded with 0. The third instruction stores zero in the effective address. Calculate the effective address by taking the base address $DF (not the contents of it) and add the contents of the X register ($04) to it, which equals $E3. The contents of location $E3 is the low byte of the effective address and the contents of $E4 is the high byte of the effective address. For example, the contents of address $E3 contain $56 and the contents of address $E4 contain $F3. Since the contents of $E3 is the low byte and the contents of $E4 is the high byte, the effective address is $F356. Indexed indirect addressing is referred to as pre-indexing because the indexing occurs before the effective address is actually obtained. Indirect X addressing is useful in addressing a series of pointers such as the zero-page memory of the Commodore 128.

## INDIRECT INDEXED ADDRESSING

This mode, also called indirect Y addressing, is post-indexed, which means the adding of the index itself obtains the effective address. This mode operates on the principle of a base address and a displacement. Here's how it works.

The first of two bytes is the op-code, the second is the operand, a pointer to a

zero-page memory address. The contents of the pointer and the contents of the Y register are added to arrive at the low byte of the effective address. The contents of the pointer act as the base address and the contents of the Y register act as the displacement. The carry, if any, is added to the memory location directly following the low-byte address which becomes the high byte of the effective address. This is true indexing, designed specifically for manipulating tables of data. In order to access different table values, just change the contents of the Y register since the base address is already established. Here's an example:

```
LDY #$08
LDA #$00
STA ($EA),Y
```

The first instruction loads the Y register with $08. The second instruction loads the accumulator with 0. The third instruction stores the contents of the accumulator in the effective address.

To find the effective address, add the contents of the zero page memory location (base address) specified in the instruction to the contents of the Y register (displacement). In this example, the contents of the address $EA equals $F0. Add $F0 to the contents of the Y register ($08) to arrive at $F8, the low byte of the effective address of the next instruction. The high byte of the effective address is obtained by adding the carry (none in this case) to the zero-page memory location immediately following the low-byte address. For example, location $F9 contains the value $3F. Since the low byte is $F8 and the high byte equals $3F, the effective address is $3FF8.

Notice the difference between indirect indexed and indexed indirect addressing modes as they can be confusing. Remember, the most important difference between the two addressing modes is the way the effective address is calculated. Indexed indirect is X indexing, which is indexed prior to the arrival of the effective address. Indirect indexed is post-indexed with the Y register.

You have just covered all the addressing modes in the Commodore 128. Each calls for different circumstances and you should use the correct mode whenever circumstances dictate it to obtain optimal performance from the microprocessor. For example, use indexed zero-page addressing when you are manipulating zero-page locations instead of using indexed absolute.

# TYPES OF INSTRUCTIONS

This section explains all the types of machine-language instructions available in the Commodore 128. They are first covered by type of instruction, such as REGISTER TO MEMORY and COMPARE instructions; then they are listed alphabetically by op-code mnemonic with all the different addressing options. This section provides important information on programming in machine language on the Commodore 128 (or any 6502-based microcomputer).

Use this information as a reference for background on each instruction. Figure 5–9 provides an alphabetized list of the 8502 microprocessor op-code mnemonics. For detailed, quick-reference information, see the following section for an alphabetic list of instructions, their hexadecimal op-codes, the different versions of the instructions for each addressing mode and the way they affect the flags in the status register.

**8502 MICROPROCESSOR INSTRUCTION SET—ALPHABETIC SEQUENCE**

| | |
|---|---|
| **ADC** | **Add Memory to Accumulator with Carry** |
| **AND** | **"AND" Memory with Accumulator** |
| **ASL** | **Shift Left One Bit (Memory or Accumulator)** |
| **BCC** | **Branch on Carry Clear** |
| **BCS** | **Branch on Carry Set** |
| **BEQ** | **Branch on Result Zero** |
| **BIT** | **Test Bits in Memory with Accumulator** |
| **BMI** | **Branch on Result Minus** |
| **BNE** | **Branch on Result not Zero** |
| **BPL** | **Branch on Result Plus** |
| **BRK** | **Force Break** |
| **BVC** | **Branch on Overflow Clear** |
| **BVS** | **Branch on Overflow Set** |
| **CLC** | **Clear Carry Flag** |
| **CLD** | **Clear Decimal Mode** |
| **CLI** | **Clear Interrupt Disable Bit** |
| **CLV** | **Clear Overflow Flag** |
| **CMP** | **Compare Memory and Accumulator** |
| **CPX** | **Compare Memory and Index X** |
| **CPY** | **Compare Memory and Index Y** |
| **DEC** | **Decrement Memory by One** |
| **DEX** | **Decrement Index X by One** |
| **DEY** | **Decrement Index Y by One** |
| **EOR** | **"Exclusive-Or" Memory with Accumulator** |
| **INC** | **Increment Memory by One** |
| **INX** | **Increment Index X by One** |
| **INY** | **Increment Index Y by One** |
| **JMP** | **Jump to New Location** |
| **JSR** | **Jump to New Location Saving Return Address** |
| **LDA** | **Load Accumulator with Memory** |
| **LDX** | **Load Index X with Memory** |
| **LDY** | **Load Index Y with Memory** |
| **LSR** | **Shift Right One Bit (Memory or Accumulator)** |
| **NOP** | **No Operation** |

| 8502 MICROPROCESSOR INSTRUCTION SET—ALPHABETIC SEQUENCE (cont'd) | |
|---|---|
| ORA | "OR" Memory with Accumulator |
| PHA | Push Accumulator on Stack |
| PHP | Push Processor Status on Stack |
| PLA | Pull Accumulator from Stack |
| PLP | Pull Processor Status from Stack |
| ROL | Rotate One Bit Left (Memory or Accumulator) |
| ROR | Rotate One Bit Right (Memory or Accumulator) |
| RTI | Return from Interrupt |
| RTS | Return from Subroutine |
| SBC | Subtract Memory from Accumulator with Borrow |
| SEC | Set Carry Flag |
| SED | Set Decimal Mode |
| SEI | Set Interrupt Disable Status |
| STA | Store Accumulator in Memory |
| STX | Store Index X in Memory |
| STY | Store Index Y in Memory |
| TAX | Transfer Accumulator to Index X |
| TAY | Transfer Accumulator to Index Y |
| TSX | Transfer Stack Pointer to Index X |
| TXA | Transfer Index X to Accumulator |
| TXS | Transfer Index X to Stack Pointer |
| TYA | Transfer Index Y to Accumulator |

**Figure 5–9. 8502 Op-Code Mnemonics**

# REGISTER TO MEMORY INSTRUCTIONS

The REGISTER TO MEMORY instructions are:

LDA STA
LDX STX
LDY STY

The register to memory instructions either place a value into the accumulator, X register or Y register from memory, or store a value from a register (A, X, or Y) into a memory address.

## LOADING THE ACCUMULATOR

The first and most common instruction is LDA, LoaD the Accumulator. This places a value into the accumulator, the most powerful and active register in the microprocessor. The value is derived from the contents of a memory location or a constant. Here's an example:

```
LDA $2000
```

This instruction loads the contents of the memory location $2000 (8192 decimal) into the accumulator. The value in the memory location $2000 remains the same. The value also remains in the accumulator until another value is placed there or another operation acts upon it.

The previous example is just one of the addressing modes for loading the accumulator. Another form the LDA instruction can take is to load a constant. To load a constant into the accumulator, you must precede the dollar sign ($) with a pound sign (#). For readability, it's a good idea to place at least one space between the op-code and the operand but it is not necessary. Here's an example of loading a constant into the accumulator:

```
LDA #$0A
```

This loads the constant $0A (10 decimal) into the accumulator. Remember, precede a constant with a pound sign, or else the assembler interprets the instruction as the contents of a memory address.

The LDX and LDY instructions work the same way as the LDA instruction. Again, you can load a constant or the contents of a memory address into the X and Y registers. Examples:

```
LDX #$0A
LDX $2000
LDX #$FB
```

### STORE: THE OPPOSITE OF LOAD

You know how to place a value into a register, but how do you do the opposite? The STORE instruction performs the opposite of a load. It places a value from the A (accumulator), X, or Y registers into a specified memory address. As you learned in the addressing section, the load, store and most other machine-language instructions have several versions, depending on the type of addressing used. Here's an example:

```
STA $FC3E
```

This stores the contents of the accumulator into memory location $FC3E. The contents of the accumulator remain the same until another instruction modifies it. The STX and STY instructions work the same way; they store the contents of the register into a specified memory address. There is no immediate version or pound sign version of the store command.

## COUNTER INSTRUCTIONS

The COUNTER instructions are

```
INC     DEC
INX     DEX
INY     DEY
```

*Counter instructions* can be used to keep track of or count the number of times an event occurs. These instructions are used for mathematical manipulations or indexing a

series of addresses. The counter instruction, INC, increments the contents of a memory address by a value of 1 each time it is encountered. These instructions are used primarily within a program loop and in conjunction with a branch instruction. Here's an example of a loop and how INC keeps track of a number of occurrences of an event:

```
        LDX #$00
        TXA
START   STA $2000,X
        INX
        BNE START
```

The first instruction loads a 0 into the X register. The second instruction transfers the contents of the X register into the accumulator (without erasing the X register). Instruction three stores the contents of the accumulator (0) into location $2000 the first time through the loop. The fourth instruction increments the contents of the X register. The last instruction branches to the instruction specified by the label START, until the value of the X register equals 0.

This program segment stores 0's in an entire page (256 locations) starting at $2000 and ending at $20FF. When the contents of the X register equals 255 and it is incremented again, it is reset to 0, since it can only hold an eight-bit number. When this occurs, the branch is skipped and the program continues with the instruction directly following the branch instruction.

The INY instruction operates in the same way as INX, since it also only uses implied addressing. The INC instruction, on the other hand, uses several different addressing modes including absolute, which uses 16-bit addresses. With the INC instruction, you can count past the capacity of an 8-bit number, though you must separate the counter into a high byte and a low byte. For example, the low byte counts the increments of less than a page and the high byte keeps track of the number of pages. When low-byte counter is at 255 and is incremented, it is set back to 0. When this occurs, increment the high-byte counter. To count up to 260 (decimal), the high-byte value equals 1 and the low byte equals 4. Here's an equation to illustrate the point:

$$(1 * 256) + 4 = 260$$

Here's the machine-language code that does this:

```
        LDA #$00
        STA HIGH
        STA LOW
LOOP    INC LOW
        BNE LOOP
        INC HIGH
LOOP 2  INC LOW
        LDA LOW
        CMP #$04
        BNE LOOP2
```

The DECrement instructions operate the same way as the increment instructions. They are the negative number counterparts of the increment counters.

## COMPARE INSTRUCTIONS

The Commodore 128 has three compare instructions that check the contents of a register with the contents of memory. A compare operation can be used to determine which instructions to execute as a result of a conditioned value. The compare instructions are:

CMP
CPX
CPY

The CMP instruction compares the contents of the accumulator with the contents of the specified address in the instruction. Compare instructions essentially *subtract* memory from a register value but change neither—they just set status flags. CPX compares the contents of the X register with the specified address. CPY compares the contents of the Y register with the specified memory location.

All three instructions have versions that will operate in immediate, zero-page and absolute addressing modes. This means you can compare the contents of a register (A,X, or Y) with the contents of a zero-page location, any other address above zero page, or against a constant. Here's an example:

```
        LDX #$00
        LDA #$00
ONE     STA $DF,X
        INX
        CPX #$0A
        BNE ONE
```

The preceding program segment stores 0's in 10 consecutive memory addresses starting at $DF. The first instruction loads the X register with 0, the second loads 0 into the accumulator. The third instruction stores 0 in location $DF plus the contents of the X register. The fourth instruction increments the X register. The fifth instruction compares the contents of the X index register with the constant $0A (10 decimal). If the contents of the X register does not equal $0A, the program segment branches back to the store instruction specified by the label ONE. After the loop cycles ten times, the X register and the constant $0A are equal. Therefore the processor does not take the branch and the program continues with the instruction immediately following BNE.

You can compare the value of a register with the contents of an absolute memory address. Here's the same example as above using the contents of a memory address instead of a constant:

```
        LDA #$0A
        STA $FB
        LDX #$00
        LDA #$00
ONE     STA $DF,X
        INX
        CPX $FB
        BNE ONE
```

Remember, if you want to compare numbers larger than eight bits can represent (greater than 255 decimal), you must separate the number into a low byte and a high byte.

The BIT instruction can also be used for comparisons. See the logical instructions next.

## ARITHMETIC AND LOGICAL INSTRUCTIONS

The accumulator is responsible for all mathematical and logical operations performed in your computer. The mathematical and logical instructions available in machine language are:

| | |
|---|---|
| ADC | EOR |
| AND | ORA |
| BIT | SBC |

Here's what each instruction means:

**ADC**—Add the contents of the specified memory address to the contents of the accumulator with a carry. It is considered a good programming practice to clear the carry bit with the CLC instruction before performing any addition. This avoids adding the carry into the result.

**AND**—Perform the logical AND operation with the contents of the accumulator and the contents of the specified memory address.

**BIT**—Compare the bits in the specified memory address with those in the accumulator. Bits 6 and 7 are transferred to the status register flags. Bit 7 is transferred to the negative status flag bit and bit 6 is sent to the overflow status flag bit.

**EOR**—Perform the exclusive OR operation with the contents of the specified memory address and the contents of the accumulator.

**ORA**—Perform the logical OR operation with the contents of the specified memory address and the contents of the accumulator.

**SBC**—Subtract the contents of the specified memory address from the contents of the accumulator with a borrow. (It is a good practice to set the carry flag before performing subtraction. This avoids subtracting the borrowed bit from the result.)

### ARITHMETIC INSTRUCTIONS (ADC, SBC)

The addition and subtraction instructions are easy to understand. Here's an example:

```
CLC
LDA #$0A
STA $FB
ADC #$04
SEC
SBC #$06
ADC $FB
STA $FD
```

This program segment essentially performs the following mathematical operation: (10 + 4)–6 + 10 = 18.

The first instruction clears the carry bit. The second instruction loads the accumulator with $0A (10 decimal). The third instruction stores the value in address $FB for later use. The fourth instruction adds the constant $04 to the value already in the accumulator. The SBC instruction subtracts the constant $06 from the contents of the accumulator. The next instruction, ADC $FB, adds the contents of memory location $FB to the contents of the accumulator. The resulting value (18($12)) of all the mathematical operations is stored in address $FD.

## LOGICAL INSTRUCTIONS (AND, EOR, AND ORA)

These instructions operate on the contents of a memory address and a register. The AND operation is a binary (Boolean) algebra operation having two operands that can result in one of two values, 0 or 1. The only way an AND operation can result in a 1 is if both the operands equal 1; otherwise the result is 0. For example, the two operands are the contents of a specified memory address and the contents of the accumulator. Here's an illustration of this concept:

**Memory address = 10001010**
**Accumulator = 11110010**
**Result of AND = 10000010**

As noted, the result of an AND operation is (true) 1, only if the two operands are equal to 1; otherwise the result is 0. Notice bit 7 (high-order bit) equals 1 because both bit 7's in the operands are 1. The only other resulting bit equal to 1 is bit 1, since both bit 1's are equal to 1. The rest of the bits are equal to zero since no other bit positions in both operands are equal to 1. A 1 and a 0 equals 0, as does a 0 and a 0.

The Boolean OR works differently. The general rule is:

If one of the operands equals 1, the resulting Boolean value equals 1.

For example, the two operands are the contents of a specified memory address and the contents of the accumulator. Each individual bit can be treated as an operand. Here's an illustration.

**Contents of Memory Address = 10101001**
**Contents of Accumulator = 10000011**
**Result of the OR operation = 10101011**

For all the bit positions that equal one in either operand, the resulting value of that bit position equals 1. The result is 1 if either operand or both operands are equal to 1.

The exclusive OR works similarly to the OR operation, except if both operands equal 1, the result is zero. This suggests the following general rule:

If either of the operands equals 1, the resulting Boolean value is 1, except if both operands are 1, then the result equals 0.

Here's an example using this rule:

| | | |
|---|---|---|
| **Contents of Memory Address** | = | **10101001** |
| **Contents of Accumulator** | = | **10000011** |
| **Result of the exclusive OR** | = | **00101010** |

In this example, the operands are the same as in the previous OR example. Notice bits 0 and 7 are now equal to 0 since both operands are equal to 1. All other bit values remain the same.

## BIT

The BIT instruction performs a logical AND operation on the contents of the specified memory address and the contents of the accumulator, but the resulting value is not stored in the accumulator. Instead, the zero flag in the status register is set by the result of the operation. The BIT instruction compares the contents of the accumulator and the contents of the memory address, bit-for-bit. If the result of the operation of the accumulator being ANDed by a memory location is 0, then the zero flag (in the status register) is set to a 1. Otherwise the zero flag is 0.

Your machine language program can then act conditionally depending on the result of the zero flag in the status register. In addition, bits 7 and 6 from the specified memory address are moved into the negative-flag and overflow-flag bit positions in the status register, respectively. These flags can also be used to perform conditional instructions depending on the value of the flag. For example, the BIT instruction performs the following:

| | | | | |
|---|---|---|---|---|
| | | | | **7 0** |
| | | | | **N V B D I Z C** |
| **Contents of Memory Address** | = | **10101001** | | |
| **Contents of Accumulator** | = | **11001101** | → | **1 0 0** |
| **Result of BIT instruction (Not stored in accumulator)** | = | **10001001** | | **Status Register** |

Since the resulting bit pattern is not 0, the zero flag in the status register is 0. In addition, bits 7 and 6 are placed in the bit positions of the negative and overflow flags, respectively, in the status register. Notice the result of the BIT instruction's AND operation is not stored in the accumulator. The original contents of the accumulator remain intact. See the following example of 2-bit pattern operands that result in 0 when ANDed:

| | | | 7 | | | | | 0 |
|---|---|---|---|---|---|---|---|---|
| | | | N | V | | B | D | I | Z | C |
| Contents of Memory Address | = 01111010 | | | | | | | | | |
| Contents of Accumulator | = 10000100 | → | 0 | 1 | | | | 1 | | |
| Result of BIT instruction | = 00001000 | | Status Register | | | | | | | |

This time the bit patterns result in 0. Therefore, the zero flag in the status register is set to 1. Bits 7 and 6 are also placed into their respective negative and overflow status register bit positions from their positions in the memory location.

Now you know how each of the arithmetic and logical instructions operate. The next section discusses branching instructions. Branching instructions are designed so you can conditionally execute a certain set of instructions, depending on the result of a condition. Many times the conditions are contingent on the results of an arithmetic or logical operation, which affects the flags in the status register. The branching instructions then act according to the flags in the status register.

## BRANCHING INSTRUCTIONS

The 8502 microprocessor has many conditional *branching instructions*. By definition, a branch temporarily redirects the otherwise sequential execution of program instructions. It transfers control to a location of a machine-language instruction other than the one immediately following the branch instruction in memory.

The conditional branch instructions cause the microprocessor to examine a particular flag in the status register. The processor, depending on the value of the tested flag, either takes the branch and transfers control of the program to another location or skips the branch and resumes with the instruction immediately following the branch.

Think of a conditional branch as a test. For example, if the condition passes the test, the program branches or shifts control to an instruction that is not the next sequential instruction in the computer's memory. If it fails the test, the branch is skipped and program control resumes with the instruction immediately following the branch instruction in memory. Remember that program control can also be shifted to an instruction that is out of sequential order if it fails a test. This means you can transfer control of the execution of your program depending on the conditions you create. You may set a condition that branches if the value of a certain flag (operand) is zero. In another instance, you may set a condition to branch if a specific flag is set to 1.

The conditional branch instructions available in the 8502 microprocessor are:

| | |
|---|---|
| BCC | BNE |
| BCS | BPL |
| BEQ | BVC |
| BMI | BVS |

Here's what the conditional branch instructions mean. The phrases in parentheses are the literal translations of the op-code mnemonics. The remainder explains the meaning behind the op-codes.

**BCC**—(Branch on Carry Clear) Branch if the Carry flag in the status register equals 0.
**BCS**—(Branch on Carry Set) Branch if the Carry flag in the status register equals 1.
**BEQ**—(Branch on result EQual zero) Branch if the zero flag in the status register equals 1.
**BMI**—(Branch on result MInus) Branch if the negative flag in the status register equals 1.
**BNE**—(Branch on result Not Equal to zero) Branch if the zero flag in the status register equals 0.
**BPL**—(Branch on result PLus) Branch if the negative flag in the status register equals 0.
**BVC**—(Branch on oVerflow Clear) Branch if the overflow flag in the status register equals 0.
**BVS**—(Branch on oVerflow Set) Branch if the overflow flag in the status register equals 1.

As you can see, all branching instructions depend on the value of a flag in the status register.

Here are some branching examples.

```
READY.

MONITOR
    PC  SR AC XR YR SP
; FB000 00 00 00 00 F8



. 01828  E6 FA    INC $FA
. 0182A  A5 FA    LDA $FA
. 0182C  D0 02    BNE $1830
. 0182E  E6 FB    INC $FB
. 01830  C8       INY
```

This program segment keeps track of the low and high pointers in $FA and $FB respectively. The first instruction (INC $FA) increments the low byte address pointer. Next, the contents of $FA is loaded into the accumulator. The branch instruction (BNE $1830) evaluates the value of the accumulator. If the value is not equal to zero, the branch is taken to the instruction located at address $1830 (INY). In this case the high byte pointer is not yet ready to be incremented, so the INC $FB instruction is skipped. If the value in the accumulator is equal to zero, the branch is skipped and the high byte address pointer is incremented.

This is an example of the BPL (Branch on Result Plus) instruction.

```
READY.

MONITOR
    PC  SR AC XR YR SP
; FB000 00 00 00 00 F8

. 01858  8E 00 D6 STX $D600
. 0185B  2C 00 D6 BIT $D600
. 0185E  10 FB    BPL $185B
. 01860  8D 01 D6 STA $D601
```

This example is a routine that checks the update ready status bit for the 8563 address register, and ensures that data is valid before writing a value to an 8563 register. The first instruction stores the contents of the X register, which was previously loaded

with an 8563 register number, into the 8563 address register. The BIT instruction places bit 7 of location $D600 into the negative flag in the 8502 status register. The BPL instruction branches to the BIT instruction in location $185B as long as the value of the negative flag is equal to 0. To the 8563 chip, this means the data is not yet valid and cannot be written to or read from until bit 7 is set. This loop continues until the value of bit 7 is 1, then it is transferred to the negative flag. The result now becomes negative so the branch is skipped and control is passed to the next instruction in memory, which stores the data into the 8563 data register. Refer to Chapter 10, Writing to an 8563 Register for an expanded version of this program.

## REGISTER TRANSFER INSTRUCTIONS

*Register transfer instructions* move a value from one register (A, X, or Y) to another. This instruction is useful since it only requires one byte of memory and saves the programmer the trouble of loading the value from one register and storing it in another. The 8502 microprocessor has the following six register transfer instructions:

**TAX**—Transfer contents of accumulator to X index register
**TAY**—Transfer contents of accumulator to Y index register
**TSX**—Transfer the contents of the stack pointer to X index register
**TXA**—Transfer the contents of X index register to the accumulator
**TYA**—Transfer the contents of the Y index register to the accumulator
**TXS**—Transfer the contents of the X register to the stack pointer

The TXS and TSX instructions transfer values from the X index register to the stack pointer and vice versa. This is useful if you need to take a value off the stack temporarily, in a mathematical operation (for example, to operate on it and then replace it on the stack). Another use is to take a value off the stack, place it in the X register for temporary storage, add a new value on the stack, and then place the old value back on top. This could be the case when you need to sort values in ascending order.

## SHIFT AND ROTATE INSTRUCTIONS

The *shift* and *rotate* instructions manipulate the bits of the accumulator or memory. Following are the shift and rotate instructions used by the 8502 family of microprocessors:

**ASL**—Shift the whole byte one bit to the left
**LSR**—Shift the whole byte one bit to the right
**ROL**—Rotate the whole byte one bit to the left
**ROR**—Rotate the whole byte one bit to the right

### SHIFT INSTRUCTIONS

The shift instructions are useful when evaluating the value of a single bit at a time in a series of bits that control your program. For example, a joystick read routine is an example that calls for the shift instruction. Locations $DC00 and $DC01 control the joystick direction (bits 0–3), and the joystick fire button (bit 4). One way to evaluate these values is to shift them to the right. This causes the value to be passed to the carry

flag. If the carry flag is enabled (1), then the joystick is being pushed in the direction corresponding to that bit. Here is a joystick read routine that uses the LSR instruction to evaluate the direction of the joystick:

```
READY.

MONITOR
    PC  SR AC XR YR SP
; FB000 00 00 00 00 F8



. 01800  AD 00 DC LDA $DC00
. 01803  A0 00    LDY #$00
. 01805  A2 00    LDX #$00
. 01807  4A       LSR
. 01808  B0 01    BCS $180B
. 0180A  88       DEY
. 0180B  4A       LSR
. 0180C  B0 01    BCS $180F
. 0180E  C8       INY
. 0180F  4A       LSR
. 01810  B0 01    BCS $1813
. 01812  CA       DEX
. 01813  4A       LSR
. 01814  B0 01    BCS $1817
. 01816  E8       INX
. 01817  4A       LSR
. 01818  86 FA    STX $FA
. 0181A  84 FB    STY $FB
. 0181C  60       RTS
```

## ROTATE INSTRUCTIONS

The rotate instructions operate a little differently. Instead of the shifted bit falling into the carry flag, the bit "falling off the edge" is placed in the carry bit, then the carry bit is placed at the opposite end of the byte. For example, if the ROR (rotate right) instruction is specified, each bit is moved one position to the right. Now bit 7 is placed in the carry bit and the carry bit is rotated around to the left and placed in the bit 7 bit position. The ROL instruction operates in the same manner, except the rotation is leftward rather than to the right. See Figure 5–10 to visualize the rotation concept of the ROR (rotate right) instruction:

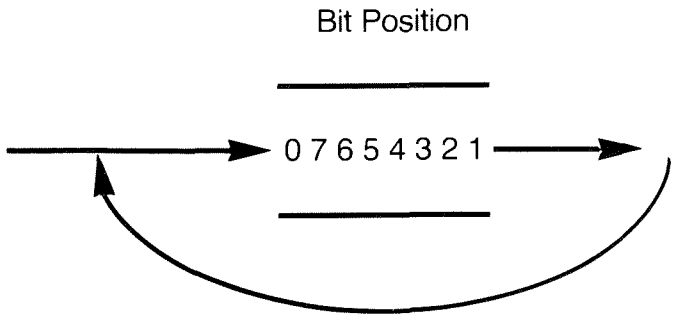



**Figure 5–10. Concept of ROR (Rotate Right) Instruction**

# SET AND CLEAR INSTRUCTIONS

The set and clear instructions are designed to manipulate the bits (flags) within the status register and control certain conditions within the microprocessor. These are the set and clear instructions available in 8502 machine language:

**SEC** Set the Carry Flag
**SED** Set Decimal Mode
**SEI** Set the Interrupt Disable Bit

**CLC** Clear the Carry Flag
**CLD** Clear Decimal Mode
**CLI** Clear the Interrupt Disable Bit
**CLV** Clear the Overflow Flag

Each of these instructions applies to a flag in the status register that controls a particular microprocessor condition. Notice that each clear instruction has a counterpart which sets the condition, except for CLV (Clear Overflow Flag). The overflow flag can be set by the BIT instruction or from the result of a signed mathematical operation overflowing into the sign bit.

Figure 5–11 shows the 8502 status register:

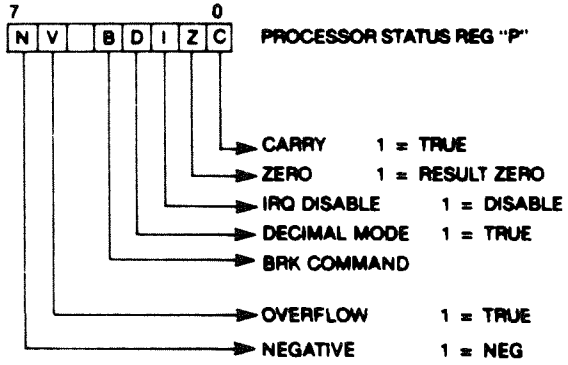



**Figure 5–11. 8502 Status Register**

The flags of the status register are set for various reasons. For example, set decimal mode when you want to perform calculations in binary coded decimal (BCD) notation rather than hexadecimal. Set the carry flag when you are performing subtraction. Set the interrupt disable bit when you want to prevent interrupts from occurring. An example of a split screen, smooth scrolling raster interrupt routine is given at the end of Chapter 8.

The clear instructions operate in the reverse of the set instructions. To make sure that a carry does not occur during an addition operation, clear the carry flag before

adding two numbers in the accumulator. To perform mathematical operations in hexadecimal or binary numbers, clear the decimal mode flag so that your calculations are not mistakenly performed in binary coded decimal. Whenever the result of a signed mathematical operation overflows into the sign bit an overflow error occurs. To correct this, clear the overflow flag with the CLV op-code.

When a program requires interrupts, first set the interrupt disable bit (SEI) to prevent interrupts from occurring. At the end of the interrupt initialization routine, issue the CLI (Clear Interrupt Disable bit) instruction to enable (allow) interrupts to occur.

# JUMP AND RETURN INSTRUCTIONS

## JUMP INSTRUCTIONS

The 8502 processor makes use of two jump instructions:

**JMP**—Jump to new location
**JSR**—Jump to new location Saving the Return address

These instructions both redirect control of the microprocessor to a location other than the one immediately following it in memory. The first instruction, JMP, is a one-way trip to the location specified in the operand field, or the contents of it (indirect). For example:

```
JMP $1800
```

jumps to location $1800 and executes the instruction contained in that location. This is a direct jump.

You can also jump indirectly. For example:

```
JMP ($1800)
```

jumps to the address specified in the contents of location $1800. For instance, location $1800 contains the value $FE and location $1801 contains the value $C0. Therefore, the above instruction jumps to location $C0FE, and not location $1800. Jumping indirectly is always denoted by parentheses around the address in the operand field, and it means to jump to the location specified by the CONTENTS OF the address in the operand field.

The JSR instruction calls subroutines and saves the return address to the stack, so when an RTS instruction is encountered at the end of the subroutine, the microprocessor knows where to resume processing in the main (calling) program. Program control resumes with the instruction in memory immediately following the JSR instruction. In short, JSR is a round trip, while JMP is one way. For example:

```
. 01804  20 58 18 JSR $1858
. 01807  A2 0C    LDX #$0C
```

jumps to the subroutine starting at location $1858. The return address is saved on the stack, so when the RTS instruction is encountered in this subroutine:

```
. 01858  8E 00 D6 STX $D600
. 0185B  2C 00 D6 BIT $D600
. 0185E  10 FB    BPL $185B
. 01860  8D 01 D6 STA $D601
. 01863  60       RTS
```

the processor resumes with the main program instruction (LDX #$0C) in location $1807.

## RETURN INSTRUCTIONS

The 8502 instruction set has two return instructions:

**RTI**—Return from Interrupt
**RTS**—Return from Subroutine

The first instruction returns from your interrupt service routine after the interrupt disable bit is cleared (CLI) and the interrupt occurs. The RTI is the last instruction in the interrupt service routine. The interrupt service routine is the series of instructions which are performed on the occurrence of an interrupt. Refer to Chapter 8, Raster Interrupt Split Screen Program with Horizontal Scrolling for a working example of an interrupt service routine.

The RTS instruction is the last instruction in a machine language subroutine called from BASIC or by the machine language JSR instruction. See the Jump instructions above for an example.

# STACK INSTRUCTIONS

Four stack instructions are included in the 8502 instruction set to manipulate the values on the stack. These instructions are as follows:

**PHA**—Push accumulator on the stack
**PHP**—Push processor status on the stack
**PLA**—Pull accumulator from the stack
**PLP**—Pull processor status from the stack

The term push means to place a value on the stack, while pull means to remove a value from the stack. The only values pushed or pulled on to or off the stack are the contents of the status register or the accumulator. The manipulation of the stack values is important to the programmer when processing interrupts. The Raster Interrupt Split Screen Program with Horizontal Scrolling section in Chapter 8 illustrates the manipulation of the stack values prior to returning from the interrupt.

# THE NOP INSTRUCTION

The NOP instruction stands for no operation. It is often used to add space between program segments for readability. This instruction is not executable.

# 8502 INSTRUCTION AND ADDRESSING TABLE

The next 16 pages contain the 8502 Instruction and Addressing Table. These are the conventions used in the table:

1. OP-CODE
2. Brief definition
3. Operation notation
4. Status flags
5. Flags affected
6. Addressing Modes
7. Assembly language form
8. OP-CODE (in hex)
9. Number of bytes
10. Number of instruction cycles

The following notation applies to this summary:

| | |
|---|---|
| **A** | Accumulator |
| **X,Y** | Index Registers |
| **M** | Memory |
| **P** | Processor Status Register |
| **S** | Stack Pointer |
| √ | Change |
| – | No Change |
| + | Add |
| **Λ** | Logical AND |
| – | Subtract |
| ∀ | Logical Exclusive Or |
| ↑ | Transfer from Stack |
| ↓ | Transfer to Stack |
| → | Transfer to |
| ← | Transfer from |
| **V** | Logical OR |
| **PC** | Program Counter |
| **PCH** | Program Counter High |
| **PCL** | Program Counter Low |
| **OPER** | OPERAND |
| # | IMMEDIATE ADDRESSING MODE |

## ADC — *Add memory to accumulator with carry* — ADC

**Operation:** A + M + C → A, C

| N | E | C | I | D | V |
|---|---|---|---|---|---|
| √ | √ | √ | – | – | √ |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Immediate | ADC # Oper | 69 | 2 | 2 |
| Zero Page | ADC Oper | 65 | 2 | 3 |
| Zero Page, X | ADC Oper, X | 75 | 2 | 4 |
| Absolute | ADC Oper | 6D | 3 | 4 |
| Absolute, X | ADC Oper, X | 7D | 3 | 4* |
| Absolute, Y | ADC Oper, Y | 79 | 3 | 4* |
| (Indirect, X) | ADC (Oper, X) | 61 | 2 | 6 |
| (Indirect), Y | ADC (Oper), Y | 71 | 2 | 5* |

* Add 1 if page boundary is crossed.

## AND — *"AND" memory with accumulator* — AND

Logical AND to the accumulator

**Operation:** A ∧ M → A

| N | E | C | I | D | V |
|---|---|---|---|---|---|
| √ | √ | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Immediate | AND # Oper | 29 | 2 | 2 |
| Zero Page | AND Oper | 25 | 2 | 3 |
| Zero Page, X | AND Oper, X | 35 | 2 | 4 |
| Absolute | AND Oper | 2D | 3 | 4 |
| Absolute, X | AND Oper, X | 3D | 3 | 4* |
| Absolute, Y | AND Oper, Y | 39 | 3 | 4* |
| (Indirect, X) | AND (Oper, X) | 21 | 2 | 6 |
| (Indirect), Y | AND (Oper), Y | 31 | 2 | 5 |

* Add 1 if page boundary is crossed.

## ASL — *ASL Shift Left One Bit (Memory or Accumulator)* — ASL

Operation: C ← [7][6][5][4][3][2][1][0] ← 0

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| √ | √ | √ | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Accumulator | ASL A | 0A | 1 | 2 |
| Zero Page | ASL Oper | 06 | 2 | 5 |
| Zero Page, X | ASL Oper, X | 16 | 2 | 6 |
| Absolute | ASL Oper | 0E | 3 | 6 |
| Absolute, X | ASL Oper, X | 1E | 3 | 7 |

## BCC — *BCC Branch on Carry Clear* — BCC

Operation: Branch on C = 0

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| – | – | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Relative | BCC Oper | 90 | 2 | 2* |

* Add 1 if branch occurs to same page.
* Add 2 if branch occurs to different page.

## BCS — *BCS Branch on carry set* — BCS

Operation: Branch on C = 1

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| – | – | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Relative | BCS Oper | B0 | 2 | 2* |

* Add 1 if branch occurs to same page.
* Add 2 if branch occurs to next page.

## BEQ — *BEQ Branch on result zero* — BEQ

**Operation: Branch on Z = 1**

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| – | – | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Relative | BEQ Oper | F0 | 2 | 2* |

* Add 1 if branch occurs to same page.
* Add 2 if branch occurs to next page.

## BIT — *BIT Test bits in memory with accumulator* — BIT

**Operation: A Λ M, $M_7 \rightarrow N$, $M_6 \rightarrow V$**
**Bit 6 and 7 are transferred to the status register.**
**If the result of AΛM is zero then Z = 1, otherwise Z = 0**

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| $M_7$ | √ | – | – | – | $M_6$ |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Zero Page | BIT Oper | 24 | 2 | 3 |
| Absolute | BIT Oper | 2C | 3 | 4 |

## BMI — *BMI Branch on result minus* — BMI

**Operation: Branch on N = 1**

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| – | – | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Relative | BMI Oper | 30 | 2 | 2* |

* Add 1 if branch occurs to same page.
* Add 2 if branch occurs to different page.

## BNE — *BNE Branch on result not zero* — BNE

**Operation: Branch on Z = 0**

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| – | – | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Relative | BNE Oper | D0 | 2 | 2* |

* Add 1 if branch occurs to same page.
* Add 2 if branch occurs to different page.

**BPL** *BPL Branch on result plus* **BPL**

**Operation: Branch on N = 0**

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| – | – | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Relative | BPL Oper | 10 | 2 | 2* |

* Add 1 if branch occurs to same page.
* Add 2 if branch occurs to different page.

**BRK** *BRK Force Break* **BRK**

**Operation: Forced Interrupt PC + 2 ↓ P ↓**

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| – | – | – | 1 | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Implied | BRK | 00 | 1 | 7 |

1. A BRK command cannot be masked by setting I.

**BVC** *BVC Branch on overflow clear* **BVC**

**Operation: Branch on V = 0**

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| – | – | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Relative | BVC Oper | 50 | 2 | 2* |

* Add 1 if branch occurs to same page.
* Add 2 if branch occurs to different page.

**BVS** *BVS Branch on overflow set* **BVS**

**Operation: Branch on V = 1**

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| – | – | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Relative | BVS Oper | 70 | 2 | 2* |

* Add 1 if branch occurs to same page.
* Add 2 if branch occurs to different page.

## CLC — *CLC Clear carry flag* — CLC

Operation: 0 → C

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| – | – | 0 | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Implied | CLC | 18 | 1 | 2 |

## CLD — *CLD Clear decimal mode* — CLD

Operation: 0 → D

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| – | – | – | – | 0 | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Implied | CLD | D8 | 1 | 2 |

## CLI — *CLI Clear interrupt disable bit* — CLI

Operation: 0 → I

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| – | – | – | 0 | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Implied | CLI | 58 | 1 | 2 |

## CLV — *CLV Clear overflow flag* — CLV

Operation: 0 → V

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| – | – | – | – | – | 0 |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Implied | CLV | B8 | 1 | 2 |

## CMP — *CMP Compare memory and accumulator* — CMP

**Operation: A – M**

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| √ | √ | √ | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Immediate | CMP #Oper | C9 | 2 | 2 |
| Zero Page | CMP Oper | C5 | 2 | 3 |
| Zero Page, X | CMP Oper, X | D5 | 2 | 4 |
| Absolute | CMP Oper | CD | 3 | 4 |
| Absolute, X | CMP Oper, X | DD | 3 | 4* |
| Absolute, Y | CMP Oper, Y | D9 | 3 | 4* |
| (Indirect, X) | CMP (Oper, X) | C1 | 2 | 6 |
| (Indirect), Y | CMP (Oper), Y | D1 | 2 | 5* |

* Add 1 if page boundary is crossed.

## CPX — *CPX Compare Memory and Index X* — CPX

**Operation: X – M**

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| √ | √ | √ | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Immediate | CPX #Oper | E0 | 2 | 2 |
| Zero Page | CPX Oper | E4 | 2 | 3 |
| Absolute | CPX Oper | EC | 3 | 4 |

## CPY — *CPY Compare memory and index Y* — CPY

**Operation: Y – M**

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| √ | √ | √ | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Immediate | CPY #Oper | C0 | 2 | 2 |
| Zero Page | CPY Oper | C4 | 2 | 3 |
| Absolute | CPY Oper | CC | 3 | 4 |

## DEC — *DEC Decrement memory by one* — DEC

Operation: M – 1 → M

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| √ | √ | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Zero Page | DEC Oper | C6 | 2 | 5 |
| Zero Page, X | DEC Oper, X | D6 | 2 | 6 |
| Absolute | DEC Oper | CE | 3 | 6 |
| Absolute, X | DEC Oper, X | DE | 3 | 7 |

## DEX — *DEX Decrement index X by one* — DEX

Operation: X – 1 → X

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| √ | √ | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Implied | DEX | CA | 1 | 2 |

## DEY — *DEY Decrement index Y by one* — DEY

Operation: Y – 1 → Y

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| √ | √ | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Implied | DEY | 88 | 1 | 2 |

## EOR — *EOR "Exclusive—Or" memory with accumulator* — EOR

Operation : A → M $\forall$ A

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| √ | √ | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Immediate | EOR #Oper | 49 | 2 | 2 |
| Zero Page | EOR Oper | 45 | 2 | 3 |
| Zero Page, X | EOR Oper, X | 55 | 2 | 4 |
| Absolute | EOR Oper | 4D | 3 | 4 |
| Absolute, X | EOR Oper, X | 5D | 3 | 4* |
| Absolute, Y | EOR Oper, Y | 59 | 3 | 4* |
| (Indirect, X) | EOR (Oper, X) | 41 | 2 | 6 |
| (Indirect), Y | EOR (Oper), Y | 51 | 2 | 5* |

* Add 1 if page boundary is crossed.

## INC — *INC Increment memory by one* — INC

Operation: M + 1 → M

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| √ | √ | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Zero Page | INC Oper | E6 | 2 | 5 |
| Zero Page, X | INC Oper, X | F6 | 2 | 6 |
| Absolute | INC Oper | EE | 3 | 6 |
| Absolute, X | INC Oper, X | FE | 3 | 7 |

## INX — *INX Increment Index X by one* — INX

Operation: X + 1 → X

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| √ | √ | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Implied | INX | E8 | 1 | 2 |

## INY — *INY Increment Index Y by one* — INY

Operation: Y + 1 → Y

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| √ | √ | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Implied | INY | C8 | 1 | 2 |

## JMP — *JMP Jump to new location* — JMP

Operation: (PC + 1) → PCL
(PC + 2) → PCH

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| – | – | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Absolute | JMP Oper | 4C | | 3 |
| Indirect | JMP (Oper) | 6C | 3 | 5 |

## JSR — *JSR Jump to new location saving return address* — JSR

Operation: PC + 2 ↓, (PC + 1) → PCL
(PC + 2) → PCH

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| – | – | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Absolute | JSR Oper | 20 | 3 | 6 |

## LDA — *LDA Load accumulator with memory* — LDA

Operation: M → A

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| √ | √ | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Immediate | LDA #Oper | A9 | 2 | 2 |
| Zero Page | LDA Oper | A5 | 2 | 3 |
| Zero Page, X | LDA Oper, X | B5 | 2 | 4 |
| Absolute | LDA Oper | AD | 3 | 4 |
| Absolute, X | LDA Oper, X | BD | 3 | 4* |
| Absolute, Y | LDA Oper, Y | B9 | 3 | 4* |
| (Indirect, X) | LDA (Oper, X) | A1 | 2 | 6 |
| (Indirect), Y | LDA (Oper), Y | B1 | 2 | 5* |

* Add 1 if page boundary is crossed.

## LDX — *LDX Load index X with memory* — LDX

Operation: M → X

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| √ | √ | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Immediate | LDX #Oper | A2 | 2 | 2 |
| Zero Page | LDX Oper | A6 | 2 | 3 |
| Zero Page, Y | LDX Oper, Y | B6 | 2 | 4 |
| Absolute | LDX Oper | AE | 3 | 4 |
| Absolute, Y | LDX Oper, Y | BE | 3 | 4* |

* Add 1 when page boundary is crossed.

## LDY — *LDY Load index Y with memory* — LDY

Operation: M → Y

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| √ | √ | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Immediate | LDY #Oper | A0 | 2 | 2 |
| Zero Page | LDY Oper | A4 | 2 | 3 |
| Zero Page, X | LDY Oper, X | B4 | 2 | 4 |
| Absolute | LDY Oper | AC | 3 | 4 |
| Absolute, X | LDY Oper, X | BC | 3 | 4* |

* Add 1 when page boundary is crossed.

## LSR — *LSR Shift right one bit (memory or accumulator)* — LSR

Operation: 0 → [7][6][5][4][3][2][1][0] → C

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| 0 | √ | √ | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Accumulator | LSR A | 4A | 1 | 2 |
| Zero Page | LSR Oper | 46 | 2 | 5 |
| Zero Page, X | LSR Oper, X | 56 | 2 | 6 |
| Absolute | LSR Oper | 4E | 3 | 6 |
| Absolute, X | LSR Oper, X | 5E | 3 | 7 |

## NOP — *NOP No operation* — NOP

Operation: No Operation (2 cycles)

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| – | – | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Implied | NOP | EA | 1 | 2 |

## ORA — *ORA "OR" memory with accumulator* — ORA

Operation: A V M → A

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| √ | √ | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Immediate | ORA #Oper | 09 | 2 | 2 |
| Zero Page | ORA Oper | 05 | 2 | 3 |
| Zero Page, X | ORA Oper, X | 15 | 2 | 4 |
| Absolute | ORA Oper | 0D | 3 | 4 |
| Absolute, X | ORA Oper, X | 1D | 3 | 4* |
| Absolute, Y | ORA Oper, Y | 19 | 3 | 4* |
| (Indirect, X) | ORA (Oper, X) | 01 | 2 | 6 |
| (Indirect), Y | ORA (Oper), Y | 11 | 2 | 5 |

* Add 1 on page crossing

## PHA — *PHA Push accumulator on stack* — PHA

Operation : A ↓

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| – | – | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Implied | PHA | 48 | 1 | 3 |

## PHP — *PHP Push processor status on stack* — PHP

Operation: P ↓

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| – | – | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Implied | PHP | 08 | 1 | 3 |

## PLA — *PLA Pull accumulator from stack* — PLA

Operation: A ↑

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| √ | √ | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Implied | PLA | 68 | 1 | 4 |

**PLP** *PLP Pull processor status from stack* **PLP**

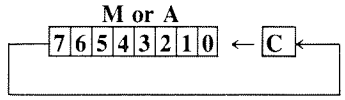
Operation: P ↑

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| From Stack | | | | | |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Implied | PLP | 28 | I | 4 |

**ROL** *ROL Rotate one bit left (memory or accumulator)* **ROL**

Operation:

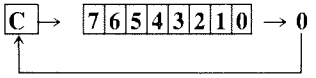



| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| √ | √ | √ | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Accumulator | ROL A | 2A | 1 | 2 |
| Zero Page | ROL Oper | 26 | 2 | 5 |
| Zero Page, X | ROL Oper, X | 36 | 2 | 6 |
| Absolute | ROL Oper | 2E | 3 | 6 |
| Absolute, X | ROL Oper, X | 3E | 3 | 7 |

**ROR** *ROR Rotate one bit right (memory or accumulator)* **ROR**

Operation:




| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| √ | √ | √ | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Accumulator | ROR A | 6A | 1 | 2 |
| Zero Page | ROR Oper | 66 | 2 | 5 |
| Zero Page, X | ROR Oper, X | 76 | 2 | 6 |
| Absolute | ROR Oper | 6E | 3 | 6 |
| Absolute, X | ROR Oper,X | 7E | 3 | 7 |

## RTI — *RTI Return from interrupt* — RTI

Operation: P ↑ PC ↑

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| From Stack | | | | | |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Implied | RTI | 40 | 1 | 6 |

## RTS — *RTS Return from subroutine* — RTS

Operation: PC ↑ , PC + 1 → PC

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| – | – | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Implied | RTS | 60 | 1 | 6 |

## SBC — *SBC Subtract memory from accumulator with borrow* — SBC

Operation: A – M – $\bar{C}$ → A
Note: $\bar{C}$ = Borrow

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| √ | √ | √ | – | – | √ |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Immediate | SBC #Oper | E9 | 2 | 2 |
| Zero Page | SBC Oper | E5 | 2 | 3 |
| Zero Page, X | SBC Oper, X | F5 | 2 | 4 |
| Absolute | SBC Oper | ED | 3 | 4 |
| Absolute, X | SBC Oper, X | FD | 3 | 4* |
| Absolute, Y | SBC Oper, Y | F9 | 3 | 4* |
| (Indirect, X) | SBC (Oper, X) | E1 | 2 | 6 |
| (Indirect), Y | SBC (Oper), Y | F1 | 2 | 5* |

* Add 1 when page boundary is crossed.

## SEC — *SEC Set carry flag* — SEC

Operation: 1 → C

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| – | – | 1 | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Implied | SEC | 38 | 1 | 2 |

## SED — *SED Set decimal mode* — SED

Operation: 1 → D

| N | X | C | I | D | V |
|---|---|---|---|---|---|
| – | – | – | – | 1 | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Implied | SED | F8 | 1 | 2 |

## SEI — *SEI Set interrupt disable status* — SEI

Operation: 1 → I

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| – | – | – | 1 | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Implied | SEI | 78 | 1 | 2 |

## STA — *STA Store accumulator in memory* — STA

Operation: A → M

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| – | – | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Zero Page | STA Oper | 85 | 2 | 3 |
| Zero Page, X | STA Oper, X | 95 | 2 | 4 |
| Absolute | STA Oper | 8D | 3 | 4 |
| Absolute, X | STA Oper, X | 9D | 3 | 5 |
| Absolute, Y | STA Oper, Y | 99 | 3 | 5 |
| (Indirect, X) | STA (Oper, X) | 81 | 2 | 6 |
| (Indirect), Y | STA (Oper), Y | 91 | 2 | 6 |

## STX — *STX Store index X in memory* — STX

Operation: X → M

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| – | – | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Zero Page | STX Oper | 86 | 2 | 3 |
| Zero Page, Y | STX Oper, Y | 96 | 2 | 4 |
| Absolute | STX Oper | 8E | 3 | 4 |

## STY — *STY Store index Y in memory* — STY

Operation: Y → M

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| – | – | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Zero Page | STY Oper | 84 | 2 | 3 |
| Zero Page, X | STY Oper, X | 94 | 2 | 4 |
| Absolute | STY Oper | 8C | 3 | 4 |

## TAX — *TAX Transfer accumulator to index X* — TAX

Operation: A → X

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| √ | √ | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Implied | TAX | AA | 1 | 2 |

## TAY — *TAY Transfer accumulator to index Y* — TAY

Operation: A → Y

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| √ | √ | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Implied | TAY | A8 | 1 | 2 |

**TSX** *TSX Transfer stack pointer to index X* **TSX**

Operation: S → X

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| √ | √ | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Implied | TSX | BA | 1 | 2 |

**TXA** *TXA Transfer index X to accumulator* **TXA**

Operation: X → A

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| √ | √ | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Implied | TXA | 8A | 1 | 2 |

**TXS** *TXS Transfer index X to stack pointer* **TXS**

Operation: X → S

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| – | – | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Implied | TXS | 9A | 1 | 2 |

**TYA** *TYA Transfer index Y to accumulator* **TYA**

Operation: Y → A

| N | Z | C | I | D | V |
|---|---|---|---|---|---|
| √ | √ | – | – | – | – |

| ADDRESSING MODE | ASSEMBLY LANGUAGE FORM | OP CODE | NO. BYTES | NO. CYCLES |
|---|---|---|---|---|
| Implied | TYA | 98 | 1 | 2 |

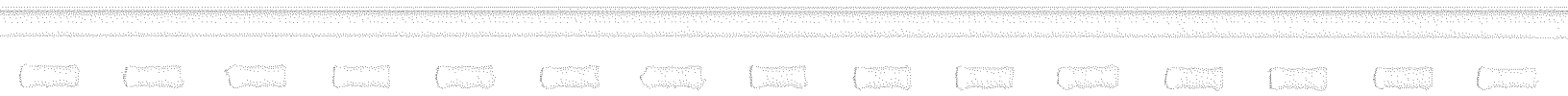
## INSTRUCTION ADDRESSING MODES AND RELATED EXECUTION TIMES (in clock cycles)

| | ACCUMULATOR | IMMEDIATE | ZERO PAGE | ZERO PAGE, X | ZERO PAGE, Y | ABSOLUTE | ABSOLUTE, X | ABSOLUTE, Y | IMPLIED | RELATIVE | (INDIRECT, X) | (INDIRECT),Y | ABSOLUTE INDIRECT |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ADC | . | 2 | 3 | 4 | . | 4 | 4* | 4* | . | . | 6 | 5* | . |
| AND | . | 2 | 3 | 4 | . | 4 | 4* | 4* | . | . | 6 | 5* | . |
| ASL | 2 | . | 5 | 6 | . | 6 | 7 | . | . | . | . | . | . |
| BCC | . | . | . | . | . | . | . | . | . | 2** | . | . | . |
| BCS | . | . | . | . | . | . | . | . | . | 2** | . | . | . |
| BEQ | . | . | . | . | . | . | . | . | . | 2** | . | . | . |
| BIT | . | . | 3 | . | . | 4 | . | . | . | . | . | . | . |
| BMI | . | . | . | . | . | . | . | . | . | 2** | . | . | . |
| BNE | . | . | . | . | . | . | . | . | . | 2** | . | . | . |
| BPL | . | . | . | . | . | . | . | . | . | 2** | . | . | . |
| BRK | . | . | . | . | . | . | . | . | . | . | . | . | . |
| BVC | . | . | . | . | . | . | . | . | . | 2** | . | . | . |
| BVS | . | . | . | . | . | . | . | . | . | 2** | . | . | . |
| CLC | . | . | . | . | . | . | . | . | 2 | . | . | . | . |
| CLD | . | . | . | . | . | . | . | . | 2 | . | . | . | . |
| CLI | . | . | . | . | . | . | . | . | 2 | . | . | . | . |
| CLV | . | . | . | . | . | . | . | . | 2 | . | . | . | . |
| CMP | . | 2 | 3 | 4 | . | 4 | 4* | 4* | . | . | 6 | 5* | . |
| CPX | . | 2 | 3 | . | . | 4 | . | . | . | . | . | . | . |
| CPY | . | 2 | 3 | . | . | 4 | . | . | . | . | . | . | . |
| DEC | . | . | 5 | 6 | . | 6 | 7 | . | . | . | . | . | . |
| DEX | . | . | . | . | . | . | . | . | 2 | . | . | . | . |
| DEY | . | . | . | . | . | . | . | . | 2 | . | . | . | . |
| EOR | . | 2 | 3 | 4 | . | 4 | 4* | 4* | . | . | 6 | 5* | . |
| INC | . | . | 5 | 6 | . | 6 | 7 | . | . | . | . | . | . |
| INX | . | . | . | . | . | . | . | . | 2 | . | . | . | . |
| INY | . | . | . | . | . | . | . | . | 2 | . | . | . | . |
| JMP | . | . | . | . | . | 3 | . | . | . | . | . | . | 5 |
| JSR | . | . | . | . | . | 6 | . | . | . | . | . | . | . |
| LDA | . | 2 | 3 | 4 | . | 4 | 4* | 4* | . | . | 6 | 5* | . |
| LDX | . | 2 | 3 | . | 4 | 4 | . | 4* | . | . | . | . | . |
| LDY | . | 2 | 3 | 4 | . | 4 | 4* | . | . | . | . | . | . |
| LSR | 2 | . | 5 | 6 | . | 6 | 7 | . | . | . | . | . | . |
| NOP | . | . | . | . | . | . | . | . | 2 | . | . | . | |
| ORA | . | 2 | 3 | 4 | . | 4 | 4* | 4* | . | . | 6 | 5* | . |
| PHA | . | . | . | . | . | . | . | . | 3 | . | . | . | . |
| PHP | . | . | . | . | . | . | . | . | 3 | . | . | . | . |

| | ACCUMULATOR | IMMEDIATE | ZERO PAGE | ZERO PAGE, X | ZERO PAGE, Y | ABSOLUTE | ABSOLUTE, X | ABSOLUTE, Y | IMPLIED | RELATIVE | (INDIRECT, X) | (INDIRECT), Y | ABSOLUTE INDIRECT |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **PLA** | . | . | . | . | . | . | . | . | 4 | . | . | . | . |
| **PLP** | . | . | . | . | . | . | . | . | 4 | . | . | . | . |
| **ROL** | 2 | . | 5 | 6 | . | 6 | 7 | . | . | . | . | . | . |
| **ROR** | 2 | . | 5 | 6 | . | 6 | 7 | . | . | . | . | . | . |
| **RTI** | . | . | . | . | . | . | . | . | 6 | . | . | . | . |
| **RTS** | . | . | . | . | . | . | . | . | 6 | . | . | . | . |
| **SBC** | . | 2 | 3 | 4 | . | 4 | 4* | 4* | . | . | 6 | 5* | . |
| **SEC** | . | . | . | . | . | . | . | . | 2 | . | . | . | . |
| **SED** | . | . | . | . | . | . | . | . | 2 | . | . | . | . |
| **SEI** | . | . | . | . | . | . | . | . | 2 | . | . | . | . |
| **STA** | . | . | 3 | 4 | . | 4 | 5 | 5 | . | . | 6 | 6 | . |
| **STX** | . | . | 3 | . | 4 | 4 | . | . | . | . | . | . | . |
| **STY** | . | . | 3 | 4 | . | 4 | . | . | . | . | . | . | . |
| **TAX** | . | . | . | . | . | . | . | . | 2 | . | . | . | . |
| **TAY** | . | . | . | . | . | . | . | . | 2 | . | . | . | . |
| **TSX** | . | . | . | . | . | . | . | . | 2 | . | . | . | . |
| **TXA** | . | . | . | . | . | . | . | . | 2 | . | . | . | . |
| **TXS** | . | . | . | . | . | . | . | . | 2 | . | . | . | . |
| **TYA** | . | . | . | . | . | . | . | . | 2 | . | . | . | . |

* Add one cycle if indexing across page boundary
** Add one cycle if branch is taken. Add one additional if branching operation crosses page boundary

A clock cycle is the speed at which the processor operates as determined by the number of bytes transferred from one internal logic component to another. The 8502 operates at a default speed of 1 MHz, which is equivalent to 1,000,000 cycles per second.



# 6

# HOW TO ENTER MACHINE LANGUAGE PROGRAMS INTO THE COMMODORE 128

Now that you know about addressing modes, types of instructions and opcodes, you need to know how to actually enter machine language instructions into the Commodore 128 memory. The C128 offers three methods of inputting instructions so that they may be operated on by the microprocessor. You can enter machine language instructions by:

1. Using the built-in machine language monitor (available in C128 mode only).
2. **POKEing** the translated decimal opcode values into memory with a BASIC program (C128 and C64 modes).
3. Using an additional software program called an *assembler*.

All three methods have advantages and disadvantages. For instance, the built-in machine language monitor is easy to use and allows you to program in machine language without any additional aids such as an assembler. It makes merging BASIC and machine language easy. In addition, you can save machine language programs as binary files with the monitor **SAVE** command. Since you are already working in an object code, there is no need to compile from source code into an object code, as is necessary with an assembler.

Though these are powerful features, the monitor does not allow the use of symbolic operand names or commented code. The monitor produces executable (object) code; hence, no source files are produced. The resulting coded program contains actual (absolute) address references, whereas an assembler source code file allows the use of symbolic addresses and arguments as well as comments. When you display a machine language program in the monitor, you do not have the luxury of comments or symbolic address variables, so you really have to know what you are looking for when reading other people's code. On the other hand, an assembler source file must be compiled into executable object code, then used often with an additional program called a *loader*. This requires three steps, whereas the monitor's machine language is ready to run as soon as you finish writing the program.

The second method, POKEing translated decimal opcode data into memory with a BASIC program, is an alternative usually implemented only when the first two options are not available. This is the case if you have no assembler and are writing a machine language routine in Commodore 64 mode, which does not make the built-in monitor available to you. However, it is sometimes handy to POKE small routines from BASIC if the application program you are writing is more suited for BASIC and you need the speed of machine language for only a small portion of the program (though for the most part, this method is tedious, bulky and time-consuming). Use it only if you have no alternative, since once it is POKED into memory, you cannot display a listing of the machine language routine as in the monitor or the assembler.

This chapter explains how to enter machine language programs in the first two methods described above. The third method, using an assembler, requires an additional software package similar to the Commodore 64 Assembler Development System. For specific details on how to enter machine language programs with the assembler, refer to the manual that is packed with the assembler software package you buy.

# ENTERING MACHINE LANGUAGE INSTRUCTIONS IN THE MONITOR

Begin entering machine language instructions by entering the monitor from BASIC with the following command:

MONITOR **RETURN**

The Commodore 128 responds with the following display:

```
MONITOR
PC      SR AC XR YR SP
; FB000 00  00  00  00  F8
```

These values indicate the contents of the microprocessor registers upon entering the monitor. The abbreviations and definitions of the register names are as follows:

| | |
|---|---|
| **PC—Program Counter** | Marks the address of the current machine language instruction |
| **SR—Status Register** | Flags that alert the microprocessor of certain conditions |
| **AC—Accumulator** | Register for all mathematical operations |
| **XR—X Index Register** | Used for effective address modification |
| **YR—Y Index Register** | Same as X register |
| **SP—Stack Pointer** | Indicates the address of the first available memory location on the stack |

Now you can begin to enter machine language instructions. The ASSEMBLE command within the monitor enters the instructions into the specified memory location. To enter instructions, follow the format of this example:

```
A 01800 LDA #$00
```

Make sure to leave at least one space between each of the fields. Here's what each part of the instruction means:

<Assemble> <Address in memory where opcode is stored> <Opcode> <Operand>

The A stands for **ASSEMBLE an opcode.** The second part (field) is the address where the opcode in the instruction is placed in the Commodore 128 memory. Notice the 5-digit hexadecimal number specifying the address. The leftmost digit (0–F) specifies the configuration of the Commodore 128 memory layout. This is the same as the BANK command in BASIC.

Once the entire machine language program is entered, reference the address that is contained in the first instruction you entered to start execution of the program. Execute the program with the **GO** command in the monitor, or exit the monitor with the **X (EXIT)** command and issue the **SYS** command from BASIC. If you SYS to the start of the program, you must use the decimal equivalent of the hexadecimal address, which

appears in the first instruction you entered. You must have an **RTS** instruction at the end of the routine if you want to return to BASIC. Often, the Kernal must be resident in the current configuration in context in order to obtain results.

The opcode is the 8502 instruction that is carried out by the microprocessor when your program is running. See the 8502 Instruction Set Table in Chapter 5 for allowable instructions.

The operand is the address or value that is acted upon by the opcode in the instruction. If the operand field is preceded by a pound sign (#), the opcode will act upon a constant value. If no pound sign is specified, the microprocessor assumes the opcode will act upon an address.

Remember to separate each field in the instruction with at least one space. If you don't, the computer indicates that an error has occurred by displaying a question mark at the end of the instruction.

Once a routine is displayed on the screen, the monitor allows shortcuts in entering instructions. To display a listing of a machine language program, issue the DISASSEMBLE command as follows:

D 04000 04010 **RETURN**

The "D" stands for disassemble. The first number (04000) specifies the starting memory location in which you want the contents displayed. The second number specifies the end address in which to display.

Now for the shortcut. Since the address where the opcodes are stored is already on the screen, you can simply move the cursor to the opcode field, type over the existing opcode and operand on the screen, erase any unwanted characters and press **RETURN** . The computer registers the instruction in memory by displaying the hexadecimal values for the opcode and operand directly to the left of the opcode mnemonic you just entered. This is a faster and easier way of entering machine-language routines, rather than typing the ASSEMBLE command and the address each time you enter an instruction.

## EXECUTING (RUNNING) YOUR MACHINE-LANGUAGE PROGRAM

Once you have finished entering your machine language routine, you may execute it in three different ways. Within the monitor, issue the GO or JUMP to Subroutine command as follows:

G F1800 (JMP)
J F1800 (JSR)

The G stands for **GO**, or go to the start address of the machine language program in memory, and begin executing it at the specified address. The value following the letter G refers to the start address of your routine. The J stands for **Jump to Subroutine**, similar to the JSR mnemonic in machine language.

The third way to invoke a machine language routine is to exit the monitor by

pressing the X key and **RETURN** . This places you back within the control of the BASIC language. Next, issue the SYS command and reference the starting address in decimal as follows:

```
BANK 15
SYS 6144
```

This SYS command is the same as the GO command (G F1800) example above. The BANK 15 command and the leading F in the 5-digit hexadecimal number F1800 specify memory configuration 15. The Kernal, BASIC and other ROM code are resident in this configuration. The only difference is that it executes the machine language routine from BASIC, instead of within the monitor.

The machine language routine given below clears the text screen. Starting at location 1024 ($0400), the value 32 ($20) is stored in each screen location. The character string value 32 is the space character, which blanks out each character position on the screen. When finished, an RTS instruction returns control to BASIC. Here's the main BASIC program and the machine language screen-clear subroutine as it appears in the machine language monitor.

```
10 FOR I= 1 TO 25
20 PRINT"FILL THE SCREEN WITH CHARACTERS"
30 NEXT
40 PRINT:PRINT
50 PRINT"NOW CALL THE MACHINE LANGUAGE"
60 PRINT" ROUTINE TO CLEAR THE SCREEN"
70 SLEEP 5
80 SYS DEC("1800")
90 PRINT"THE SCREEN IS NOW CLEARED"
```

```
READY.

MONITOR
    PC   SR AC XR YR SP
; FB000 00 00 00 00 F8

. 01800  A2 00    LDX #$00
. 01802  A9 20    LDA #$20
. 01804  9D 00 04 STA $0400,X
. 01807  9D 00 05 STA $0500,X
. 0180A  9D 00 06 STA $0600,X
. 0180D  9D E7 06 STA $06E7,X
. 01810  E8       INX
. 01811  D0 F1    BNE $1804
. 01813  60       RTS
```

In this sample program, the SYS command executes the subroutine to clear the text screen. Once the text screen is cleared, control of the microprocessor is returned to BASIC by the RTS instruction, and the READY prompt is displayed.

# MACHINE LANGUAGE MONITOR COMMANDS

The C128's built-in machine language monitor has several additional commands that manipulate your machine language routines once they are entered into memory. Figure 6–1 is a summary of all the commands available to you in the machine language MONITOR.

| KEYWORD | FUNCTION | | FORMAT |
|---|---|---|---|
| ASSEMBLE | Assembles a line of 8502 code | A | <start address> <opcode> [operand] |
| COMPARE | Compares two sections of memory and reports differences | C | <start address> <end address> <new start address> |
| DISASSEMBLE | Disassembles a line or lines of 8502 code | D | [<start address> <end address>] |
| FILL | Fills a range of memory with specified byte | F | <start address> <end address> <byte> |
| GO | Starts execution at the specified address | G | [address] |
| HUNT | Hunts through memory within a specified range for all occurrences of a set of bytes | H | <start address> <end address> <byte1> [<byte n> . . .] |
| | | H | <start address> <end address> <ascii string> |
| GOSUB | Jumps to the subroutine | J | [address] |
| LOAD | Loads a file from tape or disk | L | "<filename>"[,<device #> [,<load address>]] |
| MEMORY | Displays the hexadecimal values of memory locations | M | [<start address> [<end address>]] |
| REGISTERS | Displays the 8502 registers | R | |
| SAVE | Saves to tape or disk | S | "<filename>",<device #>, <start address> <last address +1> |
| TRANSFER | Transfers code from one section of memory to another | T | <start address> <end address> <new start address> |
| VERIFY | Compares memory with tape or disk | V | "<filename>"[,<device #>[, <load address>]] |
| EXIT | Exits Commodore 128 MONITOR | X | |
| (period) | Assembles a line of 8502 code | . | |
| (greater than) | Modifies memory | > | [address] |
| (semicolon) | Modifies 8502 register displays | ; | |

| KEYWORD | FUNCTION | FORMAT |
|---|---|---|
| (at sign) | Displays disk status, sends disk command, displays directory | @ |
| | disk status | @[device #] |
| | disk command | @[device #],<command string>] |
| | disk catalog | @[device #],$[[<drive>:<file spec>]] |

NOTES <> enclose required parameters
[] enclose optional parameters

**Figure 6–1. Summary of Commodore 128 Monitor Commands**

> **NOTE:** 5-Digit Addresses
> The Commodore 128 displays 5-digit hexadecimal addresses within the machine language monitor. Normally, a hexadecimal number is only four digits, representing the allowable address range. The extra left-most (high order) digit specifies the BANK configuration (at the time the given command is executed) according to the following memory configuration table:

| | |
|---|---|
| 0—RAM 0 only | 8—EXT ROM, RAM 0, I/O |
| 1—RAM 1 only | 9—EXT ROM, RAM 1, I/O |
| 2—RAM 2 only | A—EXT ROM, RAM 2, I/O |
| 3—RAM 3 only | B—EXT ROM, RAM 3, I/O |
| 4—INT ROM, RAM 0, I/O | C—KERNAL + INT (lo). RAM 0, I/O |
| 5—INT ROM, RAM 1, I/O | D—KERNAL + EXT (lo), RAM 1, I/O |
| 6—INT ROM, RAM 2, I/O | E—KERNAL + BASIC, RAM 0, CHARROM |
| 7—INT ROM, RAM 3, I/O | F—KERNAL + BASIC, RAM 0, I/O |

## SUMMARY OF MONITOR FIELD DESCRIPTORS

The following designators precede monitor data fields (e.g., memory dumps). When encountered as a command, these designators instruct the monitor to alter memory or register contents using the given data.

. <period> precedes lines of disassembled code.
> <right angle> precedes lines of a memory dump.
; <semicolon> precedes line of a register dump.

The following designators precede number fields (e.g., address) and specify the radix (number base) of the value. Entered as commands, these designators instruct the monitor simply to display the given value in each of the four radices.

<null> (default) precedes hexadecimal values.
$ <dollar> precedes hexadecimal (base-16) values.
+ <plus> precedes decimal (base-10) values.
& <ampersand> precedes octal (base-8) values.
% <percent> precedes binary (base-2) values.

The following characters are used by the monitor as field delimiters or line terminators (unless encountered within an ASCII string).

<space> delimiter—separates two fields.
, <comma> delimiter—separates two fields.
: <colon> terminator—logical end of line.
? <question> terminator—logical end of line.

## MONITOR COMMAND DESCRIPTIONS

The following are descriptions of each of the C128 Machine Language Monitor commands.

COMMAND: **A**
PURPOSE: Enter a line of assembly code.
SYNTAX: **A** <address> <opcode mnemonic> <operand>

| | |
|---|---|
| <address> | A number indicating the location in memory to place the opcode. (See 5-digit address note on previous page.) |
| <opcode> | A standard MOS technology assembly language mnemonic, e.g., LDA, STX, ROR. |
| <operand> | The operand, when required, can be any of the legal addresses or constants. |

A **RETURN** is used to indicate the end of the assembly line. If there are any errors on the line, a question mark is displayed to indicate an error, and the cursor moves to the next line. The screen editor can be used to correct the error(s) on that line.

**EXAMPLE:**

```
.A 01200 LDX #$00
.A 01202
```

**NOTE:** A period (.) is equal to the ASSEMBLE command.

**EXAMPLE:**

```
.02000 LDA #$23
```

COMMAND: **C**
PURPOSE: Compare two areas of memory.
SYNTAX: **C** <address 1> <address 2> <address 3>

| | |
|---|---|
| <address 1> | A number indicating the start address of the area of memory to compare against. |
| <address 2> | A number indicating the end address of the area of memory to compare against. |
| <address 3> | A number indicating the start address of the other area of memory to compare with. Addresses that do not agree are printed on the screen. |

COMMAND: **D**
PURPOSE: Disassemble machine code into assembly language mnemonics and operands.
SYNTAX: **D** [<address>] [<address 2>]

| | |
|---|---|
| <address> | A number setting the address to start the disassembly. |
| <address 2> | An optional ending address of code to be disassembled. |

The format of the disassembly differs slightly from the input format of an assembly. The difference is that the first character of a disassembly is a period rather than an A (for readability), and the hexadecimal value of the op-code is listed as well.

A disassembly listing can be modified using the screen editor. Make any changes to the mnemonic or operand on the screen, then hit the carriage return. This enters the line and calls the assembler for further modifications.

A disassembly can be paged. Typing a D **RETURN** causes the next page of disassembly to be displayed.

**EXAMPLE:**

```
D3000 3003
.03000 A9 00    LDA #$00
.03002 FF       ???
.03003 D0 2B    BNE $3030
```

COMMAND: **F**
PURPOSE: Fill a range of locations with a specified byte.
SYNTAX: **F** <address 1> <address 2> <byte>

| | |
|---|---|
| <address 1> | The first location to fill with the <byte>. |
| <address 2> | The last location to fill with the <byte>. |
| <byte value> | A 1- or 2-digit hexadecimal number to be written. |

This command is useful for initializing data structures or any other RAM area.

**EXAMPLE:**

F0400 0518 EA
Fill memory locations from $0400 to $0518 with $EA (a NOP instruction).

COMMAND: **G**
PURPOSE: Begin execution of a program at a specified address.
SYNTAX: **G** [<address>]

| | |
|---|---|
| <address> | An address where execution is to start. When address is left out, execution begins at the current PC. (The current PC can be viewed using the R command.) |

The GO command restores all registers (displayable by using the R command) and begins execution at the specified starting address. Caution is recommended in using the GO command. To return to the Commodore 128 MONITOR after executing a machine language program, use the BRK instruction at the end of the program.

**EXAMPLE:**

G 140C
Execution begins at location $140C in configuration (BANK)0. Certain applications may require that Kernal and/or I/O be present when execution begins. Precede the four-digit hexadecimal number with the hex configuration number which contains those appropriate portions of memory.)

COMMAND: **H**
PURPOSE: Hunt through memory within a specified range for all occurrences of a set of bytes.
SYNTAX: **H** <address 1> <address 2> <data>

| | |
|---|---|
| <address 1> | Beginning address of hunt procedure. |
| <address 2> | Ending address of hunt procedure. |
| <data> | Data set to search for data may be hexadecimal for an ASCII string. |

**EXAMPLE:**

H A000 A101 A9
Search for data $A9 from A000 to A101.
H2000 9800 'CASH'
Search for the alpha string "CASH".

COMMAND: **J**
PURPOSE: Jump to a machine language subroutine.
SYNTAX: **J** <address>

The JUMP to SUBROUTINE command directs program control to the machine language

subroutine located at the specified address. This command saves the return address as does the 8502 instruction JSR (Jump to Subroutine). In other words, the JUMP command is a two-way instruction, where the application gains control of the computer. Only after the subroutine encounters an RTS instruction does the machine language monitor regain control.

**EXAMPLE:**

J 2000
Jump to the subroutine starting at $2000 in configuration 0.

COMMAND: **L**
PURPOSE: Load a file from cassette or disk.
SYNTAX: **L** <"file name">[,<device>[,alt load address]]

| | |
|---|---|
| <"file name"> | Any legal Commodore 128 file name. |
| <device> | A number indicating the device to load from. 1 is cassette. 8 is disk (or 9, A, etc.). |
| [alt load address] | Option to load a file to a specified address. |

The LOAD command causes a file to be loaded into memory. The starting address is contained in the first two bytes of the disk file (a program file). In other words, the LOAD command always loads a file into the same place it was saved from. This is very important in machine language work, since few programs are completely relocatable. The file is loaded into memory until the end of file **(EOF)** is found.

**EXAMPLE:**

L "PROGRAM",8 Loads the file name PROGRAM from the disk.

COMMAND: **M**
PURPOSE: To display memory as a hexadecimal and ASCII dump within the specified address range.
SYNTAX: **M** [<address 1>] [<address 2>]

| | |
|---|---|
| <address 1> | First address of memory dump. Optional. If omitted, one page is displayed. The first digit is the bank number to be displayed, the next four digits are the first address to be displayed. |
| <address 2> | Last address of memory dump. Optional. If omitted, one page is displayed. The first digit is the bank number to be displayed; the next four digits are the ending address to be displayed. |

Memory is displayed in the following format:

>1A048 41 42 43 44 45 46 47 48:ABCDEFGH

Memory contents may be edited using the screen editor. Move the cursor to the data to be modified, type the desired correction and hit **RETURN** . If a syntax error or an attempt to modify ROM has occurred, an error flag (?) is displayed. An ASCII dump of the data is displayed in *reverse* (to contrast with other data displayed on the screen) to the right of the hex data. When a character is not printable, it is displayed as a reverse period. As with the disassembly command, paging down is accomplished by typing M and **RETURN**

**EXAMPLE:**

```
M 21C00
>21C00 41 4A 4B 4C 4D 4E 4F 50 :AJKLMNOP
```

**NOTE:** The above display is produced by the 40-column editor.

COMMAND: **R**
PURPOSE: Show important 8502 registers. The status register, the program counter, the accumulator, the X and Y index registers and the stack pointer are displayed. The data in these registers is copied into the microprocessor registers when a "G" or "J" command is issued.
SYNTAX: **R**

**EXAMPLE:**

```
 R
 PC   SR AC XR YR SP
; 01002 01  02  03  04  F6
```

**NOTE:** ; (semicolon) can be used to modify register displays in the same fashion as > can be used to modify memory registers.

COMMAND: **S**
PURPOSE: Save the contents of memory onto tape or disk.
SYNTAX: **S** <"filename">,<device>,<address 1>, <address 2>

| | |
|---|---|
| <"filename"> | Any legal Commodore 128 filename. To save the data, the file name must be enclosed in double quotes. Single quotes cannot be used. |
| <device> | A number indicating on which device the file is to be placed. Cassette is 01; disk is 08, 09, etc. |
| <address 1> | Starting address of memory to be saved. |
| <address 2> | Ending address of memory to be saved + 1. All data up to, but not including, the byte of data at this address is saved. |

The file created by this command is a program file. The first two bytes contain the starting address <address 1> of the data. The file may be recalled, using the L command.

**EXAMPLE:**

S "GAME",8,0400,0C00
Saves memory from $0400 to $0BFF onto disk.

COMMAND: **T**
PURPOSE: Transfer segments of memory from one memory area to another.
SYNTAX: **T** <address 1> <address 2> <address 3>

| | |
|---|---|
| <address 1> | Starting address of data to be moved. |
| <address 2> | Ending address of data to be moved. |
| <address 3> | Starting address of new location where data will be moved. |

Data can be moved from low memory to high memory and vice versa. Additional memory segments of any length can be moved forward or backward. An automatic "compare" is performed as each byte is transferred, and any differences are listed by address.

**EXAMPLE:**

T1400 1600 1401
Shifts data from $1400 up to and including $1600 one byte higher in memory.

COMMAND: **V**
PURPOSE: Verify a file on cassette or disk with the memory contents.
SYNTAX: **V** <"filename">[,<device>][,alt start address]

| | |
|---|---|
| <"filename"> | Any legal Commodore 128 file name. |
| <device> | A number indicating which device the file is on. Cassette is 01; disk is 08, 09, etc. |
| [alt start address] | Option to start vertification at this address. |

The VERIFY command compares a file to memory contents. If an error is found, the words VERIFY ERROR are displayed; if the file is successfully verified, the cursor reappears without any message.

**EXAMPLE:**

V"WORKLOAD",08

COMMAND: **X**
PURPOSE: Exit to BASIC.
SYNTAX: **X**

COMMAND: > (greater than)
PURPOSE: Can be used to assign values for one to eight memory locations at a time (in 40-column mode; up to 16 in 80-column mode).
SYNTAX: > <address> <data byte 1> <data byte 2 . . . 8>

| | |
|---|---|
| <address> | First memory address to set. |
| <data byte 1> | Data to be put at address. |
| <data byte 2 . . . 8> | Data to be placed in the successive memory locations following the first address (optional) with a space preceding each data byte. |

COMMAND: @ (at sign)
PURPOSE: Can be used to send commands to the disk drive.
SYNTAX: @ [<device number>], <disk cmd string>

| | |
|---|---|
| <device number> | Device unit number (optional). |
| <disk cmd string> | String command to disk. |

> **NOTE:** @ alone gives the status of the disk drive.

**EXAMPLES:**

| | |
|---|---|
| @ | checks disk status<br>00, OK, 00, 00 |
| @,I | initializes drive 8 |
| @,$ | displays disk directory on unit 8. |
| @,$0:F* | display all files on Drive 0, unit 8 starting with the letter F. |

As a further aid to programmers, the Kernal error message facility has been automatically enabled, while in the Monitor. This means the Kernal will display 'I/O ERROR#' and the error code, should there be any failed I/O attempt from the MONITOR. The message facility is turned off when exiting the MONITOR.

# MANIPULATING TEXT WITHIN THE MACHINE LANGUAGE MONITOR

Certain machine language application programs require the manipulation of strings of characters. If you are using an assembler package, it contains provisions for handling strings of characters. However, within the monitor, strings of characters must be placed in memory, either (1) through modifying a memory dump using the screen editor, or (2)

by placing the ASCII values of the characters in memory locations within a program.

To modify a memory dump using the screen editor, issue the **MEMORY** command with the address range in which you want to place the character string information. For example, suppose you want to place the word "TEXT" in memory starting at location $2000. First, enter the machine language monitor with the MONITOR command. Next, issue the memory command containing the address $2000 as follows:

M 2000

The 128 responds with this display:

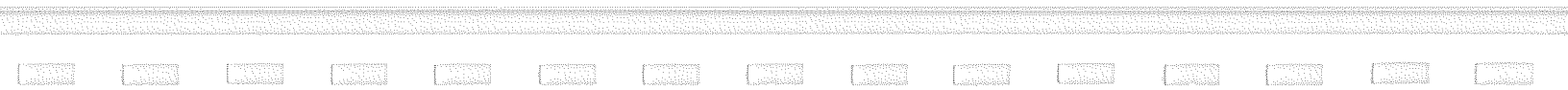
*02000 FF 00 FF 00 FF 00 FF 00: π.π.π.π.*

The entire screen is filled with the contents of the memory (dump) locations $2000 through $205F. For illustrative purposes, only one line of the memory dump is shown. This line pertains to the address range $2000 through $2007. At the right of the screen is an area that displays the corresponding ASCII character for each value within a memory location in that line of the memory dump. The left character in the display area corresponds to location $2000, the second character position to the right pertains to address $2001, and so on. To place the word "TEXT" in memory starting at location $2000, move the cursor up to the first line of the memory dump, move the cursor right to the memory address that pertains to address $2000, and place the ASCII character string code for the letter T in this position. To do this, type over the characters that are there and replace them with the hexadecimal equivalent of decimal 84 ($54) and press **RETURN** . Notice that the letter T is now displayed at the right of the screen. Refer to Appendix E, ASCII and CHR$ Codes, for a list of the Commodore ASCII codes for each character available in the Commodore 128.

Now do the same procedure for the letters E, X and T. When you are through, the word "TEXT" is displayed in the display area. The first line of the memory dump now looks like this:

02000 54 45 58 54 FF 00 FF 00: TEXT π. π.

Now the character string you wish to manipulate is in memory, starting at address $2000. Your machine language routine can now act upon the characters of the word "TEXT" in order to display them on the screen. An efficient way of manipulating entire words is to use the start address in memory where the text begins, in this case $2000. Determine the length of the string, and use an index register as an offset to the end of the word. See the section Raster Interrupt Split Screen Program with Horizontal Scrolling in Chapter 8, for a working example of manipulating text. This chapter has described the use of machine language. For additional information on machine language topics, see Chapter 5, 7, 8, 9, 10, 11, and 13.



# 7

# MIXING MACHINE LANGUAGE AND BASIC

# WHY MIX BASIC AND MACHINE LANGUAGE?

Certain application programs are better suited for a high-level language such as BASIC rather than low-level machine language. In other cases, however, certain portions of a program, such as displaying graphics, may require the speed of machine language while the rest of the program lends itself to the use of BASIC. This is the main reason for mixing BASIC programs with machine language subroutines. Another reason may be the lack of an alternative in programming machine language. For example, in C64 mode, a machine language monitor is not ordinarily available to the user. In addition, you may not have an assembler package, so the only alternative is to enter machine language programs through the BASIC language. This method has disadvantages; it can be tedious and time-consuming, and once the routine is entered into memory, you have no way of listing it to the screen for editing. This method is recommended only if no alternative is available.

# ENTERING MACHINE LANGUAGE SUBROUTINES THROUGH BASIC

In Chapter 6, you saw an example of how to use the SYS command to go from BASIC to a machine language routine that cleared the text screen. The SYS command invoked the subroutine, cleared the screen, and returned to BASIC with the RTS instruction. This example illustrated the SYS command within a program. You can also SYS to a machine language subroutine outside a BASIC program, as follows:

```
SYS 8192
```

This example assumes you entered the machine language subroutine through the monitor (and it is still in memory). What if you don't have the monitor available to you, as in C64 mode, and you want to mix a machine language subroutine with a BASIC program?

The answer to this is to POKE decimal data that represents the hexadecimal opcodes and operands into memory. To activate the subroutine, you SYS to it as you did before. This method of entering machine language programs requires these steps:

1. Write your machine language program on a piece of paper.
2. Translate the hexadecimal op-code values into decimal. Some instructions require 3 bytes of memory, while others only use 1 or 2 bytes of memory. For a list of hexadecimal opcodes for the 8502 machine language instruction set, see the 8502 Instruction and Addressing Table, in Chapter 5.
3. Enter the decimal equivalents of the opcodes into DATA statements in such a way that a 2-byte instruction, for example, is entered as follows:

```
1000 DATA 162,0: REM = LDX #$00 = $A2, $00
```

The hexadecimal number $A2 represents the 8502 instruction for LDX, which equals 162 in decimal. The 0 (zero) represents the operand 0, which is loaded into the X register in the instruction. In hex, 0 is the same as in decimal, so the second byte of the instruction is the operand value 0. The hexadecimal opcodes are translated into strings of binary digits (bits), so the microprocessor can interpret and operate them. This is the true machine language of the computer at its lowest level.

4. Once you have translated all the opcodes and operands into decimal, you must place them in memory. Accomplish this by READing a DATA value in BASIC and POKEing it into memory in an appropriate address range. For example, to enter the LDX #$00 instruction into memory, in C128 mode, perform the following routine:

```
10 ALPHA = 8192
20 I = 0
30 DO
40 READ A
45 IF A=-999 THEN EXIT
50 POKE ALPHA+I,A
60 I =I+1
70 LOOP
80 PRINT "ALL DATA IS NOW IN MEMORY"
:
:
1000 DATA 162,0,-999
```

For C64 mode use this routine:

```
10 ALPHA = 8192
20 FOR I=0 TO 1
40 READ A
50 POKE ALPHA+I,A
60 NEXT
80 PRINT "ALL DATA IS NOW IN MEMORY"
:
:
1000 DATA 162,0
```

The first example, in C128 mode, READs all DATA and POKEs it into memory, starting at location 8192 ($2000), until a data item is equal to –999. When the data item equals –999, EXIT the loop and print a message saying all the data is read into memory at the specified locations. This **DO . . . LOOP** allows you to enter as many data items as you please without the need to know how many data items are in the data list, since it checks for the terminator value –999. This programming approach makes it easy to modify the number of data entries without having to change any of the code. This program assumes that the bit map screen is not being used ($2000–$3FFF).

The second example, in C64 mode, uses a **FOR. . . NEXT** loop to enter the data. This requires you to know how many data items are in the data list. You can add an **IF . . . THEN** statement to check for a terminator value like –999, as in the first example. However, this method illustrates a different way of accomplishing the same thing.

5. The final step in entering machine language subroutines through BASIC is executing the subroutine with the SYS command. Add line 90 to your BASIC routines above as follows:

   ```
   90 SYS 8192
   ```

   This command executes the machine language routine that you POKEd into memory starting at location 8192.

Although the DATA statement in the example in Step 4 does not show it, all machine language subroutines must end with an RTS instruction so you can return to BASIC. The decimal code for an RTS machine language instruction is 96. Your last decimal data item in the final data statement in your program must be 96, unless you use a terminator like –999; then –999 will be your last decimal data item.

Figure 7–1 shows a step-by-step translation from the machine language screen-clear routine as it appears in the monitor and a complete program that mixes the clear screen routine with the BASIC program that POKEs in the data and executes the machine language subroutine. It only operates in the 40-column (VIC) screen.

| Address | Hex Opcode | Symbolic Instruction | | Decimal Equivalent | | |
|---|---|---|---|---|---|---|
| | | | | 1st Byte (Opcode) | 2nd Byte (Operand) | 3rd Byte |
| . 02000 | A2 00 | LDX #$00 | = | 162 | 0 | |
| . 02002 | A9 20 | LDA #$20 | = | 169 | 32 | |
| . 02004 | 9D 00 04 | STA $0400,X | = | 157 | 0 | 4 |
| . 02007 | 9D 00 05 | STA $0500,X | = | 157 | 0 | 5 |
| . 0200A | 9D 00 06 | STA $0600,X | = | 157 | 0 | 6 |
| . 0200D | 9D E7 06 | STA $06E7,X | = | 157 | 231 | 6 |
| . 02010 | E8 | INX | = | 232 | | |
| . 02011 | D0 F1 | BNE $2004 | = | 208 | 241 | |
| . 02013 | 60 | RTS | = | 96 | | |

**Figure 7–1. Step-by-step Translation into Decimal**

To find the hexadecimal opcodes, refer to the 8502 Instruction and Addressing Table in Chapter 5. Notice in Figure 7-1 that the hexadecimal opcodes are displayed within the monitor, directly to the left of the symbolic instruction. These hexadecimal numbers are the codes that you translate into decimal data items in a BASIC program.

Notice that the second byte in the BNE instruction is the value 241. In a branch instruction, the operand is not an absolute address, but is instead an offset to the instruction to which it will branch. In this case, the BNE instruction branches backward to location $2004; therefore, it branches backward 15 locations in memory to the first store **(STA)** instruction.

You're probably wondering how the code 241 tells the computer to branch backward by 15. The number 241 (decimal) is the 2's complement of the value 15. When bit 7 is enabled, the microprocessor branches backward. The number 241 signifies to branch backward by 15 memory locations and execute the instruction in that location. To find the root value of a 2's complement number, do this:

1. Translate the value into binary: 241 = 1111 0001
2. Subtract 1: = 1111 0000 = 240
3. Convert each bit to its complement; in other words, change each bit to the opposite value: = 0000 1111 = 15

To find the two's complement of a number, perform steps 1 and 3 above; then *add* 1 to the value instead of subtracting.

Here's the complete C128 mode program, including all the decimal DATA equivalents of the instructions in the machine language, clear-screen subroutine above:

```
10 ALPHA=8192
20 I=0
30 DO
40 :       READ A
45 :       IF A=-999 THEN EXIT
50 :       POKE ALPHA+I,A
60 :       I=I+1
70 LOOP
80 PRINT"ALL DATA IS NOW IN MEMORY"
85 SLEEP 1
90 SYS 8192
1000 DATA 162,0,169,32,157,0,4,157,0,5,157,0,6,157,231,6
2000 DATA 232,208,241,96,-999
```

Here is the corresponding program in C64:

```
10 ALPHA=8192
20 FOR I=0 TO 19
40 :       READ A
50 :       POKE ALPHA+I,A
60 NEXT
80 PRINT"ALL DATA IS NOW IN MEMORY"
85 FOR I=1 TO 2500:NEXT
90 SYS 8192
1000 DATA 162,0,169,32,157,0,4,157,0,5,157,0,6,157,231,6
2000 DATA 232,208,241,96
```

When you run this program, the computer READs the DATA, POKEs it into memory, and executes the machine language, clear-screen subroutine with the SYS command. After you RUN the program, enter the machine language monitor (assuming you are currently in C128 mode) and disassemble the code in the range $2000 through $2015 with this command:

D 2000 2015

Notice that the subroutine you POKEd in through BASIC is the same as the subroutine that appears in Figure 7–1. The two different methods accomplish the same goal—programming in 8502 machine language.

## WHERE TO PLACE MACHINE LANGUAGE PROGRAMS IN MEMORY

The Commodore 128 has 128K of RAM memory, divided into two 64K RAM banks. Much of the 128K of RAM is overlaid by ROM, but not at the same time. The Commodore 128 memory is layered, so RAM is beneath the overlaid ROM. The designers of the Commodore 128 have managed to squeeze 28K of ROM and 128K of RAM into 128K of address space. Only one bank is available or mapped in at a time, since the highest address an 8-bit microprocessor can address is 65535 ($FFFF). However, because the C128 is capable of banking RAM and ROM in and out so fast, it may seem as though 128K is always available.

In the portions of memory shared by RAM and ROM, a read operation returns a ROM data value and a write operation "bleeds through" to the RAM beneath the layered ROM. The data is stored in the RAM memory location. If the data in RAM beneath the ROM is a program, the ROM on top must be switched out before the program in RAM can be executed. The RAM and ROM layout in memory is all regulated and controlled through the **Configuration Register (CR)** of the **Memory-**

**Management Unit (MMU)**. For detailed information, refer to the sections on the Registers of the Memory Management Unit (specifically, the discussion of the Configuration Register) in Chapter 13.

## WHERE TO PLACE MACHINE LANGUAGE ROUTINES IN CONJUNCTION WITH BASIC

Within BASIC, the operating system takes care of the mapping in and out of ROM and RAM. The C128 operating system provides sixteen (default) memory configurations. They each contain different values in the Configuration Register; therefore, each has a different configuration. This gives you sixteen different configurations of memory to choose from. Figure 7–2 lists the sixteen default memory configurations available under the control of the BASIC language.

| BANK | CONFIGURATION |
|---|---|
| 0 | **RAM(0) only** |
| 1 | **RAM(1) only** |
| 2 | **RAM(2) only** |
| 3 | **RAM(3) only** |
| 4 | **Internal ROM, RAM(0), I/O** |
| 5 | **Internal ROM, RAM(1), I/O** |
| 6 | **Internal ROM, RAM(2), I/O** |
| 7 | **Internal ROM, RAM(3), I/O** |
| 8 | **External ROM, RAM(0), I/O** |
| 9 | **External ROM, RAM(1), I/O** |
| 10 | **External ROM, RAM(2), I/O** |
| 11 | **External ROM, RAM(3), I/O** |
| 12 | **Kernal and Internal ROM (LOW), RAM(0), I/O** |
| 13 | **Kernal and External ROM (LOW), RAM(0), I/O** |
| 14 | **Kernal and BASIC ROM, RAM(0), Character ROM** |
| 15 | **Kernal and BASIC ROM, RAM(0), I/O** |

**Figure 7–2. Bank Configuration Table**

If you want to place a machine language subroutine in memory while the BASIC language is running, put the subroutine in a bank that contains RAM, preferably bank 0 since this bank is composed entirely of RAM. If you place the machine language subroutine in a bank other than 0, not all of the RAM is available for programs, since ROM overlays some of the RAM. You must check the value of the Configuration Register *within that bank* to see which addresses within these banks contain ROM. Check the value of the Configuration Register within each of the sixteen configurations and compare the value with the table in Figure 13-5 to see exactly where ROM maps in.

Follow this procedure when calling machine language subroutines from BASIC:

1. Place the subroutine, preferably in bank 0, either through the monitor or by POKEing in the code through BASIC. If the Kernal, BASIC, and I/O are required, execute your program from configuration (BASIC Bank) 15. If you enter the subroutine through the monitor, place the routine in bank 0 by placing the digit 0 before the 4-digit hexadecimal address where the instructions are stored. If you are POKEing the codes in through BASIC (not the preferred method), issue the **BANK 0** command within your program, assuming you are placing the routine into BANK 0; then POKE in the decimal data for the opcodes and operands. The recommended memory range to place machine language routines in conjunction with BASIC is between $1300 and $1BFF. The upper part of BASIC text is also available, provided your BASIC program is small and does not overwrite your routine.
2. Now, to process the rest of your BASIC program, return to bank 15 (the default bank) with this command:

   ```
   BANK 15
   ```

3. Now call the subroutine with the SYS command. SYS to the start address where the first machine language instruction of your program is stored in memory. In this case, assume the subroutine starts at hex location $2000 (assuming the VIC bit map screen is not used) and enter:

   ```
   SYS 8192
   ```

The RAM in configuration 0 in Figure 7–2 is the same RAM that appears in configurations (BANKS) 4, 8, 12, 13, 14, and 15. For example, you can enter programs into BANK 15, but you must make sure that no ROM overlays your program area.

> **NOTE:** If you plan to return to BASIC, make sure your subroutine ends with an RTS instruction.

## WHERE TO PLACE MACHINE LANGUAGE ROUTINES WHEN BASIC IS DISABLED

When you are programming in machine language and you don't require the services of the BASIC ROM, you can disable BASIC by mapping out the BASIC ROMs. Do this by placing certain values into the Configuration Register in your own machine language routines as follows:

```
LDA #$0E   ;Set up the Configuration Register value
STA $D501  ;Write to Preconfiguration Register A
STA $FF01  ;Write to LCR A to change value of CR
```

You can use this sequence:

```
LDA #$0E
STA $FF00
```

When you switch out BASIC, the sixteen default configurations no longer exist

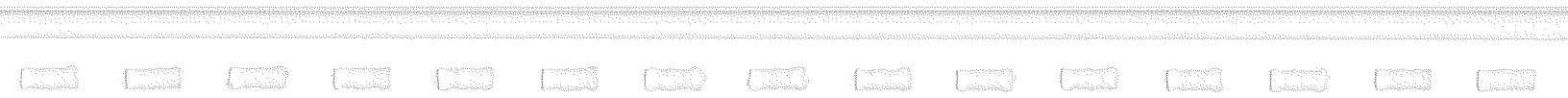
via the BASIC command, so it becomes your responsibility to manage the memory configurations by manipulating the Configuration Register in your application program. Figure 13-5, on page 462, defines the values to place in the configuration register to arrive at the different memory configurations.

When you switch out the BASIC ROMs, the address range where BASIC usually resides ($4000 through $7FFF for BASIC low and $8000 through $BFFF for BASIC high), is available for your machine language programs. Be careful when switching out the Kernal, since the Kernal controls the entire operation of the C128, including routines that seem transparent to the user (i.e., routines that you may take for granted).

At certain points within your machine language programs, you may need to disable the I/O operation of the C128 temporarily. For instance, if you want to copy portions of the character ROM into RAM when programming your own characters, you must switch out the I/O registers $D000 through $DFFF of the C128, transfer the character data into RAM, and then switch the I/O back in.

See the section discussing the Configuration Register, in Chapter 13, for a full explanation of how the C128 RAM and ROM memory is configured.

This chapter has described the use of BASIC and machine language together. For material on using BASIC alone, see Chapters 2, 3 and 4. For material on using machine language see Chapters 5, 6, 8, 9, 10, 11 and 13.



# 8

# THE POWER BEHIND COMMODORE 128 GRAPHICS

# THE RELATIONSHIP BETWEEN VIDEO BANKS, RAM BANKS AND MEMORY CONFIGURATIONS

Many of you are familiar with how the Commodore 64 manages memory. This section explains how the Commodore 128 manages video memory, and how the video banks relate to the currently selected memory configuration.

## MANAGING BANKED MEMORY

Banking is a process in which a section of memory is addressed by the microprocessor. The memory is said to be banked in when it is available to the microprocessor in the current memory configuration.

The Commodore 128 is programmable in its memory configuration. BASIC and the Machine Language Monitor give you 16 pre-programmed default configurations of memory (referred to in BASIC as *banks*). For the purposes of this discussion, BASIC banks are referred to simply as default memory configurations which are combinations of ROM and RAM in various ranges of memory. The current configuration, whether in BASIC or machine language, is determined by the value in the configuration register of the C128 Memory Management Unit (MMU) chip.

The sixteen different configurations in BASIC and the Machine Language Monitor require different values to be placed in the configuration register so that particular combinations of ROM and RAM can be banked into memory simultaneously. For example, the character ROM is only available in memory configuration 14 (Bank 14 in BASIC; the fifth digit hexadecimal prefix "E" in the Machine Language Monitor), since this configuration tells the C128 MMU to swap out the I/O registers between $D000 and $DFFF, and replace them with the character ROM. To swap the I/O capabilities back in, change to any configuration number that contains I/O. Figure 8–1 lists the sixteen default memory configurations available in BASIC and the Machine Language Monitor. Information on programming the MMU is contained in Chapter 13, The Commodore 128 Operating System.

### THE TWO 64K RAM BANKS

The Commodore 128 memory is composed of two RAM banks (labeled 0 and 1), each having 64K of RAM, giving a total of 128K. The 8502 microprocessor address bus is 16 bits wide, allowing it to address 65536 (64K) memory locations at a time. Figure 8–2 illustrates the two separate 64K RAM banks.

Although only 64K can be accessed at one time, the MMU has provisions for sharing up to 16K of common RAM between the two RAM banks. Common RAM is discussed in Chapter 13.

The 8502 microprocessor and the VIC chip can each access a different 64K RAM bank. The 8502 RAM bank is selected by the configuration register (bits 6 and 7) and

| BANK | CONFIGURATION |
|---|---|
| 0 | RAM(0) only |
| 1 | RAM(1) only |
| 2 | RAM(2) only (same as 0) |
| 3 | RAM(3) only (same as 1) |
| 4 | Internal ROM , RAM(0), I/O |
| 5 | Internal ROM , RAM(1), I/O |
| 6 | Internal ROM , RAM(2), I/O (same as 4) |
| 7 | Internal ROM , RAM(3), I/O (same as 5) |
| 8 | External ROM , RAM(0), I/O |
| 9 | External ROM , RAM(1), I/O |
| 10 | External ROM , RAM(2), I/O (same as 8) |
| 11 | External ROM , RAM(3), I/O (same as 9) |
| 12 | Kernal and Internal ROM (LOW), RAM(0), I/O |
| 13 | Kernal and External ROM (LOW), RAM(0), I/O |
| 14 | Kernal and BASIC ROM, RAM(0), Character ROM |
| 15 | Kernal and BASIC ROM, RAM(0), I/O |

**Figure 8–1. C128 Default Memory Configurations**

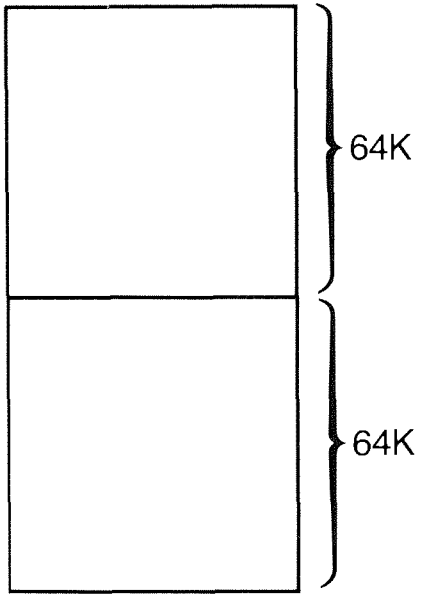



**Figure 8–2. C128 64K RAM Banks**

the VIC RAM bank is selected by the RAM configuration register (bits 6 and 7). This also is covered in detail in Chapter 13.

The configuration determined by the configuration register can be composed of RAM and ROM, where the ROM portion overlays the RAM layer underneath, as illustrated in Figure 8–3.

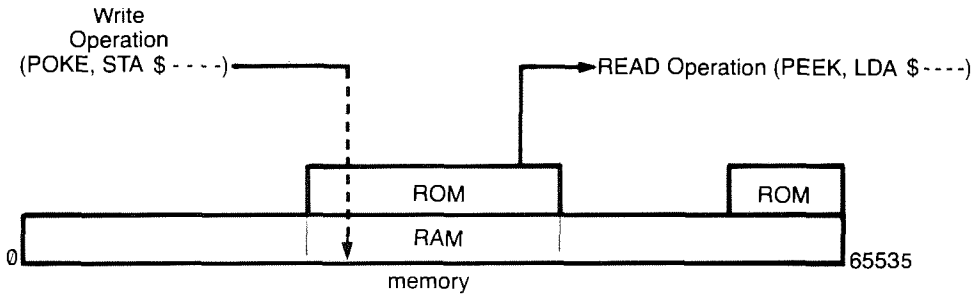



**Figure 8–3. ROM Overlay**

A read (PEEK) operation returns a ROM value, and a write (POKE) operation bypasses the ROM and stores the value in the RAM underneath.

Many different combinations of memory can be constructed to comprise a 64K configuration of accessible memory. Bits six and seven of the configuration register specify which RAM bank lies beneath the ROM layers specified by bits zero through five. The underlying RAM bank can be switched independently of any ROM layers on top. For instance, you may switch from RAM Bank 0 to RAM Bank 1, while maintaining the Kernal, BASIC and I/O.

## 16K VIDEO BANKS

In C128 graphics programming, a video bank is a 16K block of memory that contains the essential portions of memory controlling the C128 graphics system: screen and character memory. These two types of memory, which are discussed in the following section, must lie within the 16K range of memory referred to as a (VIC) video bank. The VIC chip is capable of addressing 16K of memory at any one time, so all graphics memory must be present in that 16K. Since the Commodore 128 microprocessor addresses 64K at a time, there are four video banks in each 64K RAM bank, or a total of eight video banks. (See Figure 8–4.)

Where you place the VIC video bank depends on your application program. The Commodore 128 ROM operating system expects this bank in default video Bank 0, in the bottom of RAM Bank 0. Screen and character memory may be located at different positions within each 16K video bank, though in order to successfully program the VIC chip, the current 16K bank must contain screen and character memory in their entirety. You'll understand this after reading the next few pages.

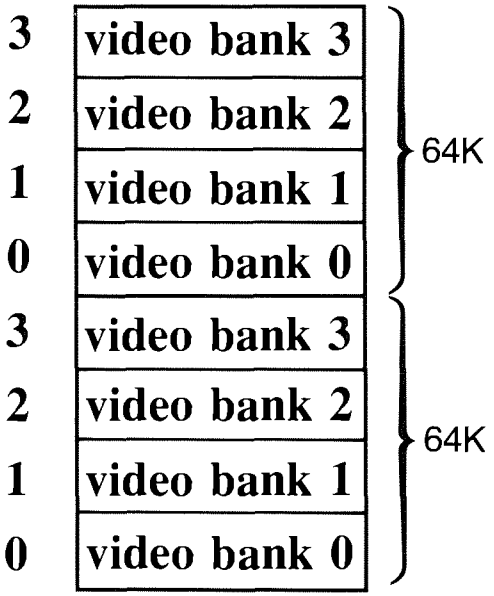



**Figure 8–4. Video Banks within RAM Banks**

The four video banks in each 64K RAM bank are set up in the memory ranges specified in Figure 8–5:

| BANK | ADDRESS RANGE | VALUE OF BITS 1 & 0 IN $DD00 | |
|---|---|---|---|
| | | BINARY | DECIMAL |
| 0 | $0–$3FFF | 11 = | 3 (DEFAULT) |
| 1 | $4000–$7FFF | 10 = | 2 |
| 2 | $8000–$BFFF | 01 = | 1 |
| 3 | $C000–$FFFF | 00 = | 0 |

**Figure 8–5. Video Banks Memory Ranges**

Each RAM bank (0 and 1) has this memory layout.

Bits 0 and 1 of location $DD00 select the video bank. To select a video bank in BASIC, type this command:

**POKE 56576, (PEEK(56576) AND 252) OR X**

Where X is the decimal value of bits 1 and 0 in Figure 8–5.

In machine language, run the following program segment to select video banks:

**LDA $DD00;** load the accumulator with contents of $DD00
**AND #$FC;** preserve the upper 6 bits of $DD00
**ORA #$X;** where X is the hex value of bits 1 and 0 from Figure 8–5 above
**STA $DD00;** Place the value in $DD00

In the third instruction, replace X with the hexadecimal value of bits 1 and 0 in Figure 8–5. The default value is $03, which selects video bank zero.

Whenever you change video banks, you must add $4000 to the address of your starting screen memory (video matrix) and character memory (bit map in bit map mode) for each bank above 0. To change to video Bank 1, add $4000 to your starting screen and character address; for Bank 2 add $8000; for Bank 3 add $C000. You must always add an offset of $4000 to the start of your screen and character memory for each video bank that is greater than zero.

## SUMMARY OF BANKING CONCEPT

The major features of the banking concept can be summarized as follows:

1. BASIC and the Machine Language Monitor have sixteen 64K memory configurations that give you sixteen different combinations of memory layouts. The MMU chip, particularly the value in the configuration register, controls most of the memory management in the Commodore 128. In order to PEEK (read) from or POKE (write) to a particular portion of memory, you must choose a BASIC or monitor configuration that contains the desired section of memory. Figure 8–1 lists the sixteen default memory configurations available in BASIC and the Machine Language Monitor.
2. The 128K of memory is divided into two 64K RAM banks. Only one bank is addressable at a time by the microprocessor. RAM bank selection is controlled by the MMU configuration register (bits 6 and 7), which is part of the C128 I/O memory. The VIC chip and 8502 microprocessor can each access a different 64K RAM bank. Figure 8–2 illustrates the two separate and independent 64K RAM banks.
3. Each 64K RAM bank is divided into four 16K video segments. The screen and character memory must both lie within the selected 16K video segment in order to successfully display graphics and characters on the screen. For each 16K video bank higher than zero, remember to add $4000 (16384 decimal) to the start address of screen and character memory. Figure 8–4 shows how four 16K video banks fit into each of the two 64K RAM banks.

Here's how the banks fit together and operate within the Commodore 128. One 64K RAM bank is always mapped into memory. Within BASIC or the Machine Language Monitor, sixteen different memory configurations are available in a 64K bank. To change the configuration, issue the BASIC BANK command, or precede the four digit hexadecimal address in the Machine Language Monitor with an additional hexadecimal digit 0 through F. Outside of BASIC or the monitor, you can select other configurations, by changing the value in the configuration register at location $FF00 (or $D500). See Chapter 13 for details.

Within the selected configuration, and part of the current 64K RAM bank, is a 16K range reserved for a video bank. The 16K video bank must encompass 1K of screen memory, and either 4K of character ROM or an 8K block of memory for the bit map data. All these components must be present in order for graphics to operate.

In essence, the bank concept can be thought of in this way: The C128 has a 16K (VIC) video bank within a selected memory configuration within a 64K RAM bank.

Figures 8–28 through 8–32 at the end of this chapter provides a graphics programming summary.

## SHADOW REGISTERS: INTERMEDIATE STORAGE LOCATIONS USED BY THE C128 SCREEN EDITOR

Users who are experienced in programming the Commodore 64 VIC chip will find that most of the graphics operations of the Commodore 128 are performed in the same way as the C64. The main difference between the Commodore 64 and the Commodore 128 graphics systems is the hardware implementation of split-screen modes.

C128 mode provides two types of split-screen displays:

1. Standard character mode and bit map mode
2. Standard character mode and multi-color bit map mode

Because the split-screens switch from one display mode to another at a given time, the screen editor must be interrupt-driven. The interrupt indicates at what point the mode is to be switched. At that point, the VIC chip is loaded with preset values already contained in RAM. These preset values are known as shadow registers. Each time an interrupt occurs, certain video-chip registers are cleared and refreshed with the values in the shadow registers. These shadow registers add a variation in programming the VIC chip compared to the way the Commodore 64 handles it.

The primary intermediate storage locations for VIC chip programming are:

| NAME | INDIRECT LOCATION | ACTUAL LOCATION | DESCRIPTION |
|---|---|---|---|
| **GRAPHM** | **BIT 7 - 216 ($00D8)** | **BIT 4 - 53270 ($D016)** | **Multicolor Mode Bit** |
| **GRAPHM** | **Bit 6 - 216 ($00D8)** | **———** | **Split Screen Bit** |
| **GRAPHM** | **BIT 5 - 216 ($00D8)** | **BIT 5 - 53265 ($D011)** | **Bit Map Mode Bit** |
| **VM1*** | **BITS 7-4 - 2604 ($0A2C)** | **BITS 7-4 - 53272 ($D018)** | **Video Matrix (screen memory) Pointer** |
| **VM1** | **BITS 3-0 - 2604 ($0A2C)** | **BITS 3-0 - 53272 ($D018)** | **Character Base Pointer** |
| **VM2**** | **BITS 7-4 - 2605 ($0A2D)** | **BITS 7-4 - 53272 ($D018)** | **Video Matrix (screen memory) Pointer** |
| **VM2** | **BITS 3-0 - 2605 ($0A2D)** | **BITS 3-0 - 53272 ($D018)** | **Bit Map Pointer** |

*VM1 applies only to standard and multi-color character (text) modes.

**VM2 applies only to standard and multi-color bit map modes.

You must store to and load from the indirect locations when accessing the above features of the VIC (8564) chip. For example, in C64 mode, this is how you set up the video matrix and bit map mode:

**10 POKE 53272, 120:** REM Select bit map @ 8192, video matrix @ 7168
**20 POKE 53265, PEEK(53265) OR 32:** REM Enter bit map mode

Line 10 sets the video matrix at 7168 ($1C00) and the bit map at 8192 ($2000). Line 20 enables bit map mode.

Normally, you would perform this operation with the high-level, 7.0 BASIC command:

**GRAPHIC 1**

The comparable way to accomplish this with POKE commands in C128 mode is as follows:

**10 POKE 2605,120**
**20 POKE 216,PEEK(216) OR 32**

In C128 machine language, use these instructions:

**LDA #$78;** set bit map @ $2000
**STA $0A2D;** set video matrix @ 7168
**LDA $00D8**
**ORA #$20**
**STA $00D8;** select bit map mode

Although these examples do more than just select bit map mode and set up the video matrix and bit map pointer (such as wait on a video retrace when the raster is off the visible coordinate plane), these examples give you an idea of how to perform these programming steps.

As you can see, C128 mode requires a slight variation in programming the VIC chip. You must keep this in mind when programming graphics in C128 mode. Usually, the high-level BASIC 7.0 commands take care of these variations. However, if you are programming in machine language, remember to address these indirect storage locations and not the actual ones. If you store values directly to the actual registers, the value will be cleared in a jiffy and no apparent action occurs.

## DISABLING THE INTERRUPT DRIVEN SCREEN EDITOR

You can disable the interrupt-driven C128 screen editor by storing the value 255 ($FF) in location 216 ($00D8). The actual VIC registers are not affected, and you can program the VIC chip the same way as the C64. This makes it unnecessary to address the indirect shadow registers. In BASIC, enter:

**POKE 216,255**

In machine language, you enter:

```
LDA #$FF
STA $00D8
```

Since disabling the interrupt allows you to program the VIC chip in the same way as the Commodore 64, you can store values directly to the actual registers. You do not have to address the indirect storage locations for VIC chip programming. However, if you don't disable the interrupt, it is still active and your values will be cleared upon the first occurrence of the raster interrupt.

Remember, you must either disable the interrupt or address the indirect storage locations. Failure to do one or the other can cause serious problems in your program.

The 80-column chip indirect memory locations are discussed in Chapter 10, Programming the 80-Column 8563 Chip. Certain other I/O functions require the use of indirect locations also. These are covered in Chapter 12, Input/Output Guide.

# THE COMMODORE 128 GRAPHICS SYSTEM

This section describes where the SCREEN, COLOR and CHARACTER memory components in the graphics system are located in character modes and bit map modes.

Screen and character memory are addressed and stored differently in the character modes than in the bit-map modes. The split-screen modes use a section of both the character screen storage and the bit map screen storage.

In graphics operations, the C128 can operate in either BASIC or machine language in both C128 and C64 modes.

This section tells you where the graphics locations and screen color character memory are stored under each graphic mode. The next section details the inner workings of each graphic display mode including how color and data are assigned and how screen, color and character memory are interpreted.

## SCREEN MEMORY (RAM)

The location in which screen RAM is stored in memory and the way the data are stored within it depends on the current graphics mode and operational mode of the C128.

### C128 BASIC

In Commodore 128 BASIC, the character screen memory is located in the default address range 1024 ($0400) through 2023 ($07E7). The text screen memory can be moved. Remember, certain addresses use indirect memory locations to change the value of the actual address. The shadow register for the pointer to the text screen memory is location 2604 ($0A2C). The actual location is 53272, but the screen editor uses a shadow since the VIC screen is interrupt-driven. A direct poke to 53272 ($D018) is changed back to its original value every sixtieth of a second. Here's how to change the location of screen memory in C128 BASIC:

**POKE, 2604 (PEEK(2604) AND 15) OR X**

where X is a value in Figure 8–6.

If you move the screen memory, make sure that the screen and character memory do not overlap. In addition, make sure to add an offset of $4000 to the start address of screen and character memory for each bank above 0. Additional commands are required to make the program work. Details follow in the discussion of each graphic mode, as well as program examples.

| | LOCATION* | | |
|---|---|---|---|
| X | BITS | DECIMAL | HEX |
| 0 | 0000XXXX | 0 | $0000 |
| 16 | 0001XXXX | 1024 | $0400 (DEFAULT) |
| 32 | 0010XXXX | 2048 | $0800 |
| 48 | 0011XXXX | 3072 | $0C00 |
| 64 | 0100XXXX | 4096 | $1000 |
| 80 | 0101XXXX | 5120 | $1400 |
| 96 | 0110XXXX | 6144 | $1800 |
| 112 | 0111XXXX | 7168 | $1C00 |
| 128 | 1000XXXX | 8192 | $2000 |
| 144 | 1001XXXX | 9216 | $2400 |
| 160 | 1010XXXX | 10240 | $2800 |
| 176 | 1011XXXX | 11264 | $2C00 |
| 192 | 1100XXXX | 12288 | $3000 |
| 208 | 1101XXXX | 13312 | $3400 |
| 224 | 1110XXXX | 14336 | $3800 |
| 240 | 1111XXXX | 15360 | $3C00 |

*Remember that the BANK ADDRESS offset of $4000 per video bank must be added if changing to a higher video bank above 0.

**Figure 8–6. Screen Memory Locations**

This register also controls where character memory is placed in memory. The upper four bits control the screen, the lower four control character memory. The "AND 15" in the POKE 2604 statement ensures that the lower nybble is not upset. (If it had been, you would not see the correct character data.)

In Commodore 128 bit map mode (standard or multi-color), the default bit map screen memory (video matrix) is located between 7168 ($1C00) and 8167 ($1FFF). Screen memory is interpreted differently in bit map mode than in text mode. The video matrix in bit map mode actually supplies color information to the bit map. This is explained in detail in the Standard Bit Map Mode section elsewhere in this chapter. To change the location of the bit map screen memory (video matrix), use this command:

**POKE 2605, (PEEK(2605) AND 15) OR X**

where X is a value in Figure 8–6. Location 2605 is also a shadow register for 53272, but only for bit map mode. When you move the video matrix you must ensure that it does not overlap the bit map (data). In addition, be sure to add an offset of $4000 to the start address of the video matrix and the bit map for each video bank above zero.

## C64 BASIC

In C64 mode, the text screen defaults to locations 1024 ($0400) through 2023 ($07E7). In bit map mode, the video matrix (screen memory) also defaults to this range though the screen memory is interpreted differently in either mode. Commodore 64 BASIC allows you to move the location of the video matrix to any one of the sixteen locations specified in Figure 8–6. The upper four bits of location 53272 ($D018) control the location of the screen memory. To change the location of screen memory, use the following command:

**POKE 53272, (PEEK(53272) AND 15) OR X**

where X is equal to one of the values in Figure 8–6

> **NOTE:** The following paragraph pertains to both C128 and C64 modes.

Bits zero and one of location 56576 ($DD00) control which of the four video banks is selected. The default bank is 0. If you change to another video bank (from 0 to 1, for example), then for each bank higher than bank zero, you must add an offset of $4000 to the starting video matrix (screen memory) address in Figure 8–6. This yields the actual address of the video matrix. For example, if you're changing from bank 0 to bank 1, add $4000. If you are going to bank 2, add $8000; if you are changing to bank 3, add $C000. Remember, this is true for both C128 and C64 modes.

## MACHINE LANGUAGE

In machine language, use the commands listed under *A* in Figure 8–7 to move the C128 (VIC) text screen. Use the commands under *B* to move the C128 bit map screen memory (video matrix). Use the commands under *C* to move the C64 text or bit map screen memory (video matrix).

| (A) | (B) | (C) |
|---|---|---|
| | | MOVE C64 |
| MOVE C128 | MOVE C128 | TEXT OR BIT MAP |
| TEXT SCREEN | BIT MAP SCREEN | SCREEN MEMORY |
| LDA $0A2C | LDA $0A2D | LDA $D018 |
| AND #$0F | AND #$0F | AND #$0F |
| ORA#$X | ORA #$X | ORA #$X |
| STA $0A2C | STA $0A2D | STA $D018 |

**Figure 8–7. Moving Screen Memory in Machine Language**

In Figure 8–7, X is the hexadecimal equivalent of the decimal value X in the left column in Figure 8–6. The second and third instructions in each example in Figure 8–7 make sure not to upset the lower four bits of location 53272 or its shadow registers, 2604 ($0A2C) and 2605 ($0A2D), since they control the character data for text and bit map modes.

# COLOR RAM

## C128 BASIC

Color RAM within the Commodore 128 is always stationary in memory. It occupies the address range 55296 ($D800) through 56295 ($DBE7). In standard character mode, screen RAM and color RAM correspond to one another on a one-to-one basis. Location 1024 gets color data from 55296, 1025 gets color from 55297 and so on. Multi-color character mode utilizes color RAM also, but in a different manner. Additional explanations and examples are provided in the Standard Character Mode section of this chapter.

## COLOR RAM BANKING

In C128 mode, the LORAM and HIRAM signal lines allow the graphics system to make use of an additional Color RAM bank, which is not available in C64 mode. This allows fast and clean switching of colors for the character or multi-color bit map screen. The LORAM signal line allows the 8502 microprocessor to access one color RAM bank, while the HIRAM control line allows the VIC chip to access either Color RAM bank independently of the microprocessor. Bit 0 of location 1 controls the LORAM signal line. LORAM selects color RAM bank 0 or 1 as seen by the 8502 microprocessor depending on the value of the bit. If the bit value is low, the color RAM bank 0 is accessed by the 8502. If the value of the bit is high, the upper color RAM bank is accessed by the 8502 microprocessor.

Bit 1 of location 1 controls the HIRAM signal line. HIRAM selects color RAM bank 0 or 1 as seen by the VIC chip, depending on the value of the bit. If the bit value is low, the color RAM bank 0 is accessed by the VIC chip. If the value of the bit is high, the upper color RAM bank 1 is accessed by the VIC chip.

These control lines add flexibility to the already powerful C128 graphics system. This allows you to change colors of the multi-color bit map or character screen on the fly, without any time delay. It allows you to swap color RAM banks instantly.

In standard bit map mode, color information is obtained from the bit map screen memory (the video matrix, $1C00 through $1FFF), not color memory. Bit map mode interprets screen memory differently than character mode. Color RAM is used in standard character mode, multi-color bit map mode, multi-color character mode and the split-screen mode.

## C64 BASIC

In standard character mode, color RAM is located in the same place as in C128 mode: 55296 ($D800) through 56295 ($DBE7).

In bit map mode, C64 BASIC receives color information from screen memory (the

video matrix) as does C128 mode, though the default location for screen memory is 1024 ($0400) through 2023 ($07E7).

## MACHINE LANGUAGE

In machine language or in BASIC, standard character mode color data always comes from the same place. Color RAM is used for multi-color character mode. In standard bit map mode, however, color data originates from screen memory, so wherever you place screen memory, the color data for the bit map comes from the specified screen memory (video matrix) range. Multi-color bit map mode receives color from three places: color RAM, screen memory and background color register 0. This is explained in depth in the sections on the multi-color character and multi-color bit map modes.

# CHARACTER MEMORY (ROM)

## C128 BASIC-CHARACTER MODES

In standard character mode, character information is stored in the character ROM in the memory range 53248 ($D000) through 57343 ($DFFF). In location 1 of the Commodore 128 memory map, the CHAREN (CHARacter ENable) signal determines whether the character set is available in any given video bank (0–3). Bit 2 of location 1 is the CHAREN bit. If the CHAREN bit is high (1), the Commodore character set is not available within the currently selected video bank in context. If the value of bit 2 in location 1 is low, equal to zero, then the C128 character set is available in the currently selected video bank. This is true in any of the four video banks in both 64K RAM banks. This feature allows the Commodore 128 character set to be available in any video bank at any time. To read the character ROM, enter BANK 14 either in BASIC or the MONITOR, and read the ROM, starting at location 53248. This configuration switches out I/O, and maps in character ROM in the range $D000 through $DFFF. Figure 8–8 shows how the character sets are stored in the character ROM:

| ADDRESS | | | VIC* | |
|---|---|---|---|---|
| BLOCK | DECIMAL | HEX | IMAGE | CONTENTS |
| 0 | 53248 | D000–D1FF | 1000–11FF | Upper case characters |
| | 53760 | D200–D3FF | 1200–13FF | Graphics characters |
| | 54272 | D400–D5FF | 1400–15FF | Reversed upper case characters |
| | 54784 | D600–D7FF | 1600–17FF | Reversed graphics characters |
| 1 | 55296 | D800–D9FF | 1800–19FF | Lower case characters |
| | 55808 | DA00–DBFF | 1A00–1BFF | Upper case & graphics characters |
| | 56320 | DC00–DDFF | 1C00–1DFF | Reversed lower case characters |
| | 56832 | DE00–DFFF | 1E00–1FFF | Reversed upper case & graphics characters |

* = in C64 mode only

**Figure 8–8. Breakdown of Character Set Storage in Character ROM**

The character memory is relocatable as is screen memory. To move standard character memory in C128 BASIC, alter the lower four bits (nybble) of location 2604 ($0A2C). Location 2604 is a shadow register for 53272 for the text screen memory (upper four bits) and character memory (lower four bits). To move the standard character memory use the following command:

**POKE 2604, (PEEK(2604) AND 240) OR Z.**

where Z is a value in Figure 8–9.

LOCATION OF CHARACTER MEMORY

| VALUE OF Z | BITS | DECIMAL | HEX | |
|---|---|---|---|---|
| 0 | XXXX000X | 0 | $0000–$07FF | |
| 2 | XXXX001X | 2048 | $0800–$0FFF | |
| 4 | XXXX010X | 4096 | $1000–$17FF | ROM IMAGE in BANK 0 & 2 (default)* |
| 6 | XXXX011X | 6144 | $1800–$1FF | ROM IMAGE in BANK 0 & 2* |
| 8 | XXXX100X | 8192 | $2000–$27FF | |
| 10 | XXXX101X | 10240 | $2800–$2FFF | |
| 12 | XXXX110X | 12288 | $3000–$37FF | |
| 14 | XXXX111X | 14336 | $3800–$3FFF | |

* = in C64 mode only.

**Figure 8–9. Character Memory Locations**

As with the other graphic system components, character data behaves differently in bit map mode than in text mode.

Remember, the upper nybble controls where the screen memory maps in, so make sure not to upset those bits. The AND 240 in the POKE statement above takes care of preserving the upper four bits.

In C128 mode the character sets are available in all video banks depending on the value of CHAR ENable

**NOTE:** Remember to add an offset of $4000 to the start address of character memory, for each bank above 0; i.e., for bank 3 add 3*$4000 = $C000

## C128 BASIC-BIT MAP MODES

In bit map mode, the character memory data, also referred to as the bit map, defaults to the range 8192 ($2000) to 16191 ($3F3F). The bit patterns of these 8000 bytes tell the computer which pixels to turn on. This block of memory "maps out" the picture on the screen, according to the data in this 8000-byte block. Since the standard bit map screen is 320 × 200, 64000 pixels make up the screen image. Divide 64000 by 8 to arrive at 8000 bytes of memory for the bit map.

Besides the upper four bits for the video matrix, bit 3 of location 2605 ($0A2D) is the only significant bit in bit map mode. Location 2605 is the indirect memory location of 53272 for bit map mode only.

When you issue the **GRAPHIC 1** command, bit 3 in 2605 is set. This specifies the bit map (data) to start at location 8192 ($2000). Whenever you enter bit map mode with the GRAPHIC command, bit 3 is always set. Outside of BASIC, you can specify the bit map to start at location $0000; therefore, the value of bit 3 is zero.

If the C128 is running under the control of C128 BASIC, the bit map always starts on a boundary of $2000 (since bit 3 is set) within a given video bank. In video bank zero, the bit map starts at $2000. For banks 1, 2 and 3, the bit map begins at $6000, $A000 and $E000, respectively, since you must add an offset of $4000 for each bank number above zero. In machine language, however, bit 3 may have a value of zero or one. Therefore the bit map may start at $0000 if bit 3 is zero, or at $2000 if bit 3 is one, in each video bank. This means that the bit map has eight possible starting locations (per 64K RAM bank) in machine language—two for each of the four video banks. In C128 BASIC, the bit map can only start at one of the four locations.

Don't forget to add the mandatory $4000 offset for each video bank above 0. The eight possible starting locations of a bit map in a machine language program are shown in Figure 8–10.

This gives you the choice of eight starting locations in which to place your bit map data. Though only one bit map can fit in each video bank, you must leave room for the video matrix. Since the C128 has two RAM banks—each with four video banks per RAM bank—you may have a total of eight bit maps in memory, one for each video bank. Each video bank can fit only one bit map, because you need 1K for screen RAM, and 8K for the bit map since a video bank has a maximum of 16K. To access another bit map, you must switch video banks. To access a bit map in the upper RAM bank (1), you may have to switch RAM banks and video banks.

| VIDEO BANK | VALUE OF BIT 3 = | |
|---|---|---|
| | 0 | 1 |
| 0 | $0000 | $2000 |
| 1 | $4000 | $6000 |
| 2 | $8000 | $A000 |
| 3 | $C000 | $E000 |

**Figure 8-10. Starting Locations for Bit Map in Machine Language**

## C64 BASIC-CHARACTER MODES

In standard character mode in C64 BASIC, the lower four bits of location 53272 control where character memory is placed. As in C128 mode, the character ROM is actually mapped into memory between 53248 ($D000) and 57343 ($DFFF). The ROM image appears in RAM in the range 4096–8191 (in video bank 0) and 36864–40959 in bank 2, since it must be accessible to the VIC chip in a 16K range in video banks 0 and 2. The character sets are not accessible in video banks 1 and 3. This ROM imaging in RAM applies only to character data as seen by the VIC chip. These memory ranges are still usable for data and programs and have no effect on the contents of RAM as far as your programs are concerned.

In C64 mode, the ROM image overlays the RAM underneath. A write operation "bleeds through" to the RAM underneath, while a read returns a ROM value depending on which memory configuration is currently in context. Since the VIC chip accesses 16K at a time, the character set images must appear in the 16K which the VIC chip is currently addressing in video banks 0 and 2. Remember, in C128 mode, the character sets are available in all video banks according to the value of the CHAREN bit in location 1.

You can change the location of character memory with the following command:

**POKE 53272, (PEEK(53272) AND 240) OR Z**

where Z is a decimal value in the table in Figure 8–9.

The breakdown of the character sets is the same as in the C128 for the character ROM (see Figure 8–8).

## C64 BASIC BIT MAP MODE

In bit map mode, bit 3 of location 53272 specifies the start of the bit map either at $0000 or $2000 depending whether the value of bit 3 is 0 or 1, respectively. Use the following command:

**POKE 53272 (PEEK (53272) AND 240) OR Z**

where Z is zero if you want the bit map to start at $0000 in each bank, or Z = 8 if you want to place the bit map starting at 8192 ($2000) in each video bank.

See Figure 8–10 for the arrangement of the bit map in each of the four 16K video banks within the two RAM banks. If you switch video banks, don't forget to add the $4000 (hex) offset for each bank above 0. See the Character Memory section under C128 Bit Map Mode in the last section for more detail on the arrangement of bit maps in memory.

## MACHINE LANGUAGE

There are three ways to select the placement of character memory, as shown in Figure 8–11. Example A places character memory using the shadow register $0A2C in place of the actual $D018 register. Example B specifies the start of the bit map at $2000 (using shadow register $0A2D). Example C specifies the start of the C64 bit map or character memory.

```
A              B              C
LDA $02AC      LDA $02AD      LDA $D018
AND #$F0       AND #$F0       AND #$F0
ORA #$Z        ORA #$08       ORA #$Z
STA $02AC      STA $02AD      STA $D018
```

**Figure 8–11. Selected Character Memory Location**

In Figure 8–11, Z is a value in the table in Figure 8–9.

# STANDARD CHARACTER MODE

## HOW TO ENTER STANDARD CHARACTER MODE

The C128 powers up in standard character mode. This mode displays characters on the default screen. The character is displayed in a single color on a single color background. This is the mode in which you write (enter) programs. When you press RUN/STOP and RESTORE, the C128 defaults to the text screen.

Location 53265 (and its shadow register $00D8) determine whether the C128 is operating in standard character mode. If bit 5 is 0, as it is on power-up, the C128 is in character mode; otherwise it is in bit map mode.

Location 53270 (and its shadow register $00D8) determine whether the characters are standard (single color) or multi-color. Bit 4 of 53270, and the shadow bit, bit 7 of $00D8, specify multi-color mode. If these bits are equal to zero, characters are standard; otherwise they are multi-color. See the Multi-color Character Mode section for more details on selecting multi-color character mode.

## SCREEN LOCATION

In standard character mode, the screen memory defaults to the range 1024 ($0400) through 2023 ($07E7). This is relocatable. See the Screen Memory section in the preceding pages.

Since the screen is 40 columns by 25 lines, the text screen requires 1,000 memory locations to store all of the screen information in memory. The final twenty-four memory locations in screen memory do not store displayed data; they are used for other purposes. Each column of every row you see on the screen has its own screen memory location. The top-left screen location, referred to as HOME, is stored at address 1024 ($0400). The second screen location marked by the cursor is 1025 ($0401), and so on. Although the screen you see is constructed in rows and columns, the screen memory within the computer is stored linearly, starting at 1024 ($0400) and ending at location 2023 ($07E7).

Figure 8–12 shows a screen memory map, so you can visualize how a screen memory location corresponds to the location on the physical screen of your video monitor.

**SCREEN MEMORY MAP**

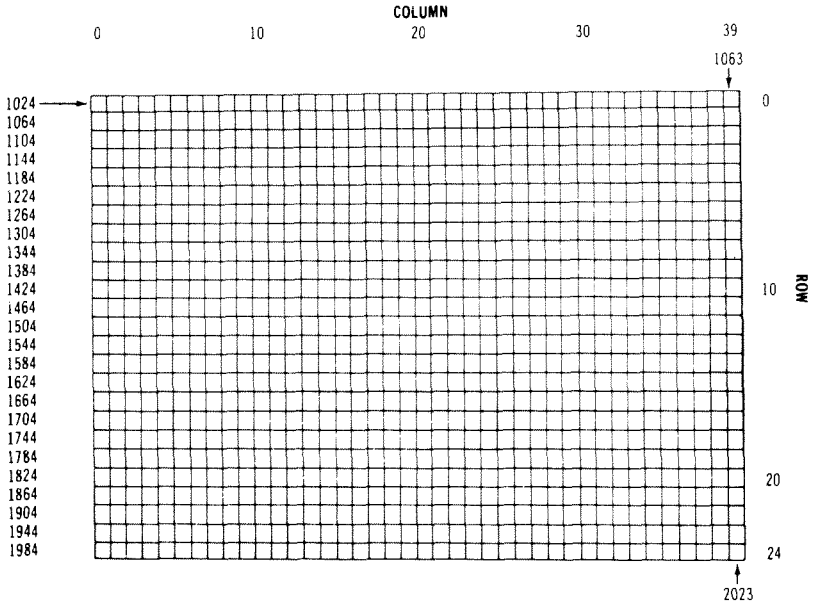



**Figure 8–12. Screen Memory Map**

## HOW THE SCREEN MEMORY DATA IS INTERPRETED

This screen memory range stores whole characters only. The characters are not represented as ASCII character string codes (**CHR$**). Instead, they are stored in memory as screen codes as shown in Appendix D. The screen codes and character string codes are

different due to the way they are stored in the character ROM. Notice in Appendix D that the screen code for an at-sign (@) is 0. The @ is numbered 0 because it is the first character to be stored in the character ROM. The letter "A" is the second character ROM; therefore its code is 1. The letter "B" is the third character in the character ROM, etc. The screen code is actually an index from the starting location of the character ROM, beginning with zero.

If you want to POKE a character directly into screen memory, use the screen code rather than the ASCII character string (CHR$) code. The same holds true for the machine language monitor. For example:

**POKE 1024,1**

places the letter A in the HOME position on the VIC screen. From the monitor, placing the value 1 in location $0400 (decimal 1024) also displays the letter A in the HOME position on the VIC screen.

## COLOR DATA

In standard character mode, color information comes from color RAM, in the address range 55296 ($D800) through 56295 ($DBE7). This memory determines the color of the characters in each of the 1,000 screen locations. The background color of the screen is determined by the background color register 0 which is location 53281.

The color RAM and the screen RAM locations correspond on a one-to-one basis. Screen location 1024 pertains to color RAM location 55296; screen location 1025 corresponds to color location 55297, etc. Figure 8–13 is the color RAM memory map. The map shows how color RAM corresponds to the locations in screen RAM and the placement on your video display.

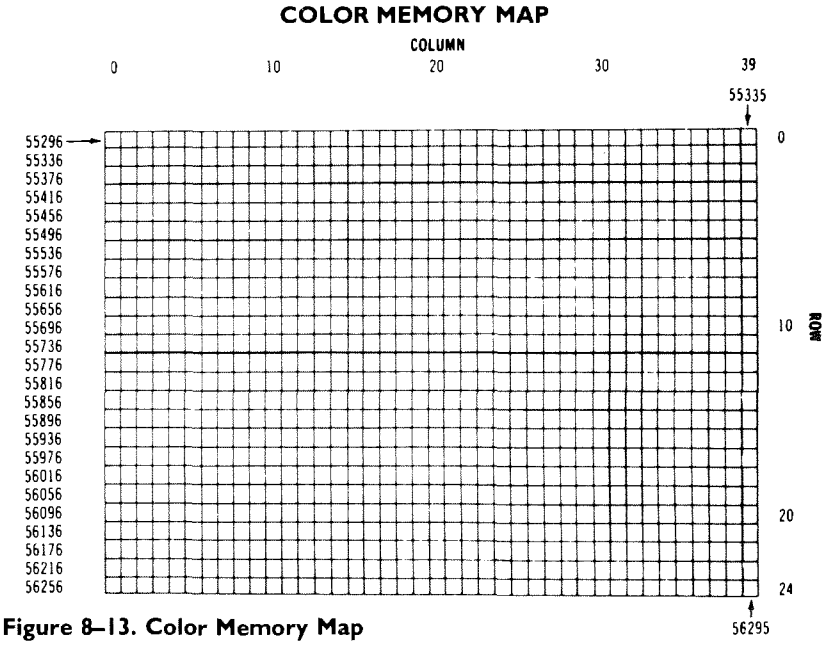



**Figure 8–13. Color Memory Map**

## HOW COLOR MEMORY IS INTERPRETED

The contents of the color RAM locations contains the color codes 0–15. Each color memory location may have a different color code. The lower four bits (nybble) of COLOR RAM are significant. Figure 8–14 shows the COLOR RAM color codes:

| | |
|---|---|
| **0 Black** | **8 Orange** |
| **1 White** | **9 Brown** |
| **2 Red** | **10 Light Red** |
| **3 Cyan** | **11 Dark Gray** |
| **4 Purple** | **12 Medium Gray** |
| **5 Green** | **13 Light Green** |
| **6 Blue** | **14 Light Blue** |
| **7 Yellow** | **15 Light Gray** |

**Figure 8–14. Color Codes—40 Columns**

Notice these color code values are one less than the color codes used by the keyboard and BASIC. If you want to store a value directly into COLOR RAM, store the values in the table above, not the color codes used by BASIC and the keyboard. For example:

**POKE 55296,1**

colors the character in the HOME position white. From the monitor, place the value 1 in location $D800, and the same results occur.

Remember, these color codes only control the color of the foreground character. The background color is controlled by background color register 0 (53281). The pixels that make up the character image are enabled by bits in character memory. If the bit is enabled, the pixel in the foreground is turned on in the foreground color, and is therefore controlled by color RAM. If the bits making up the character are turned off, they default to the color in background color register 0. The combination of on and off bits makes up the image of the character. The value of these bits determines whether the color data comes from color RAM or background color register 0. You'll learn more about character patterns in the next few paragraphs.

## CHARACTER MEMORY

In standard character mode, the C128 receives character data from the CHARACTER ROM. The character ROM is stored in the range 53248 ($D000) through 57343 ($DFFF). Since the VIC chip is capable of accessing 16K at a time, the C128 needs a way to have the character ROM available in the 16K VIC range. In C128 mode, the character ROM is available in any VIC bank in C128 mode, based on the value of

CHAREN. See the chapter set availability in the Character Memory section in the beginning of this chapter.

In C64 mode the character ROM is available only in banks 0 and 2. This is accomplished by having a ROM IMAGE of the character ROM (53248–57343) mapped into memory in place of RAM, in the range 4096–8191 ($1000–$1FFF) in video BANK 0, and 36864–40959 ($9000–$9FFF) in video BANK 2. In banks I and 3, the character ROM is not available to the VIC chip.

Notice that the range where the character ROM is actually stored (53248–57343) is also occupied by the I/O registers but not at the same time. When the VIC chip accesses the character ROM, the character ROM is switched into the currently selected video bank as a ROM image (in C64 mode only). When the character ROM is not needed, the I/O registers are available in the usual range. It is important to note the ROM image applies only to the character data as seen by the VIC chip. The RAM locations where the ROM image maps in are still usable for programs and data. The locations where the VIC chip looks for the character data are relocatable. See the Character Memory section elsewhere in this chapter for information on moving character memory.

## HOW TO INTERPRET CHARACTER MEMORY IN STANDARD CHARACTER MODE

Typically, a complete character set contains 256 characters. The C128 contains two sets of characters, for a total of 2 times 256 or 512 characters. In 40-column (VIC) output, only one character set is available at a time. Upon power-up, the uppercase/graphics character set is available through the keyboard. To access the second character set, press the Commodore key ( C= ) and shift key at the same time. The second character set is composed of the upper- and lowercase/graphics characters.

In character ROM, each character requires eight bytes of storage to make up the character pattern. Since 256*8 is equal to 2048 bytes or 2K, and since there are two character sets, the C128 has a total of 4K of character ROM. Figure 8–15 shows where each character set is stored in the character ROM.

| ADDRESS | | | VIC-II | |
|---|---|---|---|---|
| BLOCK | DECIMAL | HEX | IMAGE | CONTENTS |
| 0 | 53248 | D000–D1FF | 1000–11FF | Upper case characters |
| | 53760 | D200–D3FF | 1200–13FF | Graphics characters |
| | 54272 | D400–D5FF | 1400–15FF | Reversed upper case characters |
| | 54784 | D600–D7FF | 1600–17FF | Reversed graphics characters |
| 1 | 55296 | D800–D9FF | 1800–19FF | Lower case characters |
| | 55808 | DA00–DBFF | 1A00–1BFF | Upper case & graphics characters |
| | 56320 | DC00–DDFF | 1C00–1DFF | Reversed lower case characters |
| | 56832 | DE00–DFFF | 1E00–1FFF | Reversed upper case & graphics characters |

**Figure 8–15. Location of Character Sets in Character ROM**

Note that there is really *8K* of character ROM—4K for C64 mode and 4K for C128 mode. The system automatically selects the appropriate character ROM for each mode of operation.

The bit patterns stored in the character ROM have a direct relationship to the pixels on the screen, where the character is displayed. In memory, each character requires eight bytes of storage. On the screen, a character is made up of an 8 by 8 pixel matrix. Think of a character as eight rows of eight pixels each. Each row of pixels requires one byte of memory, so each pixel requires one bit.

Since a character is an 8 by 8 pixel matrix, each character requires a total of 64 bits or eight bytes. Within each byte, if a bit is equal to 1, the corresponding pixel in that character position is turned on. If a bit in a character ROM byte is equal to 0, the corresponding pixel within the character on that screen position is turned off. The combination of on and off pixels creates the image of the characters on the screen. Figure 8–16 demonstrates the correspondence between a character on the screen and the way it is represented in the character ROM.

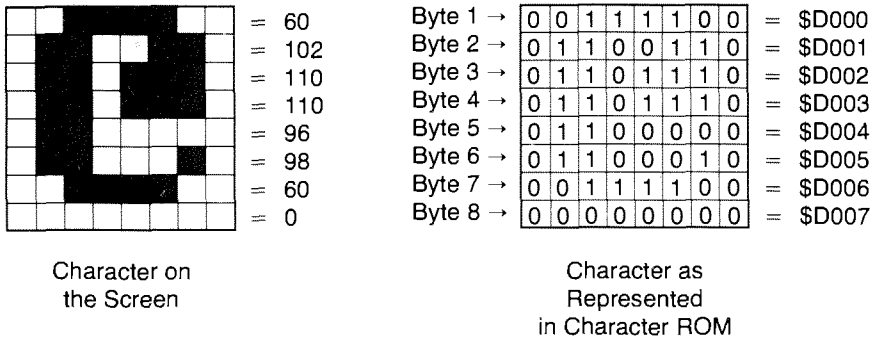



**Figure 8–16. Relationship of Screen Character to Character ROM.**

In Figure 8–16, the first eight bytes of character ROM, ($D000–$D007) are equal to 60,102,110,110,96,98,60 and 0. These decimal numbers are calculated from the binary value of the eight bytes that pertain to each row of pixels in the character. For each bit that is equal to one, raise two to the bit position (0–7). For example, the first byte of character ROM ($D000) is equal to 60, which is calculated by raising two to the following bit positions:

$$2^2 + 2^3 + 2^4 + 2^5 = 4+8+16+32=60$$

The bits that are set (on) correspond to pixels that are enabled on the screen in the foreground color. Bits that are clear correspond to pixels that are disabled, which are displayed in the background color, according to background color register 0 at location 53281.

The second byte (row of pixels) of the at-sign (@) character is equal to 102 (decimal) and is obtained by the following:

$$2^1 + 2^2 + 2^5 + 2^6 = 102$$

The last byte of the at-sign character is equal to zero, since no bits are set. Therefore, each pixel on the screen is displayed in the background color. The values of the binary digits on the right in Figure 8–16 are directly related to the image of the character as it appears on the screen on the left in Figure 8–16.

# ACCESSING CHARACTER ROM

## C128 BASIC

To access character ROM in C128 BASIC, type and run the following program:

```
10 BANK 14
20 FOR I=53248  TO 53248+7:PRINTPEEK(I);:NEXT
30 BANK 15
```

Enter Bank 14, the only BASIC bank where the character ROM is accessible. Then print the PEEK value of the first eight bytes of the character ROM. When finished, return to Bank 15.

# MACHINE LANGUAGE

To access character ROM in C128 Machine Language, type and run the following program:

```
MONITOR
    PC  SR AC XR YR SP
; FB000 00 00 00 00 F6

. 01800  A9 01    LDA #$01
. 01802  8D 00 FF STA $FF00
. 01805  A2 00    LDX #$00
. 01807  BD 00 D0 LDA $D000,X
. 0180A  9D 40 18 STA $1840,X
. 0180D  E8       INX
. 0180E  E0 07    CPX #$07
. 01810  D0 F5    BNE $1807
. 01812  A9 00    LDA #$00
. 01814  8D 00 FF STA $FF00
. 01817  60       RTS


10 SYS 6144
20 FOR I=6208 TO 6208 +7:PRINTPEEK(I);:NEXT
```

These machine language and BASIC routines accomplish the same task as the preceding four-line BASIC program. The first two machine language instructions switch in the character ROM, and switch out I/O. The next six instructions transfer the first eight

bytes of character ROM into locations 6208 ($1840) through 6215 ($1847). The last three instructions switch out the character ROM, replace it with the I/O registers and return from the machine language subroutine to BASIC.

The BASIC routine activates the machine language subroutine, then prints the values that were temporarily stored in 6208 through 6215. See Chapter 6, How to Enter Machine Language Programs, for details on how to input machine language instructions on the C128.

### C64 BASIC

To access character ROM in C64 BASIC, enter and run the following program:

```
40 POKE 56334,PEEK(56334)AND 254
50 POKE 1,PEEK(1) AND 251
80 FOR I=0 TO 7:POKE 6144+I, PEEK(53248+I):NEXT
90 POKE 1,PEEK(1) OR 4
105 POKE 56334,PEEK(56334) OR 1
130 FOR I=6144 TO 6144+7:PRINTPEEK(I);:NEXT
```

Line 40 turns off the interrupt timer. Line 50 switches out I/O and replaces it with character ROM. Line 80 transfers the first eight bytes of character ROM (53248–53255) to 6144–6151. Line 90 switches out character ROM, and replaces it with the I/O registers. Line 105 turns on the interrupt timer. Line 130 prints the first eight character ROM values that were temporarily stored in 6144 through 6151.

You may need to transfer parts of the character ROM data into RAM if you are creating your own character set, and you want the remainder to be from the C128 character set. This is covered in more detail later in the chapter. These methods of looking at the character ROM demonstrate how the character ROM is accessed, what the patterns of the characters look like and why you would want to access the character ROM.

The next section explains how to program your own custom characters in C128 mode.

## PROGRAMMABLE CHARACTERS

The Commodore 128 has a feature that allows you to redefine the character set into custom characters of your own. In most cases, you'll want to redefine only a few characters at most, while obtaining the rest of the character set from the Commodore 128 character ROM.

With programmable characters, you tell the C128 to get character information from RAM. Usually, characters are taken from the character ROM. If you only want certain characters, you can choose the ones you want, copy the character patterns into RAM and leave the rest in ROM. You cannot write to the character data in ROM; however, the character data placed in RAM can be redefined.

The first step in programming your own characters is to define the image. In the

Standard Character Mode section, you saw how a character on the screen is stored in the character ROM. Each character requires eight bytes of storage. Each byte corresponds to a row of pixels on the visible screen within the 8 by 8 character matrix; therefore, eight rows of pixels make up one character.

This section shows how to customize an uppercase cursive (script) character set for the letters A through H. Figure 8–17 shows the design for the uppercase cursive letter A. The grid in the figure demonstrates how the character appears on the screen within the 8 by 8 pixel matrix. Each row of the grid determines which bits are on within the character bit pattern, and, hence, which corresponding pixels are enabled on the screen. The eight-bit binary strings to the right of the grid are the bit patterns as stored in RAM. The numbers to the right of the binary strings are the decimal equivalents of the binary bit patterns. This decimal value is the data you POKE into RAM in order to display the character.

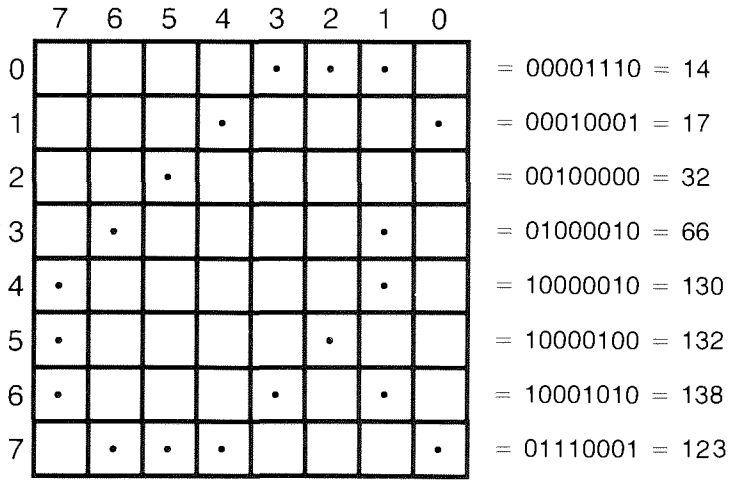



**Figure 8–17. Design for Cursive letter "A"**

The following program creates and displays the upper-case cursive characters A through H. Enter it into the computer and RUN it. You'll see the letters A through H change from uppercase block letters to uppercase cursive letters. When you press the newly defined lettered keys, they are displayed in cursive form.

Line 10 selects the uppercase character set, the set being redefined. Line 20 protects the character set from being overwritten by the BASIC program and prepares a location in RAM in which to place your character set. The end of user BASIC text and the top of string storage is moved from 65280 to 12288 (decimal), which substantially cuts down the size of BASIC programming space. The character set will be placed beginning at location 12288, but it does not have to be located there. The character set does have to be within the first 16K of memory unless another bank is selected. The VIC chip can only access 16K at a time so each video bank consists of 16K of memory.

```
10 PRINT CHR$(142) :REM SELECT UPPER CASE
20 POKE54,48:POKE58,48:CLR:REM PROTECT CHAR SET
30 BANK 14 :REM SWITCH TO BANK 14 FOR CHARACTER ROM
40 FORI=1 TO 511:POKEI+12288,PEEK(I+53248):NEXT:REM ROM TO RAM TRANSFER
50 BANK 15:REM SWITCH TO DEFAULT BANK
60 POKE 2604,(PEEK(2604) AND 240)+12:REM START CHAR BASE AT 12288
70 FORJ=12288TO12288+71 :REM PLACE CHARACTER DATA IN RAM
80 READ A
90 POKEJ,A
100 NEXT J
110 SCNCLR
120 DATA 0,0,0,0,0,0,0,0:REM @
130 DATA 14,17,32,66,130,132,138,123:REM A
140 DATA 124,66,66,124,66,81,225,126:REM B
150 DATA 62,67,130,128,128,128,131,126:REM C
160 DATA 125,98,125,65,65,193,161,254:REM D
170 DATA 225,66,64,56,120,65,66,124:REM E
180 DATA 127,129,2,4,14,228,68,56:REM F
190 DATA 193,163,253,33,249,65,99,190:REM G
200 DATA 227,165,36,36,126,36,37,231:REM H
```

You can leave yourself more BASIC text area, but to do this you must enter a video bank higher than zero. This sample program operates in Bank 0. If you place your character set in a higher video bank, remember to add an offset of 16384 ($4000) to the start of RAM character memory for each bank above video Bank 0.

The CLR in line 20 clears out memory starting at 12288 because, prior to the placement of the character set there, the memory locations are filled with random bytes. The random bytes must be cleared before new character information can be stored.

Line 30 selects BANK configuration 14. This configuration makes the character ROM visible with a PEEK command or within the machine language monitor, and temporarily switches out the I/O registers. Both the I/O functions and the 4K character ROM share the same locations ($D000–$DFFF). Depending on whether bit 0 in the configuration register (location $FF00) is on or off, the C128 addresses the I/O registers or the character ROM. Normally the C128 powers up with bit 1 turned off; therefore, the I/O registers in locations $D000–$DFFF are addressed. The BANK command in line 30 sets bit 0 in location $FF00; therefore, the character ROM in locations 53248–57343 ($D000–$DFFF) is addressed. The character ROM is accessed in order to make the transfer of characters to RAM.

Line 40 makes the actual transfer from ROM to RAM. Since the character ROM is accessed in line 30, it now begins at location 53248. The first 512 bytes (the uppercase character set) from the character ROM are POKEd into the 512 bytes of RAM beginning at location 12288 and ending at 12800. At this point the character set is ready to be redefined to custom characters.

Line 50 switches the I/O registers back in, meaning the character ROM is no longer available.

Line 60 specifies the start of the character set base at location 12288. The character set base can be stored in locations other than 12288. (See the Character Memory section earlier in this chapter for more information on moving character memory.) Location 2604 is the intermediate memory location that the interrupt-driven C128 screen editor uses to point to screen and character memory in character mode. You must use this indirect location to change the value of the actual register that points to the screen and character memory 53272. If you try to POKE directly to location 53272, the interrupt will

change the value back to the original one within a sixtieth of a second. (You can, however, disable the interrupt-driven screen editor. See the Shadow Register section for details.)

The value (AND 240) or 12 is placed in address 2604 to tell the C128 to point to character memory in RAM, starting at address 12288. If the value (AND 240) or 8 is placed into 2604, the character set will begin at 8192 and your BASIC program must be less than 6K, but the complete 4K of characters can be redefined. If the value (AND 240) or 14 is placed into that location, the character set starts at 14336. Your BASIC program then must be less than 2K, since the programmable character set must reside within a single 16K block, but the BASIC program that creates the characters can be almost 14K. If a number other than 12 is POKED into 2604, other program lines must be modified.

Lines 70 through 100 start a loop at the beginning of the character base (12288), read the values from the data statements which define the new characters (lines 120 and 200), and POKE them (line 90) into the locations allocated for the character set base, starting at location 12288. The value 12288+71 in line 70 sets aside seventy-two storage locations for the data values in lines 120 through 200 for storage in locations 12288 through 12359. When more data statements are added, the value (12288+71) must be increased to 12288 plus the number of data values in the data statements minus 1. For example, if more characters were defined and there were twenty data statements with eight values in each, then line 70 would read:

**70 FOR J=12288 TO 12288+(160–1)**

# MULTI-COLOR CHARACTER MODE

Standard character mode displays text in two colors: the foreground color of the character as determined by COLOR RAM, and the background color as determined by background color register 0 at location 53281 ($D021).

Multi-color character mode gives you the ability to display characters in four colors within an 8 by 8 character matrix. This substantially increases the freedom of using color. However, the horizontal resolution is only half the resolution of standard character mode (160*200), since multi-color mode bits and screen pixels are grouped in pairs. This means that the color definition and pixel density is twice as wide as standard character mode. The tradeoff in horizontal resolution is compensated for by the increased freedom of using color.

## HOW TO ENTER MULTI-COLOR MODE

Location 53270 and its shadow register ($00D8) determine whether the C128 is outputting standard or multi-color characters on the screen. Bit 4 of 53270 and bit 7 of 216 ($00D8) control multi-color mode for character and bit map modes. If bit 4 of 53270 (and bit 7 of 216) is equal to 1, multi-color mode is enabled. Otherwise, standard mode is enabled. Most of the new 7.0 BASIC graphics commands have provisions for multi-color mode. However, if you want to enter multi-color mode with a POKE command, type:

**10 POKE 216,255:** REM Disable IRQ Editor
**20 POKE 53270, PEEK (53270) or 16:** REM Select MCM

This enters multi-color mode, either for character mode or bit map mode.

## SCREEN LOCATION

The screen location in multi-color character mode defaults to 1024 ($0400) through 2023 ($07E7), the same as standard character mode. The screen memory locations can be relocated. See the Screen Memory section for details.

## HOW SCREEN DATA IS INTERPRETED

In multi-color character mode, the screen data from screen memory is interpreted as screen codes the same way as in standard character mode. The screen codes are listed in Appendix D. See the screen data interpretation in the standard character memory section. The only difference between standard character mode and multi-color character mode is the way color is assigned to the characters on the screen.

## CHARACTER MEMORY LOCATION

The character memory in multi-color character mode, as in standard character mode, is taken from between 53248 ($D000) and 57343 ($DFF) when I/O is switched out. See standard character mode for more detailed information on character memory.

## HOW CHARACTER MEMORY IS INTERPRETED IN MULTI-COLOR CHARACTER MODE

Character memory is interpreted virtually the same way in multi-color and in standard character modes, except for one difference: In standard character mode, if a bit in the character definition image is on, the pixel corresponding to that bit is colored in the foreground color as specified by color RAM. If a bit in the character image in the character ROM is equal to zero, the corresponding pixel on the screen is colored in the background color, as specified by background color register 0 (location 53281).

In multi-color character mode, color assignments to the pixels that make up the character on the screen are not in a direct one-to-one relationship to the bits in the character ROM data patterns. Instead, the bits that make up a character are grouped in pairs, as shown in figures 8–18, 8–19 and 8–20.

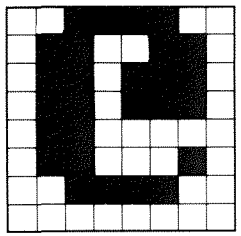


**Figure 8–18. "At" Sign (@) Character as It Appears on the Screen**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Figure 8–19. Bit Patterns of the "At" Sign (@) Character as They Appear in Character ROM**

| | | | |
|---|---|---|---|
| 00 | 11 | 11 | 00 |
| 01 | 10 | 01 | 10 |
| 01 | 10 | 11 | 10 |
| 01 | 10 | 11 | 10 |
| 01 | 10 | 00 | 00 |
| 01 | 10 | 00 | 10 |
| 00 | 11 | 11 | 00 |
| 00 | 00 | 00 | 00 |

**Figure 8–20. Bit Patterns of the "At" Sign (@) Character as They Are Grouped in Pairs in Multi-Color Character Mode**

The bits are grouped in pairs, since the horizontal resolution is only half as wide in multi-color mode. The bit pair determines the color assignments for the pixels within the character on the screen. The following section describes how the colors are assigned in multi-color character mode.

## COLOR DATA

The color of the pixels in a multi-color character originate from four sources, depending on the bit pairs. Since the bit pairs have four color possibilities, two bits are needed to represent four values: 00, 01, 10 and 11. In Figure 8–21, the value of the four bit pair combinations determines the color assignments for the pixels in a multi-color character.

| BIT PAIR | COLOR REGISTER | LOCATION |
|---|---|---|
| 00 | Background #0 color (screen color) | 53281 ($D021) |
| 01 | Background #1 color | 53282 ($D022) |
| 10 | Background #2 color | 53283 ($D023) |
| 11 | Color specified by the lower 3 bits in color memory | color RAM |

**Figure 8–21. Truth Table for Color Data**

If the bit pair equals 00 (binary), the color of those two pixels corresponding to the bit pair are colored by background color register 0 (location 53281 ($D021)). If the bit pair equals 01 (binary), the pixels are colored by background color register 1 (location 53282 ($D022)). If the bit pair equals 10 (binary), color for those two pixels within the character are colored from background color register 2 (location 53283 ($D023)). Finally, if the bit pair from the character pattern equals 11 (binary), those two pixels are colored from the color specified in the lower three bits (2, 1, 0) of color RAM. Color RAM is located between 55296 ($D800) and 56295 ($DBE7).

When multi-color character mode is selected, you can still display standard characters on some screen locations, and display others in multi-color mode. Bit 3 of each color RAM location determines whether the character is displayed in standard or multi-color mode. If bit 3 in color RAM is set (1), characters are displayed in multi-color mode. If bit 3 is clear (0), characters are displayed in standard character mode. This means that in order to display characters in multi-color mode, you must fill color RAM with a color code greater than 7. The colors greater than 7 (the ones that are displayed in multi-color mode) are shown in Figure 8–22.

| COLOR CODE | COLOR |
|---|---|
| **8** | **Orange** |
| **9** | **Brown** |
| **10** | **Light Red** |
| **11** | **Dark Gray** |
| **12** | **Medium Gray** |
| **13** | **Light Green** |
| **14** | **Light Blue** |
| **15** | **Light Gray** |

***Figure 8–22. Color Codes***

Remember, the multi-color bit (bit 3) must be set to display multi-color characters.

The following program illustrates multi-color character mode.

```
10 COLOR 0,1     :REM BKGND = BLACK
20 COLOR 2,1     :REM MULTCLR 1 = WHITE
30 COLOR 3,2     :REM MULTCLR 2 = RED
31 FOR I=1TO25
32 PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
33 NEXT
35 FOR I=55296+512 TO 55296+1023:POKE I,7:NEXT:REM PLACE YELLOW IN COLOR RAM
37 POKE 216, 255:REM DISABLE SCREEN EDITOR
40 POKE 53270,PEEK(53270) OR 16:REM SET MULTICOLOR BIT
50 FOR I=1TO25
60 PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
70 NEXT
85 FOR I=55296 TO 55296+1023:POKE I,14:NEXT:REM FILL COLOR RAM WITH BLUE
90 GRAPHIC 0:REM RETURN TO STANDARD COLOR
```

Lines 10, 20 and 30 place the color codes for black, white and red into background color registers 0, 1 and 2, respectively. Lines 31 through 33 print the letters of the

alphabet on the screen twenty-five times. Line 35 fills the last 512 bytes of color RAM with yellow. Line 37 disables the IRQ VIC screen editor. Line 40 enables multi-color mode. At this point, all the screen locations corresponding to the color RAM locations which have a color code greater than or equal to 8 are displayed in multi-color mode. Since the yellow color code is 7, all the color RAM locations having this code are placed in standard character mode. The default color for color RAM is code 13 (light green) for C128 mode, and code 14 (light blue) for C64 mode. Line 85 fills color RAM with the light blue color code. The multi-color characters displayed on the screen are red, white and blue on a black background.

# EXTENDED BACKGROUND COLOR MODE

The third type of character display mode, extended background color mode, allows you to display three colors at a time on the text screen. For example, you have the character color, the background color of the screen, and an additional background color within each 8 by 8 character matrix. This means you can display a white character with a green background in the 8 by 8 character matrix, on a black screen background. This mode offers the use of an additional color in an 8 by 8 character matrix, without any loss in screen resolution.

There is one sacrifice, however. In extended background color mode, only the first sixty-four characters of the screen code character set are available. The reason for this is that bits 6 and 7 determine which color will be selected for the background within the 8 by 8 pixel character matrix. This only leaves five bits for the computer to interpret which character is currently on the screen. The highest number you can represent with five bits is 63. This means only the screen code values between 0 and 63 are available for display on the screen within extended background color mode.

## HOW TO ENTER EXTENDED BACKGROUND COLOR MODE

Enabling bit 6 of location 53265 selects extended background color mode. Use this POKE in BASIC:

**POKE 53265, PEEK(53265) OR 64**

To turn it off, use this POKE:

**POKE 53265, PEEK(53265) AND 191**

## SCREEN LOCATION

The screen location in extended background color mode is the same as the standard character and multi-color character modes, 1024 ($0400) through 2023 ($07E7). This screen range can be relocated. See the SCREEN MEMORY section for details.

## HOW TO INTERPRET SCREEN DATA

The data in screen memory is interpreted as screen codes, which are actually the indexes into the character ROM. Instead of representing the data as ASCII characters, the screen codes represent the index into the character ROM which provide the ASCII codes. The first character in character ROM is the at sign (@); therefore the first screen code, 0, is the code for the at sign.

Remember, since extended background color mode only uses five bits to determine the screen code value, only the first 64 screen code characters (0-63) are available.

## COLOR DATA

The color assignments for the three colors on the screen stem from three sources. Just as in standard character mode, the foreground color is assigned by COLOR RAM, in the range 55296 ($D800) through 563295 ($DFE7). As described in the standard character mode section, each color RAM location has a direct one-to-one correspondence with the screen memory locations. See the Standard Character Mode section for screen and color memory maps and an explanation of how the two sections of memory correspond to one another.

The screen background color is assigned by background color register zero (location 53281 ($D021)). This is the color of the entire screen, on which the foreground and an additional 8 by 8 character matrix background is placed.

The additional 8 by 8 character matrix background colors are determined by the value of bits 6 and 7 of the screen code character value. Depending on the value of these bits, the extended background color (the color within the 8 by 8 character matrix for each character), comes from one of the four background color registers. Since there are four choices for the extended background color, the computer needs two bits to represent the four color choices. Figure 8–23 shows the four-bit combinations and the corresponding background color registers associated with them.

| CHARACTER CODE | | | BACKGROUND COLOR REGISTER | |
|---|---|---|---|---|
| RANGE | BIT 7 | BIT 6 | NUMBER | ADDRESS |
| **0–63** | **0** | **0** | **0** | **53281 ($D021)** |
| **64–127** | **0** | **1** | **1** | **53282 ($D022)** |
| **128–191** | **1** | **0** | **2** | **53283 ($D023)** |
| **192–255** | **1** | **1** | **3** | **53284 ($D024)** |

**Figure 8–23. Extended Background Color Registers**

For example, POKE the screen code for the letter A (1) into screen location 1024. Now POKE the screen value 65 into screen location 1025. You might expect the character to be a reverse A, the second character of the second screen code character set.

However, in this mode, you can only represent the first sixty-four characters, since you only have five bits to represent screen characters. By trying to represent the screen code 65, bit 6 is enabled, which tells the computer to select the background color register 1 (location 53282 ($D022)), and display the same character, but with the extended background color specified by background color register 1.

Here's a program that illustrates how extended background color mode operates:

```
5 SCNCLR
10 COLOR 0,1 :REM BKGRD=BLACK
20 COLOR 2,1 :REM MULTICOLOR 1
30 COLOR 3,2 :REM MULTICOLOR 2
40 POKE 53284,3:REM BACKGRD COLOR 3
45 POKE 53265,PEEK(53265) OR 64:REM SET EXTENDED BKGRND BIT
50 FOR I=1024 TO 1256:POKE I,I-1023:NEXT
60 PRINT:PRINT:PRINT:PRINT
```

In the program, line 5 clears the screen. Lines 10 through 40 assign colors to the four background color registers: black, white, red and cyan, respectively. Line 45 enables extended background color mode. Line 50 POKEs 232 characters into screen memory. Each time sixty-four characters are stored into the screen memory, the same set of sixty-four characters is POKEd into the next sixty-four screen locations. However, the next extended background color is displayed. First, the sixty-four characters are displayed with a black extended background, then a white extended background, then red, then cyan.

## CHARACTER DATA

Character data is interpreted the same way as in standard character mode, except only the first 64 characters of the screen character set are available. The character data is also located in the same range as in standard character mode. See standard character mode for more information on character data.

# STANDARD BIT MAP MODE

Standard Bit Map Mode, also referred to as high-resolution mode, offers the ability to display detailed graphic images in two colors. The resolution of bit map mode is 320 by 200 pixels. In this mode, the C128 no longer operates in terms of characters, which are 8 by 8 pixel images stored in those complete units at a time. Bit map mode allows you to address single pixels at a time, therefore exercising a substantial amount of control over the detail of images on your screen. The smallest unit addressed in the character display modes is an 8 by 8 pixel character. Standard bit map mode allows you to address every individual pixel of the possible 64000 pixels that make up an entire high-resolution screen image. Figure 8–24 shows how that bit-map coordinate plane is set up.

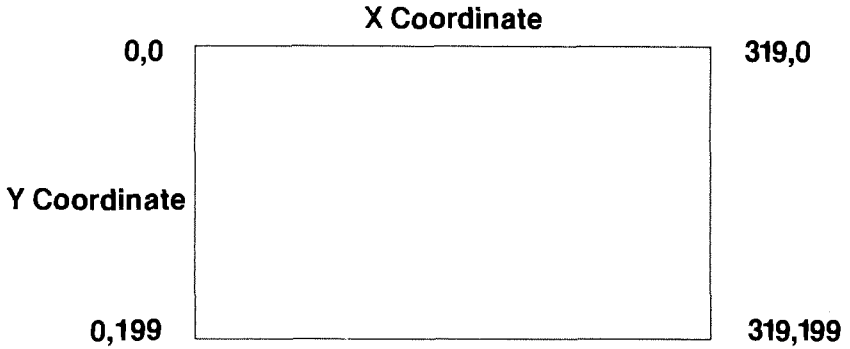



**Figure 8–24. Bit Map Screen Coordinates**

## HOW TO ENTER STANDARD BIT MAP MODE

To enter standard bit map mode, set bit 5 of location 216 ($00D8) (the shadow register of location 53265 ($D011)). When you issue the GRAPHIC 1,1 command in C128 BASIC, bit map mode is enabled and the bit map screen is cleared.

You can also use a POKE command as follows, but you should use the highest-level commands wherever possible:

```
POKE 216, PEEK (216) OR 32
```

This command turns on bit 5 of the GRAPHM register, which is the interface between the VIC chip and the interrupt-driven C128 screen editor. This indirectly turns on bit 5 in location 53265 and enters bit map mode.

You can disable the interrupt-driven screen editor and select bit map mode directly with these commands:

```
POKE 216, 255
POKE 53265, PEEK (53265) OR 32
```

In C128 mode machine language, use this program sequence:

```
LDA $00D8
ORA #$20
STA $00D8
```

In C64 mode machine language, try this:

```
LDA $D011
ORA #$20
STA $D011
```

## THE VIDEO MATRIX (SCREEN MEMORY) LOCATION

The default location of the C128 video matrix (i.e., the bit map screen memory) is 7168 ($1C00) through 8167 ($1FE7).

The default location of the C64 video matrix is 1024 ($0400) through 2023

($07E7). The video matrix can be moved, however. See the Screen Memory section elsewhere in this chapter for information on relocating the video matrix.

## HOW THE VIDEO MATRIX IS INTERPRETED

In bit map mode, the video matrix (bit map screen memory) is interpreted differently than it is in the character display modes. In the character display modes, the screen memory data is interpreted as screen codes corresponding to the characters in character ROM. However, in bit map mode, the video matrix data is interpreted as the supplier of color information for the bit map. The upper four bits (nybble) supply the color information for the bit map foreground, and the lower nybble supplies the color code for the bit map background.

The next section explains how pixels on the bit map screen are assigned to either the foreground color or the background color.

## BIT MAP DATA

In bit map mode, the character data, referred to as the bit map, is also interpreted differently than in the character display modes. The character data is not taken from character ROM at all. Instead, it is taken from an 8K section of RAM memory, known as the bit map.

The standard high-resolution screen is composed of 200 rows of 320 pixels, so that the entire screen is composed of 64,000 pixels. In bit map mode, one bit in memory is dedicated to an individual pixel on the screen. Therefore 64000 pixels require 64000 bits (or 8000 bytes) of memory to store the entire bit mapped image.

If a bit in memory in the 8000 byte bit map is set, the corresponding pixel on the screen is enabled, and becomes the color of the foreground as specified in the upper four bits of the video matrix. If a bit in the bit map is equal to zero, the corresponding pixel on the screen becomes the color of the background, as specified in the lower four bits of the video matrix. The combination of on and off bits in the bit map and the corresponding pixels on the screen define the highly detailed image on the video screen.

The bit map default location in memory ranges from 8192 ($2000) through 16191 ($3F3F). This requires 8000 bytes or just under 8K of memory. The spare bytes are used for other purposes. Location 53272 determines the location of the video matrix and bit map in memory. Since the screen editor is running on the IRQ, location 2605 ($0A2D) is an indirect address you must use to place a value in location 53272 ($D018). The upper four bits of 53272 determine where the video matrix begins and the lower four bits determine where the bit map begins. In bit map mode, only bit 3 is significant, so the bit map is either placed starting at location 0 or location 8192 ($2000) in each video bank.

If you change to a higher bank number, remember to add an offset of $4000 to the start of the bit map and the video matrix for each bank number above 0.

The video matrix is relocatable. See the Screen Memory section in the beginning of the chapter for details on how to move the video matrix.

The bit map tells the computer which pixels in the foreground to enable on the screen. Like a road map, it spells out exactly which pixels to turn on (in the foreground) and off (in the background color) in order to display a picture on the screen. For example, if the bit map started at location 8192 (the C128 BASIC default) the first byte of the bit map corresponds to the bit map pixel coordinates 0,0 through 0,7. The second byte of the bit map, location 8193, corresponds to coordinates 1,0 through 1,7 and so on. See Figure 8–25 to see how the bit map data in locations 8192–16191 correspond to the pixels on the visual screen:

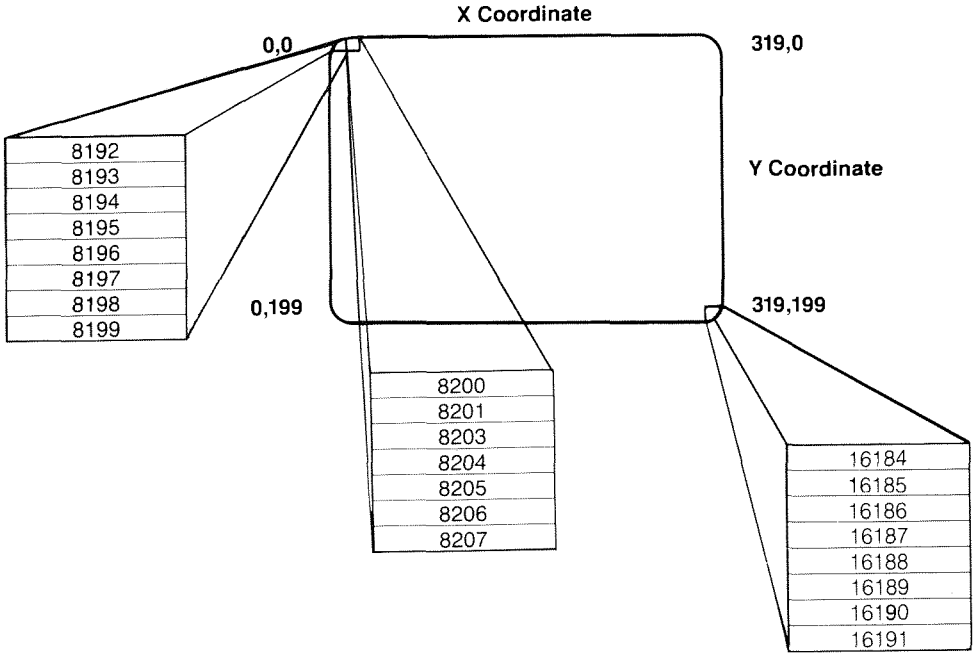



**Figure 8–25. Relationship of Bit Map Data to Screen Pixels**

Now you know how bit map mode operates internally within your C128. However, you need an easy way to turn on and off pixels in the bit map in order to display graphics on the screen. The new, high level BASIC 7.0 commands such as DRAW, CIRCLE, BOX and PAINT allow you to control the turning on and off of bits and their corresponding screen pixels. The use of the X and Y coordinates on the bit map coordinate plane easily orient you to displaying graphics. You can display high-resolution graphics in other ways, outside of the BASIC 7.0 graphics commands. This includes using commercial software packages that employ graphics tablets or joysticks to draw on the screen, writing your own draw routines using a joystick or paddle, or physically entering data into the bit map (which is painfully tedious and not recommended).

Another way to display graphics, which involves manipulating bits in the bit map, is through mathematical equations, using geometry. Several books are available which offer geometrical equations on how to draw three-dimensional objects and to move them. Refer to the Suggestions for Further Reading the back of the book for sources on graphics.

## COLOR RAM

In standard bit map mode, color RAM is not used since the color information for the bit map is taken from the upper and lower nybble of screen RAM. Color RAM is used, however, in multi-color bit map mode.

# MULTI-COLOR BIT MAP MODE

Multi-color bit map mode is a combination of standard bit map mode and multi-color character mode. Multi-color bit map mode allows the display of four colors within an 8 by 8 pixel bit map area. Like multi-color character mode, the horizontal resolution is only half of the standard bit map mode, though the tradeoff in resolution is compensated for by the use of two additional colors within an 8 by 8 pixel, bit mapped area.

## HOW TO ENTER MULTI-COLOR BIT MAP MODE

To enter multi-color bit map mode from C128 BASIC, issue the following command:

**GRAPHIC 3**

You can enter this mode with a POKE command as well. But make use of the highest level commands available for the easiest programming:

**POKE 216, PEEK (216) OR 160**

This POKEs the value 160 and turns on the multi-color mode bit 7 (value 128) and the bit map mode bit 5 (value 32) in the GRAPHM register which interfaces to the C128 interrupt driven screen editor. This indirectly turns on, respectively, bit 4 (multi-color mode) of location 53270 ($D016), and bit 5 (bit map mode) of location 53265 ($D011).

Bit 5 of location 53265 determines whether the C128 is in bit map mode or character mode. If bit 5 is equal to 1, bit map mode is enabled. Bit 4 of location 53270 determines whether the C128 is in standard or multi-color mode. If bit 4 is set, the C128 operates in multi-color mode, regardless of whether it is in character or bit map mode.

In C64 mode, you can store a value directly to these registers. But in C128 mode, the GRAPHM intermediate register must be used as a gateway to these actual registers. Again, this is because the C128 screen editor is interrupt driven, enabling the split-screen modes for text and simultaneous bit map displays. Since the screen editor is interrupt driven, an indirect register is used to restore the values that you need to use for

specific VIC registers. GRAPHM is one such register; therefore, every sixtieth of a second, the value in the GRAPHM register is loaded into the appropriate VIC registers.

## MACHINE LANGUAGE

To select multi-color bit map mode in C128 machine language, perform the following instructions:

**LDA $A0;** enables bits 4 and 5 of GRAPHM, the shadow location
**STA $00D8**

In C64 machine language, enter:

**LDA $D011**
**ORA #$20;** select Bit map mode
**STA $D011**
**LDA $D016**
**ORA #$10;** select multi-color mode
**STA $D016**

In both cases, you must clear the screen, color RAM and the bit map in your own program.

## VIDEO MATRIX LOCATIONS

The video matrix defaults to locations 7168 ($1C00) through 8167 ($1FE7) in C128 mode.

In C64 mode, the video matrix defaults to locations 1024 ($0400) through 2023 ($07E7). This is relocatable. See the Screen Memory section earlier in this chapter.

## HOW TO INTERPRET THE VIDEO MATRIX

The contents of the video matrix are interpreted the same way as in standard bit map mode. The upper nybble is the color code for the foreground color; the lower nybble is the color code for the background color of the bit map.

## ADDITIONAL COLOR DATA

The upper and lower nybbles of screen memory supply the multi-color bit map screen with two of the color sources. This mode offers two additional colors—background color register 0 (location 53281) and the lower nybble of color RAM.

As in multi-color character mode, the bit patterns of the bytes in the bit map determine the color assignments for the pixels on the screen. The bits are similarly grouped in pairs, within 8-bit bytes, so there are 4-bit pairs in each byte. Bits 0 and 1, 2 and 3, 4 and 5, and 6 and 7 are grouped in pairs respectively. Depending on the values of the bit pairs, the corresponding pixels in the bit map are assigned colors from the sources in Figure 8–26.

| BITS | COLOR INFORMATION COMES FROM |
|---|---|
| 00 | Background color #0 (screen color) |
| 01 | Upper four bits of video matrix |
| 10 | Lower four bits of video matrix |
| 11 | Color RAM |

**Figure 8–26. Multi-Color Bit Map Pixel Color Assignments**

## THE BIT MAP

Bit patterns determine how color is assigned to the multi-color bit map screen. If the bit pair is equal to 00 (binary), color is taken from background color register 0 (location 53281). If the bit pair is equal to 01 (binary), the color assigned to these two pixels comes from the upper nybble of video matrix. If the bit pair in the bit map is equal to 10 (binary), then the color assigned to those two pixels comes from the lower nybble of video matrix. Finally, if the bit pair in the bit map is equal to 11 (binary), the color is taken from the lower four bits of color RAM. Unlike multi-color character mode, the screen is either all standard bit map, or all multi-color bit map, unless you develop a sophisticated interrupt-driven application program that handles the two separate bit maps.

## COLOR RAM

In multi-color bit map mode, color RAM is used if the bit pair from the bit map equals 11 (binary). Each color RAM location may have one of sixteen color codes, which means that one 8 by 8 bit map area can have black, red, white and blue colors, respectively, for the background color register 0, the upper nybble, the lower nybble, and the color RAM. The 8 by 8 multi-color bit map area next to it can have black, red, white and green colors, since each color RAM location is independent of any other. The other three color sources usually remain constant throughout a bit map screen, though you can change the upper and lower nybbles of the video matrix. The background color register is almost always the same throughout a bit map screen.

The C128 has powerful and varied graphics display capabilities. Certain applications call for one type of display over another. Experiment with them all and see which one meets your needs best. Figures 8–28 through 8–32 provide a graphics programming summary that should be helpful in understanding graphics on the C128.

# SPLIT-SCREEN MODES

The Commodore 128 has a split-screen feature that allows you to display the top portion of the screen in bit map mode and the bottom portion in character mode. This allows you to enter a BASIC graphics program and RUN it while the BASIC program listing is

present and the bit map image is displayed, which saves time switching back and forth between bit map and character modes.

Before, you would have had to enter the graphics program (in machine language), RUN it and switch back to the text screen to make a change. Now you can display the graphic image *and* have your text screen available to you all at the same time. You can alter the program while your bit map image is still on the screen, RUN it and see the immediate results without losing the text screen.

Without the Commodore 128's split-screen capabilities, you would have to program a split screen yourself. This involves raster interrupts which utilize either two screen memories in two different video banks, or a fairly choppy single-screen memory, usually with a visible raster line. With the C128 split-screen mode, all you have to do to enter a split-screen mode is to issue the GRAPHIC command in BASIC. For example, the command:

**GRAPHIC 2,1**

sets up a standard bit map screen on top and a text screen on the bottom. Similarly, the command:

**GRAPHIC 4,1**

constructs a multi-color bit map screen on the top portion of the screen and a text screen on the bottom portion. The "1" in these commands clears the bit map screen. To leave the bit map screen intact, once you have already displayed an image, replace the "1" with a zero (0).

The GRAPHIC command has an additional parameter that allows you to define where the split occurs. The split-screen starting location is defined in terms of a character row, as if the C128 were in a character display mode. For example,

**GRAPHIC 4,1,15**

selects a split screen with multi-color bit map mode on top of the screen and the text screen on the bottom, starting at character row 15. If the start of the split screen is not defined, the C128 defaults the start to line 19.

# HOW SPLIT-SCREEN MODES ARE ORGANIZED IN MEMORY

## SCREEN LOCATIONS

The split-screen modes, both multi-color and standard, use two independent screen memories. The bit map video matrix is taken from the address range 7168 ($1C00) through 8191 ($1FFF), just as in standard and multi-color bit map modes. The text portion of the screen takes its screen memory from default character mode screen locations 1024 ($0400) through 2023 ($07E7), just as in standard and multi-color character mode. The hidden portions of the screen, the bottom portion of the bit map screen and the upper portion of the text screen, still store data, but it is invisible since the other screen memory has overlaid it.

### INTERPRETING SCREEN DATA

The text portion of the split screen is interpreted according to the standard character mode section. The bit map portion, whether standard or multi-color, is interpreted according to the description in the bit map mode section. Consult the Standard Character Mode, Standard Bit Map Mode and Multi-color Bit Map Mode sections for information on how screen data is interpreted.

### CHARACTER MEMORY LOCATIONS

The split-screen modes also take character data from two independent parts of memory. The bit map data, referred to simply as the bit map, is taken from the default range 8192 ($2000) through 16191 ($3FE7) for both the multi-color and standard bit map mode portions of the screen.

The character memory for the text portion of the split screen is derived from the character ROM. The actual character ROM occupies the memory locations 53248 ($D000) through 57343 ($DFFF) overlaying the I/O registers. The I/O registers must be switched out to view the actual character ROM, in bank configuration 14, for example.

For information on how character data is interpreted in standard character, standard bit map and multi-color bit map modes, see the sections describing these modes. See also the Color RAM Banking section.

## COLOR DATA

Each of the standard bit map, multi-color bit map and standard character modes interpret color differently. See each section for detailed information on color assignments.

## MACHINE LANGUAGE

In machine language, you must program a split screen yourself. This is not the easiest of programming tasks, since it involves raster interrupt processing, which can be tricky. In C128 mode, bit 6 in the GRAPHM register is the split screen bit. If bit 6 of $0008 (GRAPHM) is set, a split screen is displayed. Otherwise, bit 6 is clear (0) and a single screen is displayed.

The C64 mode has no corresponding split screen bit. C64 mode is programmed differently for split screens. See the Raster Interrupt Split-Screen Program at the end of the chapter to learn how to program a split screen in machine language.

> **CAUTION:** A system crash may occur if the display mode is changed while the interrupt-driven screen editor is enabled. See the Shadow Register section for details.

## THE INTERRUPT-DRIVEN SCREEN EDITOR

The intermediate memory locations, sometimes referred to in this guide as shadow registers, are designed specifically for handling the split-screen modes. In order to provide split-screen modes, the C128 screen editor has to be wedged into the system's interrupt handling routines.

Unlike the Commodore 64, the C128 handles interrupts exclusively according to the raster beam. This has made it necessary to merge the C128 screen editor into the interrupt request routines (IRQ). The C64 uses interrupt timers which makes interrupt processing less predictable. By processing the interrupts from the raster beam, the operating system always knows where and when the interrupt will occur. Timer interrupts made catching a raster interrupt less reliable because the operating system never knew exactly where an interrupt would occur in relation to the raster beam.

The raster interrupt-driven screen editor made it necessary to use indirect storage locations for certain registers of the VIC chip and 80-column chip. This way, the intermediate memory locations refresh the actual video chip registers every sixtieth of a second, each time the raster beam begins a new scan at the top of the screen. The raster beam scans the entire screen sixty times a second, so on each pass of the raster beam the intermediate memory locations refresh the actual VIC chip and 8563 chip registers.

## RASTER INTERRUPT SPLIT SCREEN PROGRAM WITH HORIZONTAL SCROLLING

This section explains how and provides a program to perform split screen raster interrupts in machine language. The program is explained as it applies to Commodore 64 mode, but it can be modified to run in Commodore 128 mode.

You already have a way to split the screen in C128 mode with the BASIC GRAPHIC command. The program provided in this section splits the screen in machine language in C64 mode. See the figure in the shadow register section to see which addresses must be changed to make this program work in C128 Mode. A few differences will occur in the timing of the raster. In Commodore 128 mode, all interrupts occur according to the position of the raster beam as it scans the screen. This is why shadow registers are necessary for certain graphics locations, since the C128 screen editor is interrupt driven to allow the split screen modes in BASIC.

The program in this section also scrolls text on the bottom quarter of a standard character screen, while the top three quarters are displayed in multi-color bit map mode. The standard character screen resides in video bank 0 ($0400–$07E7) while the multicolor bit map video matrix is stored in video bank 1 starting at $5C00 ($1C00 + $4000 = $5C00). Every time an interrupt occurs, the program changes video banks, display modes, the character memory and video matrix pointers. This program supplies the data that scrolls at the bottom of the screen, but it assumes you have placed an 8000 byte bit map starting at address 8192 ($2000) plus an offset of 16384 ($4000) for the change of video banks. This makes the absolute start address of the bit map 24576 ($6000).

An 8000 byte bit map is just too large to present in this book. However, the easy way to place a bit map in this area is as follows:

1. First start in C128 mode, and enter split screen (multi-color) bit map mode through BASIC with this command:

   **GRAPHIC 4,1**

2. Now draw on the screen with the BOX, CIRCLE, DRAW and PAINT commands either in a program or in direct mode.
3. When you are finished drawing, enter the machine language monitor either by pressing the F8 function key or by typing the MONITOR command.
4. Now transfer the video matrix and bit map from the C128 default locations of $1C00 through $1FFF and $2000 through $3FFF respectively, to $5C00 through $5FFF and $6000 through $7FFF respectively. The new start addresses are the default locations plus an offset of $4000 for both the video matrix and bit map pointers. The new start of the video matrix is at $5C00 ($1C00 + $4000). The new bit map begins at address $6000 ($2000+$4000). This transfer can be accomplished with a single transfer command within the machine language monitor as follows:

   **T 1C00 3FFF 5C00**

This command transfers the contents of memory locations $1C00 through $3FFF to $5C00 through $7FFF. Since the default locations of the video matrix and bit map are continuous in memory, the transfer can be done with a single command. If the default addresses of the video matrix and bit map had not been contiguous, two transfers would have been necessary. See Chapter 6 for details on using the Machine Language Monitor.

Now that you are still within the control of the Machine Language Monitor, begin entering the machine language instructions in the listing provided in the next few pages. Start entering the program at address $0C00. The program, including the scrolled data occupies memory up to address $0DF9, which means the program is a total of 505 bytes, or almost half a kilobyte (K).

Now save the program you just painstakingly entered with the Monitor Save (S) command as follows:

**S "filename", 08, 0C00, 0DFF**

If you have a C128 assembler, create a source file, assemble and load it. If your assembler allows it, save the program as a binary file.

> **NOTE:** If you have the Commodore 64 Assembler Development System, create a source file (with start address $0C00), assemble and load it in C64 mode. Then press the RESET button (not the ON/OFF switch) to enter C128 mode. Don't worry, your program will still be in memory, but this time it's in C128 memory. Next enter the Machine Language Monitor and use the Monitor Save (S) command to make a binary file as described above.

Now you are ready to enter C64 mode and run the program with the following command:

**GO 64**

Reply to the question "ARE YOU SURE?" by pressing the "Y" key and RETURN. You are now placed in C64 mode.

At this point, you may say to yourself, "After I just did all that work, why am I going to waste it by changing modes?"

Actually, you are not wasting any effort. When you GO 64 (or press the reset button), much of the RAM for machine language programs and data is preserved in RAM bank 0. BASIC programs are erased, however. Specifically, these are the ranges of RAM that are preserved when changing between C128 and C64 modes:

| C128 MEMORY LAYOUT | |
|---|---|
| **$0C00 - $0DFF** | **RS232 Input and Output Buffers** |
| **$1300 - $1BFF** | **Available Memory for Machine Language application programs** |
| **$1C00 - $1FFF** | **Bit Map Video Matrix** |
| **$2000 - $3FFF** | **Bit Map Data** |
| **$4000 - $FF00** | **BASIC Text Area** |

**Figure 8–27. Preserved RAM Between C128 and C64 Modes**

The rest of the RAM memory is allocated for other purposes and the contents change from mode to mode. There are other particular bytes that are preserved from mode to mode, but these small chunks of memory are not worth mentioning here. The blocks of memory mentioned above provide enough of a clue to the RAM used by both modes. Remember, however, that the RAM is only preserved if you switch from C128 mode to C64 mode with the GO 64 command, or you switch from C64 to C128 mode with the reset button (warm start). If you perform a cold start, turn the computer power off, then on again, all RAM is cleared and none is preserved.

Notice that the address ranges where you placed your program, the video matrix and the bit map are in the portions of RAM that are preserved from mode to mode.

Now start (run) the program from C64 BASIC with this command:

**SYS 12*256**

The top three quarters of the screen is the bit map screen you created with the C128 BASIC graphics commands, the lower quarter is horizontally scrolling text.

The following program is the listing that performs the split screen and scrolling.

```
1000 ; RASTER INTERRUPT SPLIT SCREEN WITH HORIZONTAL SCROLLING
1010 IVEC=$0314
1020 TEMP=$1806
1030 RASTRO=$D012
1040 BACOL=$D021
1050 POINT=$1802
1060 FLAG=$FC
1070 FLAG2=$FD
1080 SCROLL=$FE
1090 SREG=$D016
1100 TXTPTR =$FA   ;POINTER INTO SCROLLING TEXT
1110 ;
1120  *=$0C00
1130 ;
1140  LDA #<FRANK ;POINT TO TEXT
1150  STA TXTPTR
1160  LDA #>FRANK
1170  STA TXTPTR+1
1180 ;
1190  LDA #$07 ;SET HI VAL SREG
1200  STA SCROLL
1210  STA FLAG2
1220 ;
1230  LDA $DD02 ;SET TO OUTPUT
1240  ORA #$03
1250  STA $DD02
1260 ;
1270 ;
1280  LDA $D016 ;SET 38 COL.
1290  AND #$F7
1300  STA $D016
1310 ;
1320 ;
1330  LDA #$01 ;CLEAR CRAM
1340  LDX #$00
1350 BONK  STA $DB20,X
1360  INX
1370  CPX #200
1380  BNE BONK
1390 ;
1400 ;
1410 ;   INITIALIZE INTERRUPT
1420 ;
1430 LOOP1 SEI
1440  LDA IVEC
1450  STA TEMP
1460  LDA IVEC+1
1470  STA TEMP+1
1480  LDA #<MINE
1490  STA IVEC
1500  LDA #>MINE
1510  STA IVEC+1
1520 ;
1530  LDA #$00 ;DISABLE TIMER IRQ
1540  STA $DC0E
1550  STA FLAG
1560 ;
1570  LDA #49
1580  STA POINT
1590  STA RASTRO
1600 ;
1610  LDA #201
1620  STA POINT+1
1630 ;
1640 ;
1650  LDA #1
1660  STA $D01A ;ENABLE INTERRUPT
1670  STA $D019 ;RASTER COMPARE
1680  LDA $D011 ;CLEAR HI BIT
```

```
1690  AND #$7F
1700  STA $D011 ;DISABLE INTRPT DIS BIT
1710 ;
1720  CLI
1730 ;
1740 ;
1750 ;
1760 ;
1770 CHECK LDA FLAG2
1780  BPL CHECK
1790  LDA #$00
1800  STA FLAG2
1810 ;
1820  LDX #39
1830  LDY #39
1840 SHIFT LDA (TXTPTR),Y
1850  STA $0770,X
1860  DEX
1870  DEY
1880  BPL SHIFT
1890  INC TXTPTR
1900  BNE MVTIME ;TIME TO RESET POINTER
1910  INC TXTPTR+1
1920 MVTIME LDA TXTPTR ;ARE WE AT THE END OF THE TEXT YET ?
1930  CMP #<ENDTXT
1940  BNE CHECK
1950  LDA TXTPTR+1
1960  CMP #>ENDTXT
1970  BNE CHECK
1980  LDA #<FRANK ;SET POINTER BACK TO THE BEGINNING
1990  STA TXTPTR
2000  LDA #>FRANK
2010  STA TXTPTR+1
2020  JMP CHECK
2030 ;
2040 FRANK .BYT
2050  .BYT ´
2060  .BYT ´THIS IS AN EXAMPLE OF SCROLLING
2070  .BYT ´IN THE HORIZONTAL (X) DIRECTION.
2080  .BYT ´ONCE THE DATA HAS BEEN DISPLAYED,
2090  .BYT ´SCROLLING STARTS AGAIN FROM THE ´
2100  .BYT ´BEGINNING.´
2110 ;
2120 ENDTXT .BYT
2130  .BYT
2140 ;
2150 ;  INTERRUPT SERVICE ROUTINE
2160 ;
2170 MINE LDA $D019
2180  STA $D019
2190 ;
2200  LDA FLAG ;FLAG =0  .A=0
2210  EOR #1 ;FLAG =0  .A=1
2220  STA FLAG ;FLAG =1  .A=1
2230  TAX ;.X=1     .A=1
2240  LDA POINT,X ;.X=1   .A=200
2250  STA RASTRO ;.X=1 RASTRO=200
2260  CPX #1
2270  BNE BOTTOM
2280 ;
2290  LDA $D011 ;BIT MAP MODE
2300  ORA #$20
2310  STA $D011 ;BMM
2320  LDA $D018
2330  ORA #$78 ;CHAR=$2000+$4000
2340  STA $D018 ;SCR=$1C00+$4000
2350  LDA $D016
2360  ORA #$10
2370  STA $D016;SET MULTICOLOR
```

```
2380 ;
2390  LDA $DD00 ;SELECT BANK 1
2400  AND #$FE
2410  STA $DD00
2420 ;
2430  LDA SREG ;SET SCRRREG
2440  AND #$F8
2450  ORA #$03
2460  STA SREG
2470 ;
2480  LDA #14
2490  STA $D021
2500 ;
2510  PLA
2520  TAY
2530  PLA
2540  TAX
2550  PLA
2560  RTI
2570 ;
2580 ;
2590 BOTTOM LDA $D011 ; SET TEXT MODE
2600  AND #$FF-$20
2610  STA $D011
2620 ;
2630  LDA $DD00 ;CHANGE TOBANK 0
2640  ORA #$01
2650  STA $DD00
2660 ;
2670  LDA $D016 ; DISABLE MULTI
2680  AND #$FF-$10
2690  STA $D016
2700 ;
2710  LDA #23 ;   MEMORY
2720  STA $D018
2730 ;
2740  LDA #$00
2750  STA $D021
2760  LDA SCROLL
2770 ;
2780 ;
2790  DEC SCROLL
2800  LDA SCROLL
2810  STA FLAG2
2820  CMP #$FF
2830  BNE SLURP
2840  LDA #$07
2850 SLURP STA SCROLL
2860  LDA SREG
2870  AND #$FF-$07
2880  ORA SCROLL
2890  STA SREG
2900 BASIRQ JMP (TEMP)
2910 ;
2920 .END
```

For readability, the program is listed as a source file, as though it was entered through an assembler editor. It is easier to understand as a source file rather than a listing from the Machine Language Monitor. To enter this program into the Machine Language Monitor, reference the actual address in place of the variable operand addresses. Most of the actual addresses are listed in the beginning of the program (lines 1010 through 1100). Keep in mind that these are only line numbers for an assembler

editor. You will enter the program into a memory location number (address) in the Machine Language Monitor. In this case, the program is stored in memory starting at address $0C00. Line 1120 specifies this start address with:

```
* = $0C00
```

Here's an instruction-by-instruction explanation of the scrolling split screen program.

Line 1010 assigns the variable IVEC to the address $0314, the hardware interrupt request (IRQ) vector. The interrupt vector is the means by which the Commodore 128 displays split screens and scrolling. By wedging your own routine into the hardware interrupt vector (in this case scrolling and splitting the screen), it enables you to perform operations that usually take too long for the microprocessor to perform under an application program not using interrupts. The interrupt vector is checked for an interrupt routine every 60th of a second. In this program, the screen is split 60 times per second, so it appears you have two different screens displayed at the same time. You could not split the screen without requesting an interrupt; the microprocessor is not able to perform all the required operations fast enough to keep up with the raster scan of the video controller. The speed that the screen is continually updated, known as the raster scan, also occurs at the speed of 60 times per second. For a split screen to occur, you tell the computer the point on the screen where one type of display ends and the new one (bit map for example) begins. The way you tell the computer this is by placing the number of a pixel row, also called a raster row, in the raster compare register located at address $D012. Line 1030 assigns this address to the variable RASTRO.

You'll see later in the program that the value placed in the Raster Compare Register starts the text screen at raster row 201. The raster scan is again interrupted at raster row 50 at the top of the screen to display the bit map screen. This is repeated 60 times every second, so it appears to the human eye that two different display modes are active at the same time.

On with the program explanation. Line 1040 defines the BACOL variable for the background color register zero, located at address $D021. Line 1050 assigns the variable POINT to location $1802. POINT is used to store the raster row value where the text screen begins. Line 1060 assigns the variable FLAG to location $FC. FLAG is used later in the program (lines 2200–2270) to determine where the interrupt occurred, either raster row 50 or raster row 200.

Lines 1070 and 1080 assign the variables FLAG2 to location $FD and SCROLL to location $FE respectively. Both FLAG2 and SCROLL store the value of the scrolling register. SREG is assigned to location $D016, the scrolling register. Only bits 0 through 2 are used as the scrolling bits. The other bits in this address are used for other purposes. Three scrolling bits are necessary since characters that are scrolled are moved over seven pixels then shifted to the next character position to the left or right, then scrolled smoothly again seven more pixels.

Line 1100 assigns the variable TXTPTR to address $FA. This variable marks the starting address in memory where the scrolled characters are stored.

As was mentioned earlier, line 1120 specifies where the program storage begins in memory. This is the address where the execution of the program begins in memory. You

will SYS to this address to start the program in BASIC, or GO to this address from the Machine Language Monitor.

The first sequence of program instructions starting at line 1140 places the contents of the address (named FRANK) of the beginning of the scrolling text into the memory locations ($FA and $FB) called TXTPTR and TXTPTR + 1. FRANK is the label in line 2040 which marks the location where the first scrolled character is stored. In this program the first character is a space; in fact the first forty characters that are scrolled across the screen are spaces. The forty-first character marks the beginning of the data 'THIS IS AN EXAMPLE OF SCROLLING' in line 2060. The scrolled data in lines 2040 through 2130 is stored starting at location $0C98, once this source file is assembled into object code. In this case, the low byte stored in $FA is $98 and the high stored in $FB is $0C. The full 16-bit address $0C98 is important, since this is the base address which you increment as you scroll each letter across the VIC screen. When the text pointer (TXTPTR) reaches the end of the scrolling text, the base address $0C98 is again stored in TXTPTR.

The second sequence of instructions starting at line 1190 sets the high value of the two scrolling variables SCROLL ($FE) and FLAG2 ($FD) to 7. These are used and will be explained later in the program.

Sequence three starting at line 1230 sets the data direction register to output.

The next module of instructions in lines 1280 through 1300 set the screen size to 38 columns, reducing the screen width by a column on each side. In order to scroll smoothly, you must set the screen size to 38 columns. Clearing bit 3 of location $D016 sets 38 column size. Setting bit 3 restores the VIC screen to its normal 40 column size.

The extra column on each side of the screen border provides a place for the scrolled data to scroll smoothly to and from offscreen. This program scrolls left, so each newly scrolled character is placed in column 39 on the right, before it becomes visible in column 38. At the same time, the lead character on the left scrolls from column 2 to offscreen column 1. This occurs in lines 1770 through 2020 and is explained as the program progresses.

Lines 1330 through 1380 set the text screen color RAM to a white foreground. The lower four bits specify the foreground character color in standard character mode.

The screen memory is stored in video bank 0 where the scrolling text appears at the bottom fourth of the screen. The bit map and video matrix are stored in video bank 1, the 16K range between $4000 and $7FFF.

The reason only 200 color RAM locations are filled is because only the lower five rows are visible on the text screen. There is no point clearing the other 800 locations since they are not visible.

Lines 1410 through 1700 make up the interrupt initialization routine. Line 1430, labeled LOOP1, sets the interrupt disable bit in the status register. When this bit is set, interrupts are disabled and none can occur. Only when the interrupt disable bit is cleared

(0) can interrupts occur. The last line (1720) of the interrupt initialization routine clears the interrupt disable and allows interrupts to occur.

Lines 1440 through 1470 store the original contents of the Interrupt Request (IRQ) vector into temporary storage locations TEMP (low byte $1806) and TEMP + 1 (high byte $1807). This is necessary in order to store the original contents of the IRQ vector so you can jump back to this location once the interrupt is serviced as in line 2900.

Lines 1480 through 1510 store the starting location of the interrupt service routine into the IRQ vector. In this case, MINE is the source file label in line 2170 where the interrupt service routine begins. In the assembled object file, as in the Machine Language Monitor, the low byte is $71 and the high byte is $0D, to form the 16 bit address $0D71. Once the interrupt disable bit is cleared and an interrupt occurs, the 8502 microprocessor finishes executing its current instruction and sets the interrupt disable status bit, so no other interrupts can occur. The processor then places the contents of the high byte and low byte of the program counter and the status register on the stack respectively. Finally, the 8502 fetches the address contained in the IRQ vector and executes the routine starting at this address, in this case $0D71.

Lines 1530 and 1540 disable the CIA timer in location $DC0E. In addition, line 1550 initializes the variable FLAG to zero. This variable is used later in the program to figure out where the raster interrupt has occurred, either at the top of the screen (raster row 49) for bit map mode or near the bottom fourth of the screen (raster row 201) for standard character mode.

The instructions in lines 1570 through 1590 define the variable POINT ($1802) as the value of the raster row 49 ($31) where the interrupt occurs to select bit map mode. In addition, this value is also stored in the Read/Write Raster Register ($D012) for raster row comparisons later in the program.

The C128 VIC screen consists of 200 raster rows, each row one pixel tall, having 320 pixel columns. You know how BASIC addresses the bit map coordinates on a coordinate plane of 0,0 in the top left corner and 319,199 in the bottom right. However, the visible raster rows are not labeled in the same way. The visible raster rows start at 50 at the top of the screen and end at 250 at the bottom. These are the same row numbers as sprites use. Notice there are still 200 rows but that they offset the bit map coordinate row number by 50. The raster row numbers below 50 and above 250 are off the visible screen. These offscreen raster rows are referred to as the vertical retrace.

The instructions in lines 1610 and 1620 define the variable POINT + 1 ($1803) as the value of the raster row 201 ($C9) where the interrupt occurs to select standard character mode.

Lines 1650 and 1660 enable the raster IRQ Mask Register. Line 1670 sets the Raster Compare IRQ Flag. By setting bit one in these registers, the raster interrupt attached to the IRQ line is allowed to occur (once lines 1680 through 1720 are executed), depending upon whether the physical raster row compares and matches with either of the values in POINT or POINT + 1. If either of these match, the interrupt occurs.

The instructions in lines 1680 through 1700 clear the raster compare high bit (bit 8). This is an extra bit from location $D012 for raster compares.

The instruction in line 1720 clears the interrupt disable status bit, which enables interrupts to occur. This is the last operation to be performed by the interrupt initialization routine. Now interrupts are ready to occur and be serviced.

Lines 1770 through 1800 check the value of FLAG2. If FLAG2 is positive, the program branches to the label CHECK in line 1770 and checks the value of FLAG2 again. The value stored in FLAG2 represents the value of the lower three bits in the horizontal scrolling register at location $D016. The variable SCROLL is also associated with the variable FLAG2. In the interrupt service routine (in lines 2760 through 2810), the value in SCROLL is decremented and stored in FLAG2. This value pertains to the value placed in the actual horizontal scrolling register at $D016. The reason this is counted is as follows.

The direction of the scrolling is right to left; therefore, you must place the maximum value (7) in the lower three bits of $D016 and decrement that value by one. If the program had scrolled left to right, you would initialize the scrolling register to zero and increment the lower three bits. When the scrolling register value is decremented, the characters in the screen memory locations which are to be scrolled are moved to the left by one pixel. Each time the scrolling register is decremented, the characters move another pixel to the left. When the value (the lower three bits) of the scrolling register equals zero, you must move the scrolled characters up in screen memory by one location. This routine is contained in lines 1840 through 1880. After the shift, the lower three bits of the scrolling register are set back to 7, the characters are again shifted by the VIC chip 7 pixels to the left and your routine shifts the characters up in memory again by one. The additional details are covered in the explanation of lines 2760 through 2900.

The instructions in lines 1820 through 1880 shift the text to be scrolled up in memory by one location for each cycle of the loop. First the X and Y registers are loaded with the decimal value 39 ($27). The instruction in line 1840 loads the value of the memory location where the scrolled text begins using indirect Y addressing. The address is calculated by taking the contents of zero page memory variable TXTPTR ($98) and adding the offset 39 to its contents to arrive at $BF. The effective address gives the low byte where the scrolled data begins. The first scrolled character is actually a space. Subsequent data elements are accessed by modifying the value of the Y register.

The store instruction in line 1850 stores the first character of the scrolling text in screen location 1943 ($0770 + $27), which is the fortieth column of the twenty-third row. This column is not visible when the screen size is set to 38 columns for horizontal scrolling. Each newly scrolled character must be placed in this position in order to scroll smoothly from the offscreen location. Lines 1860 and 1870 decrement the X and Y registers respectively. Line 1880 branches to the label SHIFT if the Y register is positive (greater than zero).

The second time through the loop, the low byte of the scrolled character (at location $98+$26=$BE) is stored in screen location 1942 ($0770+$26), so the (space) character in $0CBE is stored in screen location $0796. The third time through the loop, $0CBD is stored in $0795 and so on. This process continues until the X and Y registers equal zero. So far, only the series of 39 space characters in lines 2040 and 2050 have been shifted across the screen.

Once the X and Y registers have been decremented to zero, the TXTPTR is incremented in line 1890 so that subsequent characters such as "THIS IS AN EXAMPLE . . ." can be scrolled. In the first 39 cycles through the loop (in lines 1840 through 1880) 39 spaces are shifted (scrolled) one character position on the twenty-third character row on the screen. The next 39 cycles shift 38 spaces and the letter "T" in "THIS" one character position across the screen. The next 39 cycles, 37 spaces and the letters "TH" in "THIS" are shifted in memory and scrolled one character position on the screen and so on. This process occurs until all characters in the data (in lines 2040 through 2130) are scrolled.

Line 1900 branches to the label MVTIME while TXTPTR (the low byte of the start of scrolled data) is greater than zero, otherwise TXTPTR + 1 incremented to update the high byte. Lines 1920 through 1970 check to see if the text pointers are at the end of the scrolled character data ($0D70 in the assembled program). If the pointers are not at the end of the character data, the program branches to the label CHECK and the data is shifted by the VIC chip by seven pixels and the scrolling process repeats again. If the text pointers are at the end of the scrolled character data, lines 1980 through 2010 set the text pointers to the beginning of the scrolled data in memory and the process is repeated continuously as specified by the JMP CHECK instruction.

Lines 2040 through 2130 represent the data to be scrolled by the program. The data is stored in .BYTE statements as it appears in the Commodore Assembler 64 Development System. In your case, the Machine Language Monitor handles data by simply storing it in an absolute memory range. In the assembled object code program the data turns out to be stored in the range $0C98 through $0D70. You will refer to the data with these addresses and not with a label as in this explanation.

## THE INTERRUPT SERVICE ROUTINE

The program instructions in lines 2170 through the end of the program make up the interrupt service routine. Depending upon the value of the raster comparison, particular segments of the routine are executed upon the detection of an interrupt. For instance, if the result of the raster comparison detects an interrupt to occur at raster row 49, lines 2290 through 2560 are executed. This selects bit map mode and performs additional functions that are explained in a moment. If the raster comparison detects the interrupt to occur at raster row 201, lines 2590 through 2900 are executed. These instructions select standard character mode, among other things. Keep in mind that both segments of the interrupt service routine are executed within a single, complete raster scan of the screen sixty times per second. Instructions 2170 through 2270 are always executed when an interrupt occurs.

The instruction in line 2170 clears the raster compare IRQ flag after the interrupt has occurred. The address of the label MINE, which is loaded into the IRQ vector in lines 1480 through 1510, tells the 8502 where the interrupt service routine resides upon the occurrence of an interrupt. In the assembled object code, this is an absolute address ($0D71).

Lines 2200 through 2270 determine the location (raster row) in which the raster interrupt has occurred. Line 2200 loads the value of FLAG, which was initialized to zero, into the accumulator. Line 2210 XOR's the accumulator with 1, which effectively places a one in the accumulator for the first pass through this routine. This value is then stored back into FLAG. In each subsequent occurrence of an interrupt, the value of both the accumulator and FLAG are toggled between zero and one. The accumulator is then transferred to the X register in line 2230. Line 2240 loads the value of POINT or POINT + 1 depending upon the value in the X register. If the X register equals 0, POINT is loaded into the accumulator, which specifies the interrupt to occur at raster row 49. If the X register equals 1, POINT + 1 is loaded into the accumulator, which specifies the interrupt to occur at raster row 201. Line 2250 stores the accumulator value into the variable RASTRO. The X register is compared with 1 in line 2260. If the X register equals 1, the interrupt has occurred at raster row 201 and the program branches to the instructions in lines 2590. If the value of the X register equals 0, the branch in line 2270 falls through and the instructions in lines 2290 through 2650 are performed.

The instructions in lines 2290 through 2560 perform the operations associated with bit map mode. Lines 2290 through 2310 select bit map mode. Lines 2320 through 2340 set the video matrix at $1C00 and the bit map at $2000. Both of these start addresses are offset by the compulsory $4000, since this screen appears in video bank 1 ($4000-$7FFF). Lines 2350 through 2370 set multi-color mode. Lines 2390 through 2410 select video bank 1. Lines 2430 through 2460 set the lower two bits of the scrolling register (to the value 3).

The instructions in 2510 through 2550 restore the original values of the X, Y and A (accumulator) registers. Line 2560 returns from the interrupt and exits the interrupt service routine.

Lines 2590 through 2900 perform all the associated text mode operations. Lines 2590 through 2610 select standard character mode. Lines 2630 through 2650 changes back to video bank 0, the default bank ($0000-$3FFF). Lines 2670 through 2690 disable multi-color mode, and return to the standard color mode. Lines 2710 and 2720 set the default screen location 1024 ($0400) and the default start of character memory, with the decimal value 23 ($17). All numbers which are not preceded by a dollar sign are assumed to be decimal in this particular assembler editor. Lines 2740 and 2750 set the background color for the text screen to black.

Lines 2760 through 2890 set the value of the scrolling register, which scrolls the characters across the screen by 7 pixels, before they are shifted in memory with the routine in lines 1840 through 1880.

Finally, line 2900 jumps to the default IRQ vector which was saved early in the program into the variable TEMP. This allows the 8502 to process the normal interrupt services as though this program's service routine had not occurred.

Although this program example is long and complex, it contains useful routines and explanations that have never appeared before in any Commodore text. Study these routines carefully and add them into your own programs. This section includes a wealth of information for the novice and experienced software developer alike. Figures 8–28 through 8–32 on the following four pages provide a summary of graphics programming.

| | CHANGING VIDEO BANKS | MOVING SCREEN RAM |
|---|---|---|
| **C128 BASIC** | **POKE 56576, (PEEK (56576) AND 252) OR X**<br>WHERE X IS THE DECIMAL VALUE OF BITS 0 AND 1 IN TABLE 8-30 ON PAGE 262 | TEXT<br>**POKE 2604, (PEEK(2604 AND 15) OR X**<br>WHERE X IS THE DECIMAL VALUE IN TABLE 8-29 ON P. 262<br>BIT MAP<br>**POKE 2605, (PEEK(2605 AND 15) OR X**<br>WHERE X IS A VALUE IN TABLE 8-29 ON P. 262 |
| **C128 MACHINE LANGUAGE** | **LDA $DD00**<br>**AND #$FC**<br>**ORA #$X**<br>**STA $DD00**<br>WHERE X IS THE HEX VALUE OF THE BITS IN TABLE 8-30 ON P. 262 | TEXT / BIT MAP<br>**LDA $0A2C** / **LDA $0A2D**<br>**AND #$0F** / **AND #$0F**<br>**ORA #$X** / **ORA #$X**<br>**STA $0A2C** / **STA $0A2D**<br>WHERE X IS A HEX EQUIVALENT OF THE DECIMAL VALUE IN FIGURE 8-29 ON P. 262 |
| **C64 BASIC** | **POKE 56576, (PEEK (56576) AND 252) OR X**<br>WHERE X IS THE DECIMAL VALUE OF BITS 0 and 1 IN TABLE 8-30 on P. 262 | TEXT OR BIT MAP<br>**POKE 53272, (PEEK(53272) AND 15) OR X**<br>WHERE X IS A DECIMAL VALUE IN FIGURE 8-29 ON P. 262 |
| **C64 MACHINE LANGUAGE** | **LDA $DD00**<br>**AND #$FC**<br>**ORA #$X**<br>**STA $DD00**<br>WHERE X IS THE HEX VALUE OF BITS 0 AND 1 IN TABLE 8-30 ON P. 262 | TEXT OR BIT MAP<br>**LDA $D018**<br>**AND #$0F**<br>**ORA #$X**<br>**STA $D018**<br>WHERE X IS A HEX EQUIVALENT OF THE DECIMAL VALUE IN FIGURE 8-29 ON P. 262<br>IN C64 MODE, YOU CAN SET UP TWO DIFFERENT SCREENS, ONE FOR TEXT AND THE OTHER FOR BIT MAP, AS THE C128 KERNAL DOES. |

**Figure 8-28. Graphics Programming Summary—PART 1**

| MOVING CHARACTER MEMORY | ACCESSING CHARACTER ROM |
|---|---|
| TEXT<br>**POKE 2604, (PEEK(2604)AND 240) OR Z**<br>BIT MAP*<br>**POKE 2605, (PEEK (2605)AND 240) OR Z**<br>WHERE Z IS A DECIMAL VALUE IN FIGURE 8–31 ON P. 262<br>*= ONLY BIT 3 IS SIGNIFICANT IN BIT MAP MODE | **10 BANK 14: REM SWAP IN CHAR ROM**<br>**20 FOR I=0 TO 7**<br>**30 ? PEEK (I)**<br>**40 NEXT**<br>**50 BANK 15** |
| TEXT BIT MAP*<br>**LDA $0A2C LDA $0A2D**<br>**AND #$F0 AND #$F0**<br>**ORA #$Z ORA #$Z**<br>**STA $0A2C STA $0A2D**<br>WHERE Z IS THE HEX EQUIVALENT OF A DECIMAL VALUE IN FIGURE 8-31 ON P. 262<br>*= ONLY BIT 3 IS SIGNIFICANT IN BMM | **LDA #$01**<br>**STA $FF00**<br>**LDX #$00**<br>**LDA #$D000,X**<br>**LOOP STA $TEMP,X**<br>**INX**<br>**CPX #$07**<br>**BNE LOOP**<br>**LDA #$00**<br>**STA $FF00** |
| TEXT OR BIT MAP*<br>**POKE 53272, (PEEK(53272)AND240) OR Z**<br>WHERE Z IS A DECIMAL VALUE IN FIGURE 8-31 ON P. 262<br>*= ONLY BIT 3 IS SIGNIFICANT IN BIT MAP MODE | **5 TEMP=6144**<br>**10 POKE 56334,PEEK (56334) AND 254**<br>**20 POKE 1, PEEK (1) AND 251**<br>**30 FOR I=0 TO 7**<br>**40 POKE (TEMP + I, PEEK (53248 + I)**<br>**50 NEXT**<br>**60 POKE 1, PEEK (1) OR 4**<br>**70 POKE 56334, PEEK (56334) OR 1**<br>**80 FOR I=TEMP TO TEMP + 7**<br>**90 ? PEEK (I);**<br>**100 NEXT** |
| TEXT OR BIT MAP*<br>**LDA $D018**<br>**AND #$F0**<br>**ORA #$Z**<br>**STA $D018**<br>WHERE Z IS THE HEX EQUIVALENT OF A DECIMAL VALUE IN FIGURE 8-31 ON P. 262<br>*=ONLY BIT 3 IS SIGNIFICANT IN BIT MAP MODE | **LDA $DC0E**<br>**AND #$FE**<br>**STA $DC0E**<br>**LDA $01**<br>**AND #$FB**<br>**STA $01**<br>**LDX #$00**<br>**LOOP LDA $D000,X**<br>**STA $TEMP,X**<br>**INX**<br>**CPX #$07**<br>**BNE LOOP**<br>**LDA $01**<br>**ORA #$01**<br>**STA $01**<br>**LDA $DC0E**<br>**ORA #$01**<br>**STA $DC0E** |

| | LOCATION* | | |
|---|---|---|---|
| X | BITS | DECIMAL | HEX |
| 0 | 0000XXXX | 0 | $0000 |
| 16 | 0001XXXX | 1024 | $0400 (DEFAULT) |
| 32 | 0010XXXX | 2048 | $0800 |
| 48 | 0011XXXX | 3072 | $0C00 |
| 64 | 0100XXXX | 4096 | $1000 |
| 80 | 0101XXXX | 5120 | $1400 |
| 96 | 0110XXXX | 6144 | $1800 |
| 112 | 0111XXXX | 7168 | $1C00 |
| 128 | 1000XXXX | 8192 | $2000 |
| 144 | 1001XXXX | 9216 | $2400 |
| 160 | 1010XXXX | 10240 | $2800 |
| 176 | 1011XXXX | 11264 | $2C00 |
| 192 | 1100XXXX | 12288 | $3000 |
| 208 | 1101XXXX | 13312 | $3400 |
| 224 | 1110XXXX | 14336 | $3800 |
| 240 | 1111XXXX | 15360 | $3C00 |

*Remember that the BANK ADDRESS offset of $4000 for each video bank above zero must be added to the screen memory address.

**Figure 8-29. Screen Memory Locations**

| BANK | ADDRESS RANGE | VALUE OF BITS 1 & 0 IN $DD00 |
|---|---|---|
| | | BINARY DECIMAL |
| 0 | $0–$3FFF | 11 = 3 (DEFAULT) |
| 1 | $4000–$7FFF | 10 = 2 |
| 2 | $8000–$BFFF | 01 = 1 |
| 3 | $C000–$FFFF | 00 = 0 |

**Figure 8–30. Video Bank Memory Ranges**

| | LOCATION OF CHARACTER MEMORY* | | | |
|---|---|---|---|---|
| VALUE OF Z | BITS | DECIMAL | HEX | |
| 0 | XXXX000X | 0 | $0000–$07FF | |
| 2 | XXXX001X | 2048 | $0800–$0FFF | |
| 4 | XXXX010X | 4096 | $1000–$17FF | ROM IMAGE in BANK 0 & 2 (default)* |
| 6 | XXXX011X | 6144 | $1800–$1FFF | ROM IMAGE in BANK 0 & 2* |
| 8 | XXXX100X | 8192 | $2000–$27FF | |
| 10 | XXXX101X | 10240 | $2800–$2FFF | |
| 12 | XXXX110X | 12288 | $3000–$37FF | |
| 14 | XXXX111X | 14336 | $3800–$3FFF | |

Remember to add an offset of $4000 to the start address of character memory for each bank above 0; for bank 3 add 3*$4000 = $C000

* = in C64 mode only.

**Figure 8–31. Character Memory Locations**

| | SCREEN DATA | | COLOR DATA | | CHARACTER DATA | |
|---|---|---|---|---|---|---|
| | TEXT | BIT MAP | TEXT | BIT MAP | TEXT | BIT MAP |
| C128 BASIC (DEFAULTS) | 1024–2023 ($0400–$07E7) | 7168–8167 ($1C00–$1FE7) | 55296–56295 ($D800–$DBE7) | *TAKEN FROM BIT MAP VIDEO MATRIX | 53248–57343 ($D000–$DFFF) | 8192–16383 ($2000–$3FFF) |
| C128 MACHINE LANGUAGE | 1024–2023 ($0400–$07E7) THIS IS ALSO PROGRAMMABLE | 7168–8167 ($1C00–$1FE7) THIS IS ALSO PROGRAMMABLE SEE GRAPHICS SUMMARY | 55296–56295 ($D800–$DBE7) | *TAKEN FROM BIT MAP VIDEO MATRIX | TEXT 53248–57343 ($D000–$DFFF) | 8192–16383 ($2000–$3FFF) THIS IS ALSO PROGRAMMABLE SEE GRAPHICS SUMMARY |
| C64 BASIC (DEFAULTS) | 1024–2023 ($0400–$07E7) THIS IS ALSO PROGRAMMABLE | 1024–2023 ($0400–$07E7) | 55296–56295 ($D800–$DBE7) | *TAKEN FROM BIT MAP VIDEO MATRIX | ROM IMAGE IS AT 4096–8191* ($1000–1$FFF) | NO DEFAULT MUST BE PROGRAMMED SEE GRAPHICS SUMMARY |
| C64 MACHINE LANGUAGE | 1024–2023 ($0400–$07E7) THIS IS ALSO PROGRAMMABLE | 1024–2023 ($0400–$07E7) | 55296–56295 ($D800–$DBE7) | *TAKEN FROM BIT MAP VIDEO MATRIX UPPER NYBBLE = FOREGROUND LOWER NYBBLE = BACKGROUND | ROM IMAGE IS AT 4096–8191* ($1000–$1FFF) * = actual character ROM location = 53248–57343 ($D000–DFFF) | NO DEFAULT MUST BE PROGRAMMED SEE GRAPHICS SUMMARY |

**Figure 8–32. Default Graphics Memory Locations**

**NOTE:** These locations pertain to video bank zero (0) only.

# 9

# SPRITES

# SPRITES: MOVABLE OBJECT BLOCKS

A *sprite* is a movable bit-mapped object that you can define into a particular shape for display on the screen. The sprite image can be as large as 24 pixels wide by 21 pixels tall. Each pixel corresponds to a bit in memory in the sprite storage range; therefore, each sprite requires 63 bytes of storage. The C128 has predefined storage locations for sprite data in the range 3584 ($0E00) through 4095 ($0FFF).

The C128 graphics system has 8 sprites. Each sprite moves on its own independent plane. A sprite may move in front of or behind objects or other sprites on the screen, depending on the specified priority. Standard bit-mapped sprites may be any one of the sixteen available colors. Multi-color sprites may have three colors. The colors that are assigned to the pixels within the sprite depend on the bit patterns of the image. In sprite storage memory, the on bits (1) *enable* the sprite pixels to display the color selected by the sprite color register; the off bits (0) *disable* the corresponding sprite pixels, making them transparent and thus allowing the background color to pass through and be displayed. Sprites also can be expanded to twice the normal size in both vertical and horizontal directions.

Most of the commercially available graphics software packages for the Commodore 128 and C64 rely on sprites. For graphics programming applications, sprites offer superior animation capabilities. Single sprites are useful for small moving objects. However, you can adjoin and overlay several sprites to give greater detail to animated graphic images. For example, suppose you are writing a program that animates a person running on the screen. You can make the image of the person as a single sprite, but the effect looks much more realistic if you allocate separate sprites for different parts of the person's body. The arms can be one sprite, the body another, and the legs a third. Then, you can define two additional sprites: one as a second set of legs in a different position, and the other as a second set of arms in a different position. Position the first set of arms, the body and the first set of legs on the screen so that they are joined into a full body. By continually turning on and off the two different sets of arms and legs, the image appears to be running. This process involves overlaying and adjoining sprites. The explanation given here is a simplified algorithm, and the actual programming can be tricky. Sprite programming has been made easy with the new BASIC 7.0 sprite commands.

The first part of this section explains the new BASIC sprite commands and illustrates the procedure for overlaying and adjoining sprites. The second part explains the internal operations of sprites, including storage information, color assignments, sprite expansion and addressing the sprite registers in machine language.

# BASIC 7.0 SPRITE COMMAND SUMMARY

Here's a brief description of each BASIC 7.0 sprite command:

**COLLISION:** Defines the type of sprite collision on the screen, either sprite to sprite or sprite to data collision
**MOVSPR:** Positions or moves sprites from one screen location to another
**SPRCOLOR:** Defines colors for multi-color sprites
**SPRDEF:** Enters sprite definition mode to edit sprites
**SPRITE:** Enables, colors, sets sprite screen priorities, and expands a sprite
**SPRSAV:** Stores a text string variable into a sprite storage area and vice versa or copies data from one sprite to another
**SSHAPE:** Stores the image of a portion of the bit-map screen into a text-string variable

# BASIC 7.0 SPRITE COMMAND FORMATS

## COLLISION

Define sprite collision priorities
where:

**COLLISION type [,statement]**

| | |
|---|---|
| **type** | Type of collision, as follows:<br>1 = Sprite-to-sprite collision<br>2 = Sprite-to-display data collision<br>3 = Light pen (40 columns only) |
| **statement** | BASIC line number of a subroutine |

**EXAMPLE:**

| | |
|---|---|
| COLLISION 1,5000 | Detects a sprite-to-sprite collision and program control sent to subroutine at line 5000. |
| COLLISION 1 | Stops interrupt action which was initiated in above example. |
| COLLISION 2,1000 | Detects sprite-to-data collision and program control directed to subroutine in line 1000. |

## MOVSPR

Position or move sprite on the screen (using any of the following four formats):

| | |
|---|---|
| 1. **MOVSPR number ,X,Y** | Place the specified sprite at absolute sprite coordinate X,Y. |

| | | |
|---|---|---|
| 2. | **MOVSPR number, + X, + Y** | Move sprite relative to its current position. |
| 3. | **MOVSPR number, X;Y** | Move sprite distance x at angle y relative to its current position. |
| 4. | **MOVSPR number, angle # speed** | Move sprite at an angle relative to its original coordinates, in the clockwise direction and at the specified speed. |

where:
**number** is sprite's number (1 through 8)
**X,Y>** is coordinate of the sprite location.

**ANGLE** is the angle (0–360) of motion in the clockwise direction relative to the sprite's original coordinate.

**SPEED** is the speed (0–15) at which the sprite moves.

This statement positions a sprite at a specific location on the screen according to the SPRITE coordinate plane (not the bit map plane). MOVSPR also initiates sprite motion at a specified rate. This chapter contains a diagram of the sprite coordinate plane.

### EXAMPLES:

| | |
|---|---|
| MOVSPR 1, 150, 150 | Position sprite 1 at coordinate 150,150. |
| MOVSPR 1, +20, +30 | Move sprite 1 to the right 20 (X) coordinates and down 30 (Y) coordinates. |
| MOVSPR 4, 50; 100 | Move sprite 4 by 50 coordinates at a 100 degree angle. |
| MOVSPR 5, 45 #15 | Move sprite 5 at a 45 degree angle in the clockwise direction, relative to its original x and y coordinates. The sprite moves at the fastest rate (15). |

**NOTE:** Once you specify an angle and a speed in the fourth form of the MOVSPR statement, the sprite continues on its path (even if the sprite is disabled) after the program stops, until you set the speed to zero (0) or press RUN/STOP and RESTORE.

## SPRCOLOR

Set multi-color 1 and/or multi-color 2 colors for all sprites

**SPRCOLOR [smcr–1] [,smcr–2]**

where:

**smcr–1** Sets multi-color 1 for all sprites.
**smcr–2** Sets multi-color 2 for all sprites.

Either of these parameters may be any color from 1 through 16.

### EXAMPLES:

SPRCOLOR 3,7 Sets sprite multi-color 1 to red and multi-color 2 to blue.

SPRCOLOR 1,2 Sets sprite multi-color 1 to black and multi-color 2 to white.

## SPRDEF

Enter the SPRite DEFinition mode to create and edit sprite images.

**SPRDEF**

The SPRDEF command defines sprites interactively.

Entering the SPRDEF command displays a sprite work area on the screen which is 24 characters wide by 21 characters tall. Each character position in the grid corresponds to a sprite pixel in the displayed sprite to the right of the work area. Here is a summary of the SPRite DEFinition mode operations and the keys that perform them:

| USER INPUT | DESCRIPTION |
|---|---|
| **1–8 keys** | **Selects a sprite number at the SPRITE NUMBER? prompt only.** |
| **A** | **Turns on and off automatic cursor movement.** |
| **CRSR keys** | **Moves cursor.** |
| **RETURN key** | **Moves cursor to start of next line.** |
| **RETURN key** | **Exits sprite designer mode at the SPRITE NUMBER? prompt only.** |
| **HOME key** | **Moves cursor to top left corner of sprite work area.** |
| **CLR key** | **Erases entire grid.** |
| **1–4 keys** | **Selects color source and enables sprite pixels.** |
| **CTRL key, 1–8** | **Selects sprite foreground color (1–8).** |
| **Commodore key, 1–8** | **Selects sprite foreground color (9–16).** |
| **STOP key** | **Cancels changes and returns to prompt.** |
| **SHIFT RETURN** | **Saves sprite and returns to SPRITE NUMBER? prompt.** |
| **X** | **Expands sprite in X (horizontal) direction.** |
| **Y** | **Expands sprite in Y (vertical) direction.** |
| **M** | **Multi-color sprite.** |
| **C** | **Copies sprite data from one sprite to another.** |

This SPRite DEFinition area is shown in Figure 9–1.

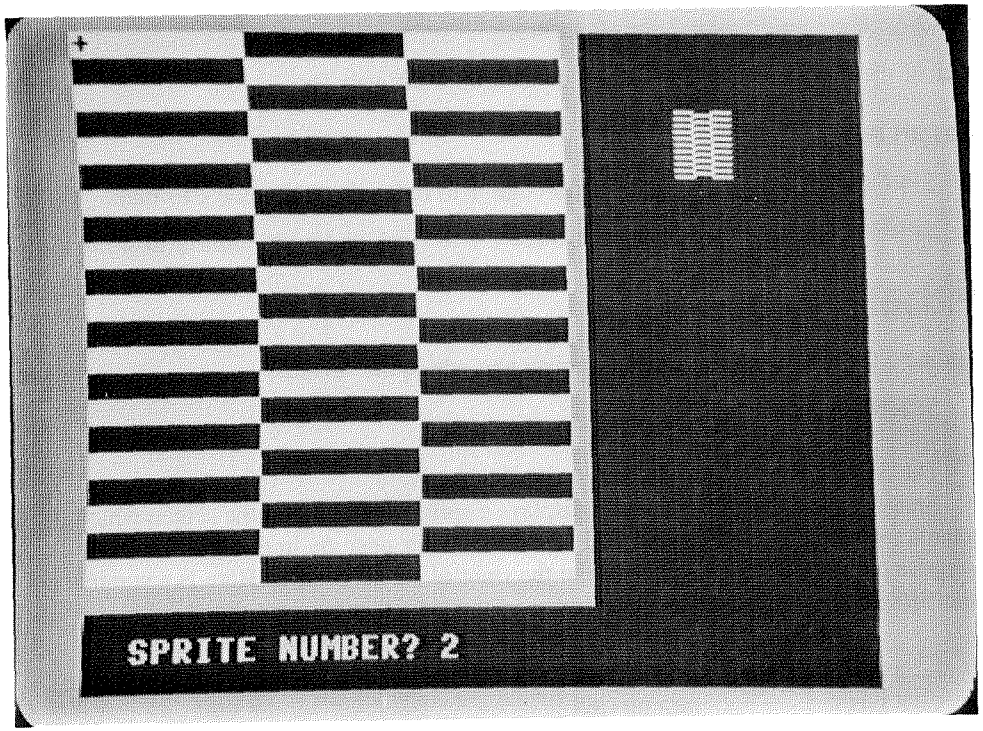



**Figure 9–1. SPRite DEFinition Area**

## SPRITE CREATION PROCEDURE IN SPRITE DEFINITION MODE

Here's the general procedure to create a sprite in SPRite DEFinition mode:

1. Clear the work area by pressing the shift and CLR/HOME keys at the same time.
2. If you want a multi-color sprite, press the M key and the cursor ( + ) appears twice as large as the original one. The double-width cursor appears since multi-color mode actually turns on two pixels for every one in standard sprite mode. Multi-color sprites have only half the horizontal resolution of standard sprites.
3. Select a sprite color. For colors between 1 and 8, hold down the CONTROL key and press a key between 1 and 8. To select color codes between 9 and 16, hold down the Commodore (C=) key and press a key between 1 and 8.
4. Now you are ready to create the shape of your sprite. The numbered keys 1 through 4 fill the sprite and give it shape. For a single-color sprite, use the 2 key to fill a character position within the work area. Press the 1 key to erase what you have drawn with the 2 key. If you want to fill one character

position at a time, press the A key. Now you have to move the cursor manually with the cursor keys. If you want the cursor to move automatically to the right while you hold it down, press the A key again. As you fill in a character position within the work area, you can see the corresponding pixel in the displayed sprite turn on. The sprite image changes as soon as you edit the work area.

In multi-color mode, the 2 key fills two character positions in the work area with the multi-color 1 color, the 3 key fills two character positions with the multi-color 2 color.

You can turn off (color the pixel in the background color) filled areas within the work area with the 1 key. In multi-color mode, the 1 key turns off two character positions at a time.

5. While constructing your sprite, you can move freely in the work area without turning on or off any pixels using the RETURN, HOME and cursor keys.
6. At any time, you may expand your sprite in both the vertical and horizontal directions. To expand vertically, press the Y key. To expand horizontally, press the X key. To return to the normal size sprite display, press the X or Y key again.

   When a key turns on AND off the same control, it is referred to as toggling, so the X and Y keys toggle the vertical and horizontal expansion of the sprite.
7. When you are finished creating your sprite and are happy with the way it looks, save it in memory by holding down the SHIFT key and pressing the RETURN key. The Commodore 128 stores the sprites data in the appropriate sprite storage area. The displayed sprite in the upper right corner of the screen is turned off and control is returned to the SPRITE NUMBER prompt. If you want to create another sprite enter another sprite number and edit the new sprite just as you did with the first one. If you want to display the original sprite in the work area again, enter the original sprite number. If you want to exit SPRite DEFinition mode, simply press RETURN at the SPRITE NUMBER prompt.
8. You can copy one sprite into another with the C= key.
9. If you do not want to SAVE your sprite, press the STOP key. The Commodore 128 turns off the displayed sprite and any changes you made are cancelled. You are returned to the SPRITE NUMBER prompt.
10. To EXIT SPRite DEFinition mode, press the RETURN key while the SPRITE NUMBER prompt is displayed on the screen without a sprite number following it. You can exit under either of the following conditions:

- Immediately after you SAVE your sprite in memory (shift RETURN)
- Immediately after you press the STOP key

Once you have created a sprite and have exited SPRite DEFinition mode, your sprite data is stored in the appropriate sprite storage area in the Commodore 128's

memory. Since you are now back in the control of the BASIC language, you have to turn on your sprite in order to see it on the screen. To turn it on, use the SPRITE command you learned. For example, you created sprite 1 in SPRDEF mode. To turn it on in BASIC, color it blue and expand it in both the X and Y directions and enter this command:

SPRITE 1,1,7,0,1,1,0

Now use the MOVSPR command to move it at a 90-degree angle at a speed of 5, as follows:

MOVSPR 1, 90 # 5

Now you know all about SPRDEF mode. First, create the sprite, save the sprite data and exit from SPRDEF mode to BASIC. Next, turn on your sprite with the SPRITE command. Move it with the MOVSPR command. When you're finished programming, SAVE your sprite data in a binary file with the BSAVE command as follows:

BSAVE "filename", B0, P3584 TO P4096 (This saves all 8 sprites.)

## SPRITE

Turn on and off, color, expand and set screen priorities for a sprite

**SPRITE number> [,on/off][,fngd][,priority] [,x–exp] [,y–exp] [,mode]**

The SPRITE statement controls most of the characteristics of a sprite. The brackets signify optional parameters. If you omit a parameter, you still must include a comma in its place.

| PARAMETER | DESCRIPTION |
|---|---|
| **number** | **Sprite number (1–8)** |
| **on/off** | **Turn sprite on (1) or off (0)** |
| **foreground** | **Sprite foreground color (1–16)** |
| **priority** | **Priority is 0 if sprites appear in front of objects on the screen. Priority is 1 if sprites appear in back of objects on the screen.** |
| **x–exp** | **Horizontal expansion on (1) or off (0)** |
| **y–exp** | **Vertical expansion on (1) or off (0)** |
| **mode** | **Select standard sprite (0) or multi-color sprite (1)** |

Unspecified parameters in subsequent sprite statements take on the characteristics of the previous SPRITE statement. You may check the characteristics of a SPRITE with the RSPRITE function.

### EXAMPLES:

SPRITE 1,1,3 Turn on sprite number 1 and color it red.

| | |
|---|---|
| SPRITE 2,1,7,1,1,1 | Turn on sprite number 2, color it blue, make it pass behind objects on the screen and expand it in the vertical and horizontal directions. |
| SPRITE 6,1,1,0,0,1,1 | Turn on SPRITE number 6, color it black. The first 0 tells the computer to display the sprites in front of objects on the screen. The second 0 and the 1 following it tell the C128 to expand the sprite vertically only. The last 1 specifies multi-color mode. Use the SPRCOLOR command to select the sprite's multi-colors. |

## SPRSAV

Store sprite data from a text string variable into a sprite storage area or vice versa.

**SPRSAV origin>, destination>**

This command copies a sprite image from a string variable to a sprite storage area. It also copies the data from the sprite storage area into a string variable. Either the origin or the destination can be a sprite number or a string variable but both cannot be string variables. If you are copying a string into a sprite, only the first 63 bytes of data are used. The rest are ignored since a sprite can only hold 63 data bytes.

### EXAMPLES:

| | |
|---|---|
| SPRSAV 1,A$ | Copies the bit pattern from sprite 1 to the string variable A$. |
| SPRSAV B$,2 | Copies the data from string variable B$ into sprite 2. |
| SPRSAV 2,3 | Copies the data from sprite 2 to sprite 3. |

## SSHAPE

Save/retrieve shapes to/from string variables

SSHAPE and GSHAPE are used to save and load rectangular areas of multi-color or bit-mapped screens to/from BASIC string variables. The command to save an area of the screen into a string variable is:

**SSHAPE string variable, X1, Y1 [,X2,Y2]**

where:

| | |
|---|---|
| **string variable** | String name to save data in |
| **X1,Y1** | Corner coordinate (0,0 through 319,199) (scaled) |
| **X2,Y2** | Corner coordinate opposite (X1,Y1) (default is the PC) |

Also see the LOCATE command described in Chapter 3 for information on the pixel cursor.

**EXAMPLES:**

| | |
|---|---|
| SSHAPE A$,10,10 | Saves a rectangular area from the coordinates 10,10 to the location of the pixel cursor, into string variable A$. |
| SSHAPE B$,20,30,47,51 | Saves a rectangular area from top left coordinate (20,30) through bottom right coordinate (47,51) into string variable B$. |
| SSHAPE D$, +10, +10 | Saves a rectangular area 10 pixels to the right and 10 pixels down from the current position of the pixel cursor. |

## ADJOINING SPRITES

The following program is an example of adjoining sprites. The program creates an outer space environment. It draws stars, a planet and a spacecraft similar to Apollo. The spacecraft is drawn, then stored into two data strings, A$ and B$. The front of the spaceship, the capsule, is stored in sprite 1. The back half of the spaceship, the retro rocket, is stored in sprite 2. The spacecraft flies slowly across the screen twice. Since it is traveling so slowly and is very far from Earth, it needs to be launched earthward with the retro rockets. After the second trip across the screen, the retro rockets fire and propel the capsule safely toward Earth.

Here's the program listing:

```
5 COLOR 4,1:COLOR 0,1:COLOR 1,2:REM SELECT BLACK BORDER & BKGRND, WHITE FRGRD
10 GRAPHIC 1,1:REM SET HI RES MODE
17 FOR I=1TO40
18 X=INT(RND(1)*320)+1:REM DRAW STARS
19 Y=INT(RND(1)*200)+1:REM DRAW STARS
21 DRAW 1,X,Y:NEXT    :REM DRAW STARS
22 BOX 0,0,5,70,40,,1:REM CLEAR BOX
23 BOX 1,1,5,70,40:REM BOX-IN SPACESHIP
24 COLOR 1,8:CIRCLE 1,190,90,35,25:PAINT 1,190,95:REM DRAW & PAINT PLANET
25 CIRCLE 1,190,90,65,10:CIRCLE 1,190,93,65,10:CIRCLE 1,190,95,65,10:COLOR 0,1
26 DRAW 1,10,17 TO 16,17 TO 32,10 TO 33,20 TO 32,30 TO 16,23 TO 10,23 TO 10,17
28 DRAW 1,19,24 TO 20,21 TO 27,25 TO 26,28:REM BOTTOM WINDOW
35 DRAW 1,20,19 TO 20,17 TO 29,13 TO 30,18 TO 28,23 TO 20,19:REM TOP WINDOW
38 PAINT 1,13,20:REM PAINT SPACESHIP
40 DRAW 1,34,10 TO 36,20 TO 34,30 TO 45,30 TO 46,20 TO 45,10 TO 34,10:REM SP1
42 DRAW 1,45,10 TO 51,12 TO 57,10 TO 57,17 TO 51,15 TO 46,17:REM ENG1
43 DRAW 1,46,22 TO 51,24 TO 57,22 TO 57,29 TO 51,27 TO 45,29:REM ENG2
44 PAINT 1,40,15:PAINT 1,47,12:PAINT 1,47,26:DRAW 0,45,30 TO 46,20 TO 45,10
45 DRAW 0,34,14 TO 44,14 :DRAW 0,34,21 TO 44,21:DRAW 0,34,28 TO 44,28
47 SSHAPE A$,10,10,33,32:REM SAVE SPRITE IN A$
48 SSHAPE B$,34,10,57,32:REM SAVE SPRITE IN B$
50 SPRSAV A$,1:REM SPR1 DATA
55 SPRSAV B$,2:REM SPR2 DATA
60 SPRITE 1,1,3,0,0,0,0:REM SET SPR1 ATTRIBUTES
65 SPRITE 2,1,7,0,0,0,0:REM SET SPR2 ATTRIBUTES
82 MOVSPR 1,150 ,150:REM ORIGINAL POSITION OF SPR1
83 MOVSPR 2,172 ,150:REM ORIGINAL POSITION OF SPR2
85 MOVSPR 1,270 # 5 :REM MOVE SPR1 ACROSS SCREEN
87 MOVSPR 2,270 # 5 :REM MOVE SPR2 ACROSS SCREEN
90 FOR I=1TO 5950:NEXT:REM DELAY
92 MOVSPR 1,150,150:REM POSITION SPR1 FOR RETRO ROCKET LAUNCH
93 MOVSPR 2,174,150:REM POSITION SPR2 FOR RETRO ROCKET LAUNCH
95 MOVSPR 1,270 # 10 :REM SPLIT ROCKET
96 MOVSPR 2, 90 # 5  :REM SPLIT ROCKET
97 FOR I=1TO 1200:NEXT:REM DELAY
98 SPRITE 2,0:REM TURN OFF RETRO ROCKET (SPR2)
99 FOR I=1TO 20500:NEXT:REM DELAY
100 GRAPHIC 0,1:REM RETURN TO TEXT MODE
```

Here's an explanation of the program:

Line 5 COLORs the background black and the foreground white.
Line 10 selects standard bit map mode and clears the bit map screen.
Lines 17 through 21 DRAW the stars.
Line 23 BOXes in a display area for the picture of the spacecraft in the top-left corner of the screen.
Line 24 DRAWs and PAINTs the planet.
Line 25 DRAWs the CIRCLEs around the planet.
Line 26 DRAWs the outline of the capsule portion of the spacecraft.
Line 28 DRAWs the bottom window of the space capsule.
Line 35 DRAWs the top window of the space capsule.
Line 38 PAINTs the space capsule white.
Line 40 DRAWs the outline of the retro rocket portion of the spacecraft.
Line 42 and 43 DRAW the retro rocket engines on the back of the spacecraft.
Line 44 PAINTs the retro rocket engines and DRAWs an outline of the back of the retro rocket in the background color.
Line 45 DRAWs lines on the retro rocket portion of the spacecraft in the background color. (At this point, you have displayed only pictures on the screen. You have not used any sprite statements, so your rocketship is not yet a sprite.)
Line 47 positions the SSHAPE coordinates above the first half (24 by 21 pixels) of the capsule of the spacecraft and stores it in a data string, A$.
Line 48 positions the SSHAPE coordinates above the second half (24 by 21 pixels) of the spacecraft and stores it in a data string, B$.
Line 50 transfers the data from A$ into sprite 1.
Line 55 transfers the data from B$ into sprite 2.
Line 60 turns on sprite 1 and colors it red.
Line 65 turns on sprite 2 and colors it blue.
Line 82 positions sprite 1 at coordinate 150,150.
Line 83 positions sprite 2, 23 pixels to the right of the starting coordinate of sprite 1.
Lines 82 and 83 actually join the two sprites.
Lines 85 and 87 move the joined sprites across the screen.
Line 90 delays the program. This time, delay is necessary for the sprites to complete the two trips across the screen. If you leave out the delay, the sprites do not have enough time to move across the screen.
Lines 92 and 93 position the sprites in the center of the screen, and prepare the spacecraft to fire the retro rockets.
Line 95 propels sprite 1, the space capsule, forward. The number 10 in line 95 specifies the speed in which the sprite moves. The speed ranges from 0 (stop ) to 15 (fastest).
Line 96 moves the expired retro rocket portion of the spacecraft backward and off the screen.

Line 97 is another time delay so the retro rocket, sprite 2, has time to move off the screen.
Line 98 turns off sprite 2, once it is off the screen.
Line 99 is another delay so the capsule can continue to move across the screen.
Line 100 returns you to text mode.

## SPRITE PROGRAM EXAMPLES

The best way to create sprites is with SPRDEF. The following examples assume you have created your sprites in SPRDEF mode.

The first example sprite program illustrates the use of the SPRITE and MOVSPR commands. It positions all eight sprites so they appear to converge on one screen location, then scatter in all eight directions. Here's the listing:

```
10 REM MOVE SPRITE EXAMPLE
20 FOR I=1 TO 8
30 MOVSPR I,100,100
40 NEXT
50 FOR I=1 TO 8
60 SPRITE I,1,I,1,1,1,0
70 MOVSPR I,I*30 # I
80 NEXT
```

Lines 20 through 40 place all eight sprites at sprite coordinate location 100,100. At this point, the sprites are not yet enabled, but when they are, all eight are on top of one another.

Lines 50 and 60 turn on each of the eight sprites in eight different colors. The first "I" is the sprite number parameter. The first "1" in line 60 signifies the enabling of each sprite. The second "I" specifies the color code for each sprite. The second "1" (the fourth parameter) sets the display priority for all the sprites. A display priority of one tells the C128 to display the sprites behind objects on the screen. A zero display priority enables sprites to pass in front of objects on the text or bit-map screen. The fifth and sixth parameters, both of which are ones (1), expand the sprites' size in both the vertical and horizontal directions to twice their original size. The final parameter in the SPRITE statement selects the graphics display mode for the sprites; either standard bit-map sprites (0) or multi-color bit-map sprites (1). In this example, the sprites are displayed as standard bit-map sprites.

Line 70 moves the sprites on the screen. The first parameter, 1, represents the sprite number. The second parameter, "I*30", defines the angle at which the sprites travel on the screen. The pound sign (#) notation signifies that the sprites move according to a particular angle and speed. The final parameter "I" specifies the speed at which the sprites travel on the screen. In this example, sprite 1 moves at the slowest rate of 1, sprite 2 moves at the next highest speed of 2, while sprite 8 moves the fastest of the eight sprites at speed 8. The highest speed a sprite can move is 15.

Finally, line 80 completes the FOR . . . NEXT structure of the loop.

Notice that the sprites move continuously even after the program has stopped RUNning. The reason for this is that sprites are wedged into the interrupt processing of the C128. To turn off and stop the sprites on the screen, either issue a SPRITE command that turns them off, or press RUN/STOP and RESTORE.

The second sprite program example provides a simplified adjoining sprite algorithm. It moves two adjoined sprites across the screen at a ninety-degree angle, assuming that your sprites already reside in the sprite storage range between 3584 ($0E00) and 4095 ($0FFF). For simplicity, if you don't have any actual sprite images stored in the sprite data area, fill the sprite data area with 255 ($FF) from within the Machine Language Monitor with this command:

```
F 0E00 0FFF FF
```

For now, this command turns on all pixels within each sprite. Now you can see how the adjoining algorithm places and moves sprites 7 and 8 side by side.

Here's the listing:

```
10 REM ADJOINING SPRITE ALGORITHM
20 REM THIS PROGRAM ASSUMES YOUR SPRITES ALREADY EXIST IN SPRITE STORAGE
30 I=1  :REM INITIALIZE DISTANCE I
35 SCNCLR
40 MOVSPR 8,50,100:REM SET ORIG POSITION OF SPRITE 8
50 MOVSPR 7,73,100:REM SET ORIG POSITION OF SPRITE 7 TO ADJOIN SPR 8
60 DO
70 SPRITE 8,1,3:REM ENABLE SPR 8
80 SPRITE 7,1,4:REM ENABLE SPR 7
90 MOVSPR 8,I ;90:REM MOVE SPR 8 I UNITS AT A 90 DEGREE ANGLE
100 MOVSPR 7,I ;90:REM MOVE SPR 8 I UNITS AT A 90 DEGREE ANGLE
110 I=1+1 :REM INCREMENT LOOP
120 LOOP
```

Line 30 initializes the distance variable I to 1.

Line 40 positions sprite 8 at absolute coordinates 50,100. Since this program moves two adjoining sprites from the left to right at a ninety-degree angle, sprite 7, which is attached to sprite 8 must be positioned in such a way that it is touching the right edge of sprite 8. Line 50 places sprite 7 on the exact right edge of sprite 8. Since a sprite is 24 pixels wide (before expansion), to adjoin two sprites together, place the ajoining sprite exactly 24 pixels to the right of the top left corner coordinate position of sprite 8. The position of a sprite is placed on the sprite coordinate plane according to the upper leftmost pixel of the sprite. Since the original position of sprite 8 is 50,100, add 24 (inclusive) to the X (horizontal) coordinate to make them join exactly on the respective edges of both sprites. This is provided your sprites are exactly 24 pixels wide. If you don't fill the entire dimensions of a sprite, you may have to adjust the coordinates so that they meet correctly.

At this point, the sprite coordinates are perfectly adjoined. Line 60 initiates a loop, so that the distance can be updated to enable the sprites' movement across the screen. Lines 70 and 80 enable sprites 8 and 7 and color them red and cyan, respectively.

Lines 90 and 100 move sprite 8 and 7, respectively, at a 90-degree angle according to the distance specified by the variable I. Line 110 updates the distance of 1 each cycle through the loop. Line 120 circulates the loop until the distance variable I is equal to 320.

The third sprite example provides an algorithm to overlay two sprites and move them on the screen on a 45-degree angle. Again, this program assumes your sprite data resides in sprite storage. If your sprite images are not stored there, fill the sprites with data as you did in the last adjoining example.

Here's the listing:

```
10 REM OVERLAY EXAMPLE
20 REM THIS PROGRAM ASSUMES SPRITE DATA RESIDES IN SPRITE STORAGE
30 I=1  :REM INITIALIZE DISTANCE I
35 SCNCLR
40 MOVSPR 8,50,100:REM SET ORIG POSITION OF SPRITE 8
50 MOVSPR 7,50,100:REM SET ORIG POSITION OF SPRITE 7 TO OVERLAY SPR 8
60 DO
70 SPRITE 8,1,3   :REM ENABLE SPR 8
80 MOVSPR 8,I;45  :REM MOVE SPR 8 I UNITS AT A 45 DEGREE ANGLE
90 SPRITE 8,0,3   :REM TURN OFF SPR8
100 SPRITE 7,1,4  :REM ENABLE SPR 7
110 MOVSPR 7,I;45 :REM MOVE SPR 8 I UNITS AT A 45 DEGREE ANGLE
120 SPRITE 7,0,3  :REM TURN OFF SPR 7
140 LOOP
```

As in the last program, line 30 initializes the distance variable I to 1.

Lines 40 and 50 position sprites 8 and 7, respectively, at coordinate 50,100. At this point the sprites are not yet enabled, but when they are, sprite 7 will overlay sprite 8 since the lower sprite number has display priority over the higher sprite number.

Line 60 initiates a DO loop to move the sprites along the sprite coordinate plane.

Line 70 enables sprite 8 and colors it red. Line 80 moves sprite 8 a distance of one coordinate according to the current value of I. Line 90 disables sprite 8.

Lines 100 through 120 perform the same operations for sprite 7 as lines 70 through 90 did for sprite 8: enable, move a single distance coordinate according to I and disable. Line 140 repeats the process.

Since this process is repeated so quickly, it appears as though the two sprites alternate movements. When you create the actual images you will use in your overlay sprite program, the images between which you alternate will be ones that simulate the movement of two images and create one animated image.

Create two sprites that appear to form a single animated image. You may have to perfect the timing of the enabling and disabling of the images to make the animated image appear more smooth. Nonetheless, you have a basis for animating two objects into one single moving object.

Although these program examples are written in BASIC, the algorithms are the same whether you are programming in BASIC or machine language. The next section discusses sprite operations independent of the BASIC language. Since this section explained sprites according to BASIC, the next section elaborates on the inner workings of sprites from a machine level (language) perspective.

# THE INNER WORKINGS OF SPRITES

You have seen how to create, move, color and expand sprites with the BASIC 7.0 sprite commands. This section explains how to control sprites *outside* of the BASIC sprite commands (except SPRDEF). This tells you which VIC registers are affected and the specific bits that must be set or cleared to manipulate the sprite features.

Registers of the VIC chip control all aspects of sprites. The enabling of specific bits in certain VIC registers turns on the features of the eight available sprites. The order in which you turn on these features is critical to sprite animation. Following is a summary of the steps necessary to display, color, move and expand sprites. Next to each step is the VIC chip register or other memory location involved in each element of sprite programming.

| SPRITE PROGRAMMING SEQUENCE | REGISTERS INVOLVED |
|---|---|
| **1. Create the sprite image** | **Sprite Data Storage: 3584–4095 ($0E00–$0FFF) This is also programmable. You must change the sprite pointer values.** |
| **2. Point to the sprite data** | **Sprite Data Pointers: 2040–2047 ($07F8–$07FF), or 8184–8191 ($1FF8–$1FFF) when the bit map screen has been cleared with GRAPHIC 1,1** |
| **3. Enable (turn on) the sprite** | **53269 ($D015) (bits 7–0, depending on sprite number)** |
| **4. Color the sprite** | **Standard 53287–53294 ($D027–$D02E) Multi-color 53276 ($D01C), 53285 ($D025), 53286 ($D026)** |
| **5. Position the sprite** | **53248–53264 ($D000–$D010)** |
| **6. Expand the sprite** | **53271 ($D017) (Y direction), 53277 ($D01D) (X direction)** |
| **7. Define sprite display priorities** | **53275 ($D01B)** |
| **8. Define sprite collision priorities** | **53278 ($D01E), 53279 ($D01F)** |

These registers control most sprite characteristics. Once you learn these programming steps, you will be able to exercise full control over the display and movement of sprites.

## CREATING THE SPRITE IMAGE

An easy way to create sprites on the C128 is through SPRite DEFinition mode (SPRDEF). For an explanation of SPRDEF see the SPRDEF entry in the beginning of this chapter. This section assumes your sprite image is already created, and it resides in the sprite data storage area. Before leaving SPRDEF, remember to press the SHIFT key and RETURN at the same time; this allows SPRDEF to store the sprite data in the sprite data storage area. Press RETURN a second time to exit SPRDEF.

The Commodore 128 has a dedicated portion of memory ranging from decimal address 3584 ($0E00) through 4095 ($0FFF), where sprite data is stored. This portion of

memory takes up 512 bytes. As you know, a sprite is 24 pixels wide by 21 pixels tall. In standard sprites, each pixel corresponds to one bit in memory. If the bit in a sprite is off (equal to 0), the corresponding pixel on the screen is transparent, which allows the background to pass through the sprite. If a bit within a sprite is on (equal to 1), the corresponding pixel on the screen is turned on in the foreground color as determined by the sprite color registers. The combination of zeroes and ones produces the image you see on the screen. Multi-color sprites assign colors differently. See the multi-color sprite section later in this chapter for details.

Since a sprite is 24 by 21 pixels and each pixel is represented by one bit of storage in memory, one sprite uses up 63 bytes of memory. See Figure 9–2 to understand the storage requirements for a sprite's data.

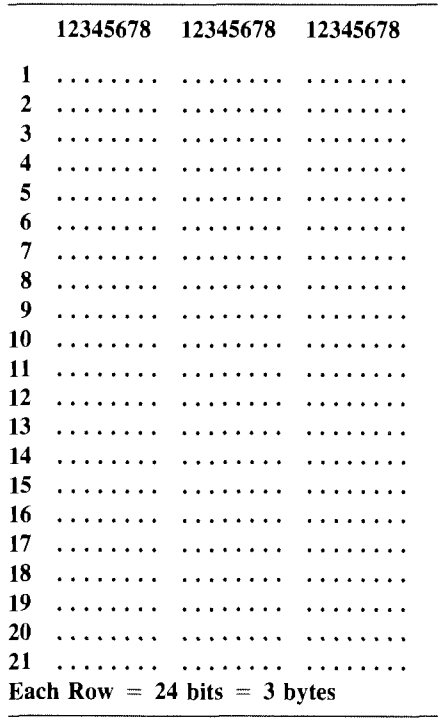



**Figure 9-2. Sprite Data Requirements**

A sprite requires 63 bytes of data. Each sprite block is actually made up of 64 bytes; the extra byte is not used. Since the Commodore 128 has eight sprites and each one consists of a 64-byte sprite block, the computer needs 512 ($8 \times 64$) bytes to represent the data of all eight sprite images.

The area where all eight sprite blocks reside starts at memory location 3584 ($0E00) and ends at location 4095 ($0FFF). Figure 9–3 lists the memory address ranges where each individual sprite stores its data.

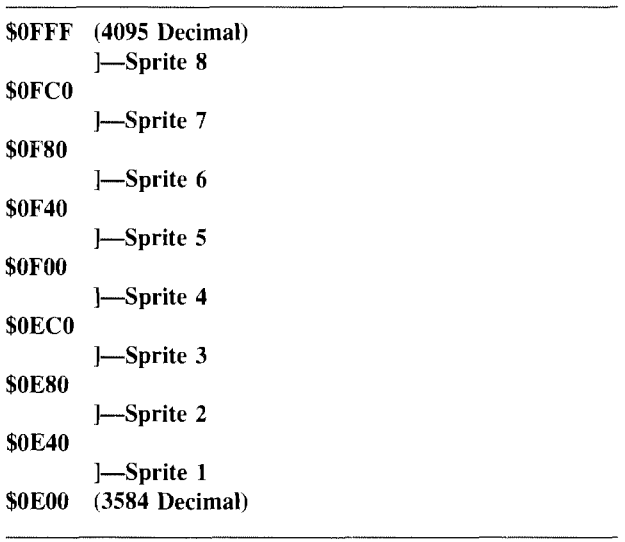



**Figure 9–3. Memory Address Ranges for Sprite Storage**

Keep in mind that sprites are referred to as 1 through 8 in BASIC, but 0 through 7 in machine language.

## SPRITE POINTERS

The VIC chip needs to know where to look for the bit patterns (data) that make up the sprite image in memory. The sprite pointers are used explicitly for this purpose.

Unlike the Commodore 64, the C128 has automatically filled the sprite data pointers with values that direct the VIC chip to point to the data stored in the sprite data range 3584 ($0E00) through 4095 ($0FFF). These sprite data pointers are located at 2040 ($07F8) through 2047 ($07FF) for sprites 0 and 7 respectively. They are also located in the address range 8184 ($1FF8) through 8191 ($1FFF), once the bit map screen is cleared with the GRAPHIC 1,1 command. The default contents of these locations are:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Hexadecimal | 38 | 39 | 3A | 3B | 3C | 3D | 3E | 3F |
| Decimal | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |

To find the actual location where the sprite data pointers are looking for data in memory, multiply the contents of the sprite data pointer by 64 (decimal). By multiplying these values, you'll see that the pointers look for data in the default sprite storage locations in Figure 9–3. See Figure 9–4 for an illustration.

The way the Commodore 128 automatically points to the correct sprite data is convenient for programming, since it eliminates a step (provided the original values of the sprite pointers have not been modified). If you want to store sprite data somewhere else in memory, however, you'll have to change the original value of the sprite pointer (from location 2040 through 2047, or 6184 through 8191) to a value that is equal to:

Start of Sprite Data / 64 = new contents of sprite pointer

| | DATA POINTER CONTENTS** | | START OF SPRITE DATA |
|---|---|---|---|
| **Sprite 0 Data Pointer** | **= 56** | * **64** | **= 3584 ($0E00)** |
| **Sprite 1 Data Pointer** | **= 57** | * **64** | **= 3648 ($0E40)** |
| **Sprite 2 Data Pointer** | **= 58** | * **64** | **= 3712 ($0E80)** |
| **Sprite 3 Data Pointer** | **= 59** | * **64** | **= 3776 ($0EC0)** |
| **Sprite 4 Data Pointer** | **= 60** | * **64** | **= 3840 ($0F00)** |
| **Sprite 5 Data Pointer** | **= 61** | * **64** | **= 3904 ($0F40)** |
| **Sprite 6 Data Pointer** | **= 62** | * **64** | **= 3968 ($0F80)** |
| **Sprite 7 Data Pointer** | **= 63** | * **64** | **= 4032 ($0FC0)** |

** = This applies to video bank 0 only.

**Figure 9-4. Sprite Data Locations**

The start of sprite data is divided by 64 because the data area is allocated in 64-byte sections. For example, you want to place your sprite 0 data in the new location 6144 ($1800). Divide 6144 by 64 to get 96. Place the value 96 ($60) in address 2040 ($078F).

## ENABLING A SPRITE

Once the sprite image has been defined, and the data pointer is pointing to the correct data, you can turn on the sprite. You do this by placing a value in the Sprite Enable Register, location 53269 ($D015). The value placed in this register depends on which sprite(s) you want to turn on. Bits 0 through 7 correspond to sprites 0 through 7. To enable sprite 0, set bit 0. To enable sprite 1, set bit 1 and so on. The value you place in the sprite enable register is equal to two raised to the bit position in decimal.

If you are programming in machine language and want to enable more than one sprite at a time, add the values of two raised to the bit positions together and store the result in the sprite enable register. For example, to enable sprite 5, raise two to the fifth power (32 ($20)) and store it as follows:

```
LDA #$20
STA $D015
```

To enable sprites 5 and 7, raise two to the fifth (32 ($20)) and add it to two to the seventh (128($80)) to obtain the result 160 ($A0):

```
LDA $A0
STA $D015
```

An easier way of perceiving the idea is through binary notation in the Machine Language Monitor as follows:

```
LDA     # % 10100000
STA     $D015
```

To disable the sprite display, clear the bits in the sprite enable register.

# ASSIGNING COLOR TO SPRITES

Sprites have two kinds of color displays: standard bit-map and multi-color bit-map sprites. The color assignments to the pixels within the sprites work in a similar way to standard bit-map and multi-color bit-map modes for the screen.

## STANDARD BIT-MAP SPRITES

Standard bit-map sprites each have their own color register. The lower four bits of each sprite color register determine the sprite color as specified by the sixteen C128 color codes. Figure 9–5 shows the standard bit-map sprite color registers.

| ADDRESS | | DESCRIPTION |
|---|---|---|
| 53287 | ($D027) | SPRITE 0 COLOR REGISTER |
| 53288 | ($D028) | SPRITE 1 COLOR REGISTER |
| 53289 | ($D029) | SPRITE 2 COLOR REGISTER |
| 53290 | ($D02A) | SPRITE 3 COLOR REGISTER |
| 53291 | ($D02B) | SPRITE 4 COLOR REGISTER |
| 53292 | ($D02C) | SPRITE 5 COLOR REGISTER |
| 53293 | ($D02D) | SPRITE 6 COLOR REGISTER |
| 53294 | ($D02E) | SPRITE 7 COLOR REGISTER |

**Figure 9–5. Standard Bit Map Sprite Color Registers**

Figure 9–6 lists the color codes that are placed in the standard bit-map sprite color registers:

| | | | |
|---|---|---|---|
| 0 | Black | 8 | Orange |
| 1 | White | 9 | Brown |
| 2 | Red | 10 | Light Red |
| 3 | Cyan | 11 | Dark Gray |
| 4 | Purple | 12 | Medium Gray |
| 5 | Green | 13 | Light Green |
| 6 | Blue | 14 | Light Blue |
| 7 | Yellow | 15 | Light Gray |

**Figure 9–6. Sprite Color Codes**

In standard bit-map sprites, the data in the sprite block determine how the colors are assigned to the pixels on the screen within the visible sprite. If the bit in the sprite storage block is equal to 1, the corresponding pixel on the screen is assigned the color from the standard sprite color register. If the bit in the sprite data block

is equal to zero, those pixels on the sprite are transparent and the background data from the screen passes through the sprite.

## MULTI-COLOR SPRITES

Multi-color sprites offer a degree of freedom in the use of color, but you trade the higher resolution of standard sprites for the added color. Multi-color sprites are displayed in three colors plus a background color. Multi-color bit-map sprites are assigned colors similar to the way the other multi-color modes work. Before you can assign a sprite multiple colors, you must first enable the multi-color sprite. The sprite multi-color register in location 53276 ($D01C) operates in the same manner as the sprite enable register. Bits 0 through 7 pertain to sprites 0 through 7. To select a multi-color sprite, set the bit number that corresponds to the sprite number. This requires that you raise two to the bit position of the sprite that you want displayed in multi-color. For example, to select sprite 4 as a multi-color sprite, raise two to the fourth power (16) and place it in the multi-color sprite register. In machine language, perform the following instructions:

```
LDA  #$10
STA  $D01C
```

To select more than one multi-color sprite, add the values of two raised to the bit positions together and store the value in the multi-color sprite register.

The VIC chip provides two multi-color registers (0 and 1), in which to place color codes. These are the locations of the sprite multi-color registers:

| | ADDRESS |
|---|---|
| **Sprite Multi-Color Register 0** | **53285 ($D025)** |
| **Sprite Multi-Color Register 1** | **53286 ($D026)** |

**The color codes are those listed in Figure 9–6.**

Like multi-color character mode, the pixels in the multi-color sprites are assigned color according to the bit patterns in the sprite storage block. In this mode, the bits in the sprite block are grouped in pairs. The bit pair determines how the pixels are assigned their individual colors, as follows:

| BIT PAIR | DESCRIPTION |
|---|---|
| **00** | **TRANSPARENT (SCREEN COLOR)** |
| **01** | **SPRITE MULTI-COLOR REGISTER #0 (53285) ($D025)** |
| **10** | **SPRITE COLOR REGISTER** |
| **11** | **SPRITE MULTI-COLOR REGISTER #1 (53286) ($D026)** |

If the bit pair is equal to 00, the pixels are transparent and the background from the screen passes through the sprite. If the bit pattern equals 10 (binary), the color is taken from the sprite color register (locations 53287–53294) of the sprite being defined. Otherwise, the other two bit pair possibilities (01 and 11) are taken from sprite multi-color registers 0 and 1 respectively.

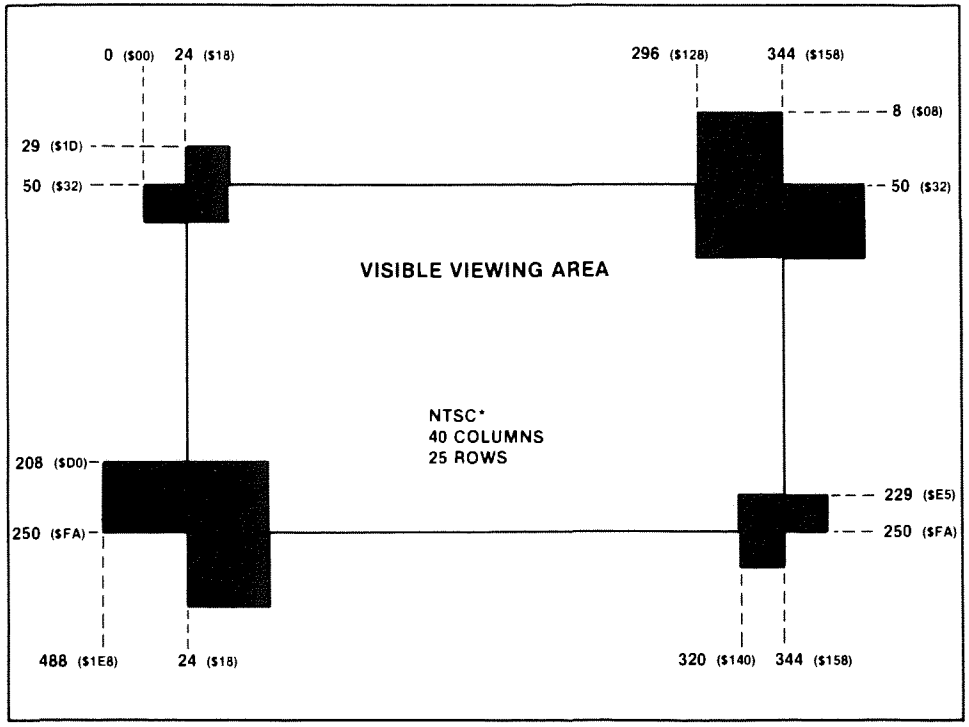
## POSITIONING SPRITES ON THE SCREEN

Each sprite has two position registers to control the sprite's position on the visible screen: horizontal (X coordinate) and vertical (Y coordinate) positions. Figure 9–7 gives the memory locations of the sprite position registers as they appear in the C128 memory.

| LOCATION | | |
|---|---|---|
| DECIMAL | HEX | DESCRIPTION |
| 53248 | ($D000) | SPRITE 0 X POSITION REGISTER |
| 53249 | ($D001) | SPRITE 0 Y POSITION REGISTER |
| 53250 | ($D002) | SPRITE 1 X POSITION REGISTER |
| 53251 | ($D003) | SPRITE 1 Y POSITION REGISTER |
| 53252 | ($D004) | SPRITE 2 X POSITION REGISTER |
| 53253 | ($D005) | SPRITE 2 Y POSITION REGISTER |
| 53254 | ($D006) | SPRITE 3 X POSITION REGISTER |
| 53255 | ($D007) | SPRITE 3 Y POSITION REGISTER |
| 53256 | ($D008) | SPRITE 4 X POSITION REGISTER |
| 53257 | ($D009) | SPRITE 4 Y POSITION REGISTER |
| 53258 | ($D00A) | SPRITE 5 X POSITION REGISTER |
| 53259 | ($D00B) | SPRITE 5 Y POSITION REGISTER |
| 53260 | ($D00C) | SPRITE 6 X POSITION REGISTER |
| 53261 | ($D00D) | SPRITE 6 Y POSITION REGISTER |
| 53262 | ($D00E) | SPRITE 7 X POSITION REGISTER |
| 53263 | ($D00F) | SPRITE 7 Y POSITION REGISTER |
| 53264 | ($D010) | SPRITE X MSB REGISTER |

**Figure 9–7. Memory Location of Sprite Position Register**

The sprite position registers together plot the sprite on a vertical and horizontal coordinate. The position of reference for the calculated vertical and horizontal coordinate is taken from the upper-left corner pixel within the sprite. Whenever you want to place the sprite on a particular screen position, calculate the position using the upper left corner pixel within the sprite. The sprite coordinate plane is not the same as the bit-map coordinate plane. The bit-map coordinate plane starts in the upper-left corner of the screen at coordinate 0,0. The lower right corner of the bit map coordinate plane is point 319,199. The sprite coordinate plane starts at point 24,50 in the top-left corner of the visible screen. The final visible point on the sprite coordinate plane at the bottom-right corner of the screen is 343,249. Figure 9–8 shows how the sprite coordinate plane relates to the visible screen.




*North American television transmission standards for your home TV.

**Figure 9-8. Visible Sprite Coordinates**

After seeing the sprite coordinate plane, you may have noticed something unusual. The vertical coordinate positions have a range of 200. The horizontal coordinate positions have a range of 320 coordinates. Since the C128 is an 8-bit computer, the highest value any register can represent is 255.

How do you position a sprite past the 255th horizontal screen position? The answer is, you have to borrow a bit from another register in order to represent a value greater than 255.

An extra bit is already set aside in the Commodore 128 memory in case you want to move a sprite past the 255th horizontal coordinate. Location 53264 controls sprite movement past position 255. Each of the 8 bits in 53264 controls a sprite. Bit 0 controls sprite 0, bit 1 controls sprite 1 and so on. For example, if bit 7 is set, sprite 7 can move past the 255th horizontal position.

Each time you want a sprite to move across the entire screen, turn on the borrowed bit in location 53264 when the sprite reaches horizontal position 255. Once the sprite moves off the right edge of the screen, turn off the borrowed bit so the sprite can move back onto the left edge of the screen. The following commands allow sprite seven to move past the 255th horizontal position:

```
LDA  $D010
ORA  #$80
STA  $D010
```

The number 128 is the resulting decimal value from setting bit 7. You arrive at this value by raising two to the seventh power. If you want to enable bit 5, raise two to the fifth power, which, of course, is 32. The general rule is to raise two to the power of the sprite number that you want to move past the 255th horizontal screen position. Now you can borrow the extra bit you need to move a sprite all the way across the screen. To allow the sprite to reappear on the left side of the screen, turn off bit seven again, as follows:

```
LDA  $D010
AND  #$7F
STA  $D010
```

Not all of the horizontal (X) and vertical (Y) positions are visible on the screen. Only vertical positions 50 through 249 and horizontal positions 24 through 343 are visible. Location 0,0 is off the screen as is any horizontal location less than 24 and greater than 343. Any vertical location less than 50 and greater than 249 is also off the screen. The off-screen locations are set aside so that an animated image can move smoothly on and off the screen.

## EXPANDING THE SIZE OF SPRITES

The VIC chip offers a feature that allows sprites to be expanded in size, in both the horizontal and vertical directions. When the sprite is expanded, the sprite resolution does not increase, the pixels within the sprite just cover twice as much area; therefore, the sprite is twice as large. Here are the locations in memory for vertical and horizontal sprite expansion:

| | ADDRESS |
|---|---|
| **Vertical (Y) Sprite Expansion Register** | **53271 ($D017)** |
| **Horizontal (X) Sprite Expansion Register** | **53277 ($D01D)** |

These registers operate in the same manner as the sprite enable register. Bits 0 through 7 pertain to sprites 0 through 7. To expand the sprite size in either direction, raise two to the bit position and place it in the expansion register(s). For example, to expand sprite 7 in both directions, perform these machine language instructions:

```
LDA  #$80  (%10000000 = binary notation in the Monitor)
STA  $D017
STA  $D01D
```

To expand more than one sprite, add two raised to the bit position for the sprite numbers you want to expand, and store the result in the expansion registers.

To return the sprites to their original size, clear the bits in the expansion registers.

## SPRITE DISPLAY PRIORITIES

In your sprite programs, you have the option of displaying sprites in front of or behind other sprites or objects on the screen. This is known as defining the sprite's display priorities. The VIC chip defines two distinct sprite display priorities:

1. Sprite-to-sprite
2. Sprite-to-data

### SPRITE-TO-SPRITE DISPLAY PRIORITIES

Each of the eight sprites available on the Commodore 128 is assigned its own plane in which it moves independent of other sprites or of the pictures on the screen background. Visualize the sprite planes as in Figure 9–9.

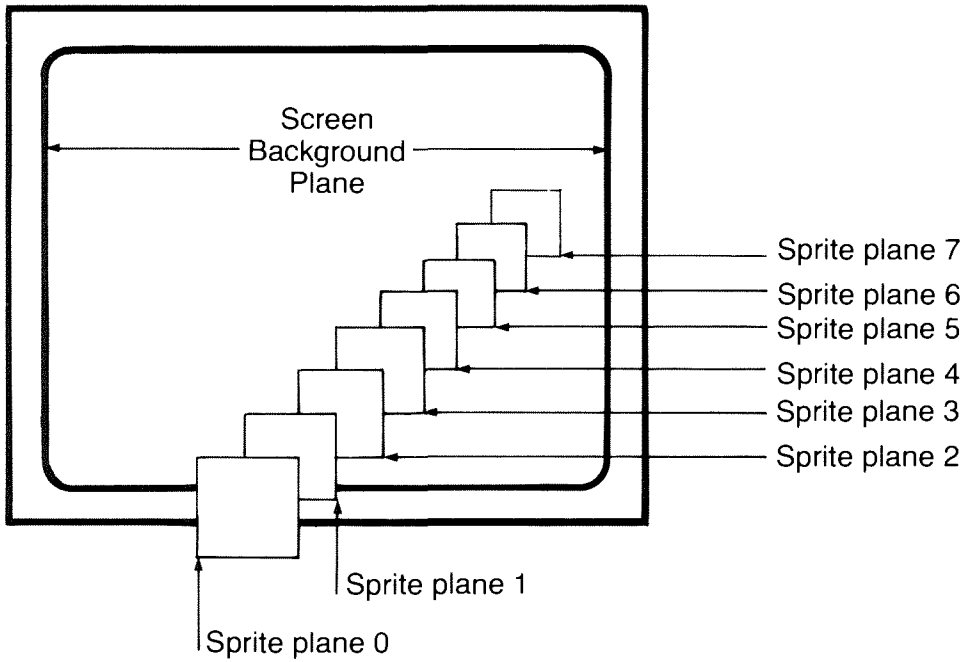



**Figure 9-9. Relationship of Sprite Planes**

The display priority of each sprite plane depends on the sprite number. The sprite-to-sprite display priorities are predefined according to the sprite number. This

feature is preset by the C128 hardware, and is not controlled through a software register. The lower the sprite number, the higher the priority. The higher priority sprite passes in front of the lower priority sprite. Sprite 0 has the highest priority; therefore, it passes in front of any other sprites that may be on the screen should they meet at a particular screen location. For example, sprites 1 and 5 are moving toward a common location on the screen. When the sprites reach the common location, sprite 1 passes in front of sprite 5, since the lower sprite number has the higher display priority. Keep this in mind when you want sprites to intersect paths and pass behind or in front of one another. This is important when overlaying sprites. Assign the sprite you want to pass in front of other sprites to the lower sprite number.

Portions of sprites (pixels) that are not assigned color (the bits in the sprite storage block corresponding to those pixels are equal to zero) are transparent. Holes in these sprites allow background data to pass through the transparent area and create a "window" effect. The same window effect occurs when a higher priority sprite with "holes" passes in front of a lower priority sprite. Even though one sprite has a higher priority, if portions of that higher priority sprite are transparent, the lower priority sprite or the background display data is allowed to pass through the higher priority sprite.

## SPRITE-TO-DATA DISPLAY PRIORITIES

The sprite to data (background screen) priority is selected through a software register within the VIC chip. The Sprite to Background Display Priority Register (location 53275 ($D01B)) specifies whether a sprite passes in front of or behind the objects on the screen background plane. This feature makes sprites seem more realistic. The default value of this register is zero; therefore, all sprites pass in front of objects on the screen background unless you change the bit values in this register. In other words, upon power-up, all sprites have a higher priority than the screen background.

Bits 0 through 7 pertain to sprites 0 through 7. If a bit in this register is set, the sprite passes behind the objects on the screen background. The bit number correspondence to the sprite number works the same way as the sprite enable register and many of the other sprite registers. To set these bits, raise two to the power of the bit position for the sprite number that you want to pass behind the screen background objects. For instance, to cause sprite 6 to pass behind objects on the screen background, raise 2 to the sixth power and place the hexadecimal equivalent in location 53275 ($D01B) in machine language as follows:

```
LDA   #$40   (%01000000 = binary rotation in the Monitor)
STA   $D01B
```

To set the sprite-to-data display priorities for more than one sprite, add the values together for two raised to the respective bit positions and store the result in location 53275 ($D01B).

# SPRITE COLLISION PRIORITIES

The VIC chip has a feature that enables you to detect when a collision occurs between sprites, or between a sprite and screen objects.

## SPRITE-TO-SPRITE COLLISIONS

A sprite-to-sprite collision occurs when an enabled pixel in the foreground of one sprite overlaps an enabled pixel from the foreground of another sprite at any point on the sprite coordinate plane. The collision may also occur at an off-screen coordinate location. Location 53278 ($D01E) flags whether a sprite-to-sprite collision has occurred. This register, like most of the sprite registers, has a bit which detects a collision for each sprite. Bits 0 through 7 pertain to sprites 0 through 7. If a sprite is involved in a sprite-to-sprite collision, the bit corresponding to the sprite involved in the collision is set; therefore, at least two bits are always set in a sprite-to-sprite collision. These bits remain set until they are read at which time the VIC clears the register. You should store the value of this register in a variable until the collision or conditional code depending on the collision is fully processed.

Once a sprite-to-sprite collision is detected, the sprite-to-sprite collision Interrupt Request (IRQ) flag, bit 2 of location 53273 ($D019), is set and an interrupt occurs if enabled in the IRQ Mask Register at 53274 ($D01A). When this occurs, you can incorporate an interrupt routine to be activated upon the collision of two sprites. Therefore, your sprite-to-sprite collision interrupt routine is only executed upon the condition that two sprites collide. This built-in feature gives you a way to conditionally wedge an interrupt (IRQ) routine into your application program, depending on the behavior of the sprites on the screen.

## SPRITE-TO-DATA COLLISIONS

A sprite-to-data collision occurs when an enabled pixel in the foreground of a sprite overlaps a pixel from the foreground of an object on the screen. Location 53279 ($D01F) flags whether a sprite-to-data collision has occurred. This register has a bit which detects a collision for each sprite. Bits 0 through 7 pertain to sprites 0 through 7. If a sprite is involved in a sprite-to-data collision, the bit corresponding to the sprite involved in the collision is set. These bits remain set until they are read at which time the VIC chip clears the entire register. A recommended programming practice is to store the value of this register in a variable until the collision or conditional code depending on the collision is fully processed.

Once a sprite-to-data collision is detected, the sprite-to-data collision IRQ flag in bit 1 of location 53273 ($D019) is set and an interrupt occurs if enabled in the IRQ Mask Register at 53274 ($D01A). When this occurs, you can incorporate an interrupt routine to be activated upon the collision of two sprites. Therefore, your sprite-to-data collision interrupt routine is only executed upon the condition that a sprite has collided with an object on the screen foreground. Again, this gives you a way to conditionally wedge an IRQ routine into your application program depending on the behavior of your animated graphic objects on the screen.

Note that sprite-to-data collisions do not occur with multi-color bit pair 01 (binary). This permits those bits to be interpreted as background display data, without interfering with sprite-to-data collisions.

# 10

# PROGRAMMING THE 80-COLUMN (8563) CHIP

The Commodore 128 computer offers two types of video output: 40-column, composite video through the VIC chip and 80-column, RGBI through the 8563 chip. The 80-column display adds an important feature to the Commodore family of home and business computers: The C128 can be regarded as a business machine. The 8563 chip enables the C128 to display spreadsheets, wordprocessors, database managers and existing CP/M applications in 80 columns. Now, the latest in the family of inexpensive Commodore computers runs the *Perfect* series of business applications, and many other business applications in C128 mode. In CP/M mode, the C128 runs *Wordstar,* and many other popular business applications. In addition, the C128 supports all the hardware and software available for the Commodore 64. The Commodore 128 is truly the complete personal computer.

## THE 8563 VIDEO CHIP FEATURES

The primary purpose of the 8563 video chip is to display characters on the screen. The 8563 has two sets of characters, each with 256 elements. Unlike the VIC chip, however, the 8563 can display all 512 characters simultaneously. The VIC chip displays only one character set at a time.

The 8563 chip supports a limited bit map mode. Bit mapping can be achieved through your own programs, preferably machine language. The BASIC 7.0 graphics commands do not support the 80-column screen. Programming the bit-mapped screen in BASIC is not recommended, since the language is not geared to manipulating single display bits at a time. Later in this chapter, bit mapping the 80-column screen is illustrated in machine language.

Another feature of the 8563 is smooth scrolling in the vertical and horizontal directions. The 8563 chip is equipped with a set of scrolling registers that enable text to be scrolled up, down, left or right. This is discussed later in the chapter.

## PROGRAMMING THE 80-COLUMN (8563) CHIP

Programming the 8563 video chip is a quite different from programming the VIC chip. As you know, the registers of the VIC chip are located in the range 53248 ($D000) through 53296 ($D030) in bank 15. Unlike the VIC chip, the 8563 has only two memory locations in Commodore 128 I/O memory. $D600 and $D601. This means that only two memory locations in the Commodore 128 I/O memory pertain to the 8563 video chip. Internally, the 8563 has thirty-seven internal registers, though they are not addressable in C128 I/O memory. In addition, the 8563 has 16K of RAM of its own that

is independent of the Commodore 128 RAM. You must address locations $D600 and $D601 as the gateways through which you indirectly address the thirty-seven internal registers and 16K of 8563 RAM. You cannot directly access any of the thirty-seven internal registers or the 16K of 8563 RAM.

Location $D600 is the **Address Register**, and $D601 is the **Data Register.** Generally, you place an 8563 register number in the address register ($D600) then either write to or read from the data register in location $D601. This is a simplified explanation—the more-detailed information given in the following sections is needed to program the 8563 successfully.

**8563 NOTES:**

1. You cannot use BASIC PEEK, POKE, or WAIT instructions to access the 8563, because these commands are implemented using indirect operations. Any indirect machine instructions (such as LDA ( ),Y or STA( ),Y) must be avoided because they result in 'false' bus states which are sensed by the 8563 and subsequently acted upon as though they were valid instructions.
2. You should not, directly or indirectly, access the 8563 during interrupts, because there is no way to save and restore the 8563 registers without disrupting any I/O that might have been in progress at the time of the interrupt.

## DETAILS BEHIND PROGRAMMING THE 80-COLUMN CHIP

So far, you have learned that the 8563 chip has:

1. Thirty-seven internal registers.
2. 16K of independent RAM (that is not addressable in the C128 memory map).
3. An address register ($D600 and a data register $D601) in C128 I/O memory.

Figure 10–1 is a summary of the 8563 registers, in the form of a register map:

| REG | BITS 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | HT7 | HT6 | HT5 | HT4 | HT3 | HT2 | HT1 | HT0 | Horizontal Total |
| 1 | HD7 | HD6 | HD5 | HD4 | HD3 | HD2 | HD1 | HD0 | Horizontal Displayed |
| 2 | HP7 | HP6 | HP5 | HP4 | HP3 | HP2 | HP1 | HP0 | Horizontal Sync Position |
| 3 | VW3 | VW2 | VW1 | VW0 | HW3 | HW2 | HW1 | HW0 | Vert/Horiz Sync Width |
| 4 | VT7 | VT6 | VT5 | VT4 | VT3 | VT2 | VT1 | VT0 | Vertical Total |
| 5 | — | — | — | VA4 | VA3 | VA2 | VA1 | VA0 | Vertical Total Adjust |
| 6 | VD7 | VD6 | VD5 | VD4 | VD3 | VD2 | VD1 | VD0 | Vertical Displayed |
| 7 | VP7 | VP6 | VP5 | VP4 | VP3 | VP2 | VP1 | VP0 | Vertical Sync Position |
| 8 | — | — | — | — | — | — | IM1 | IM0 | Interlace Mode |
| 9 | — | — | — | CTV4 | CTV3 | CTV2 | CTV1 | CTV0 | Character Total Vertical |
| 10 | — | CM1 | CM0 | CS4 | CS3 | CS2 | CS1 | CS0 | Cursor Mode, Start Scan |
| 11 | — | — | — | CE4 | CE3 | CE2 | CE1 | CE0 | Cursor End Scan Line |
| 12 | DS15 | DS14 | DS13 | DS12 | DS11 | DS10 | DS9 | DS8 | Display Start Address hi |
| 13 | DS7 | DS6 | DS5 | DS4 | DS3 | DS2 | DS1 | DS0 | Display Start Address lo |
| 14 | CP15 | CP14 | CP13 | CP12 | CP11 | CP10 | CP9 | CP8 | Cursor Position hi |
| 15 | CP7 | CP6 | CP5 | CP4 | CP3 | CP2 | CP1 | CP0 | Cursor Position lo |
| 16 | LPV7 | LPV6 | LPV5 | LPV4 | LPV3 | LPV2 | LPV1 | LPV0 | Light Pen Vertical |
| 17 | LPH7 | LPH6 | LPH5 | LPH4 | LPH3 | LPH2 | LPH1 | LPH0 | Light Pen Horizontal |
| 18 | UA15 | UA14 | UA13 | UA12 | UA11 | UA10 | UA9 | UA8 | Update Address hi |
| 19 | UA7 | UA6 | UA5 | UA4 | UA3 | UA2 | UA1 | UA0 | Update Address lo |
| 20 | AA15 | AA14 | AA13 | AA12 | AA11 | AA10 | AA9 | AA8 | Attribute Start Adr hi |
| 21 | AA7 | AA6 | AA5 | AA4 | AA3 | AA2 | AA1 | AA0 | Attribute Start Adr lo |
| 22 | CTH3 | CTH2 | CTH1 | CTH0 | CDH3 | CDH2 | CDH1 | CDH0 | Character Tot(h), Dsp(V) |
| 23 | — | — | — | CDV4 | CDV3 | CDV2 | CDV1 | CDV0 | Character Dsp(v) |
| 24 | COPY | RVS | CBRATE | VSS4 | VSS3 | VSS2 | VSS1 | VSS0 | Vertical smooth scroll |
| 25 | TEXT | ATR | SEMI | DBL | HSS3 | HSS2 | HSS1 | HSS0 | Horizontal smooth scroll |
| 26 | FG3 | FG2 | FG1 | FG0 | BG3 | BG2 | BG1 | BG0 | Foregnd/Bgnd Color |
| 27 | AI7 | AI6 | AI5 | AI4 | AI3 | AI2 | AI1 | AI0 | Address Increment / Row |
| 28 | CB15 | CB14 | CB13 | RAM | — | — | — | — | Character Base Address |
| 29 | — | — | — | UL4 | UL3 | UL2 | UL1 | UL0 | Underline scan line |
| 30 | WC7 | WC6 | WC5 | WC4 | WC3 | WC2 | WC1 | WC0 | Word Count |
| 31 | DA7 | DA6 | DA5 | DA4 | DA3 | DA2 | DA1 | DA0 | Data |
| 32 | BA15 | BA14 | BA13 | BA12 | BA11 | BA10 | BA9 | BA8 | Block Start Address hi |
| 33 | BA7 | BA6 | BA5 | BA4 | BA3 | BA2 | BA1 | BA0 | Block Start Address lo |
| 34 | DEB7 | DEB6 | DEB5 | DEB4 | DEB3 | DEB2 | DEB1 | DEB0 | Display Enable Begin |
| 35 | DEE7 | DEE6 | DEE5 | DEE4 | DEE3 | DEE2 | DEE1 | DEE0 | Display Enable End |
| 36 | — | — | — | — | DRR3 | DRR2 | DRR1 | DRR0 | DRAM Refresh rate |
| 37 | | HSyNC | VSyNC | — | — | — | — | — | Horiz, Vert Sync Polarity |

**Figure 10–1. 8563 VDC Register Map**

The numbers in the left column are the register numbers. When programming the 8563, the registers are referenced by number only, since they have no actual address in the C128 memory map. Within the register map, columns 7 through 0 apply to the bits within the registers. To the right are the register names, by function. Many of the registers control more than one operation of the chip, so the names only reference the primary purpose of the register. These registers are discussed individually at the end of this chapter. Certain key registers are discussed in the next section. For an explanation of each register, see the register-by-register description in the back of this chapter.

The 16K of 8563 RAM is organized by the Kernal in the following default structure:

| TYPE OF MEMORY | 8563 RAM LOCATION |
|---|---|
| **Character Display area (screen)** | **$0000–$07FF** |
| **Character Attributes (color)** | **$0800–$0FFF** |
| **Character Definitions** | **$2000–$3FFF** |

The layout can be changed by your application program. Registers 12 and 13 (high byte/low byte) specify the start of the character display area, which is comparable to screen RAM in the VIC chip. Registers 20 and 21 (high byte/low byte) specify the character attribute base address. These registers determine the starting address of display characteristics of a character such as reverse video, underlining, flashing and color. Register 28 determines the character set base address, the start of the character definitions in 8563 RAM.

As you can see, the 8563 display has the same three components as the VIC display: screen memory, color memory, and character definitions. All three lie in the 16K of 8563 RAM and do not show up in the C128 memory map. These components of the 8563 video display are discussed in greater detail in the next section.

The third element of 8563 programming is the address and data registers that are located in locations $D600 and $D601. These locations are the transitions, the doorways in and out of the 8563 chip. You cannot directly reference an 8563 register or RAM location. You must indirectly read or write to $D600 and $D601 in order to communicate between the 8563 chip and the rest of the Commodore 128. Figure 10-2 gives description of each of the mapped registers and the bits within them:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **$D600 →** | **address (write):** | | **—** | **—** | **R5** | **R4** | **R3** | **R2** | **R1** | **R0** |
| | **status (read) :** | | **STATUS** | **LP** | **VBLANK** | **—** | **—** | **VER0** | **VER1** | **VER2** |
| **$D601 →** | **data** | **(r/w) :** | **D7** | **D6** | **D5** | **D4** | **D3** | **D3** | **D2** | **D0** |

**Figure 10–2. Mapped Registers**

In location $D600, bit 7 acts as a status bit. When you require a read or write operation to the 8563 RAM or a register, you must check the value of bit 7 of $D600. While it is 0, no read or write operations may take place.

This is the sequence of steps to perform a read or write operation:

1. Address the registers (by number) or internal RAM location.
2. Check the value of bit 7 until it equals 1 in order for the data to be valid.
3. Once bit 7 becomes high (1), perform the read or write operation.

Later in this chapter, you'll see example programs that illustrate the behavior of these registers.

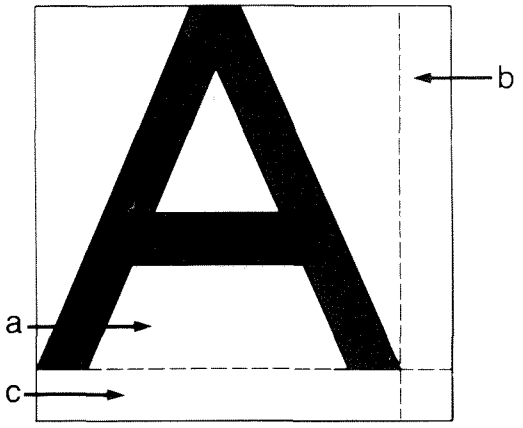
The rest of the bits in $D600 except bit 6, which involves a light pen are used to represent a register number for read or write access. Six bits are necessary since the 8563 has 37 registers; therefore, enough bits are needed to represent numbers up to 37.

Location $D601 is strictly a data register that passes values between the 8563 and the 8502 during read and write operations respectively. Each bit in this register is a data bit. All data that passes between 8563 and the 8502 *must* pass through this register.

# FUNDAMENTAL BACKGROUND ABOUT THE 8563 CHIP

## CHARACTERS

As you know, the 8563 chip is primarily designed for displaying characters within an 80 by 25 character matrix. The visible character is an 8 by 8 pixel matrix. The character can be defined by 16 or 32 bytes and can be displayed less than 8 by 8 pixels.




**a. Visible 8 by 8 pixel character.**
**b. Horizontal intercharacter spacing.**
**c. Vertical intercharacter spacing.**

**Figure 10–3.**

The horizontal unit of measure within a character is a pixel. The vertical unit of measure within a character is called a *scan line*. A scan line is a single horizontal row of scanned pixels referred to as one raster. The pixel and the scan line are the smallest units of measurement by the 8563 chip in the horizontal and the vertical directions, respectively. The size of the displayed character image in both pixels and scan lines is programmable and determined by four registers within the 8563: the Character Total (R9) (vertical), Character Total (R22) (horizontal), Character Displayed (R23) (vertical) and Character Displayed (R22) (horizontal) Registers. These registers are described in the register by register description at the end of this chapter.

## FRAMES

A *frame* is a two-dimensional array (matrix) of characters, also referred to as simply the screen. The default frame is the 80-column by 25-line screen in which characters are displayed. Each character on the frame has a *character pointer* (a Display RAM byte) that tells the 8563 which character to display in each character position within the frame. Think of a frame as a framed picture on a wall containing a two-dimensional array (matrix) of characters. For example, a frame specified to be 16 by 12 characters is filled with characters as in Figure 10-4.

The frame also contains the vertical retrace and all other nonvisible portions of the screen. The frame is programmable in size, similar to a window. The screen frame determines the size of the viewed image, like the frame surrounding a window.

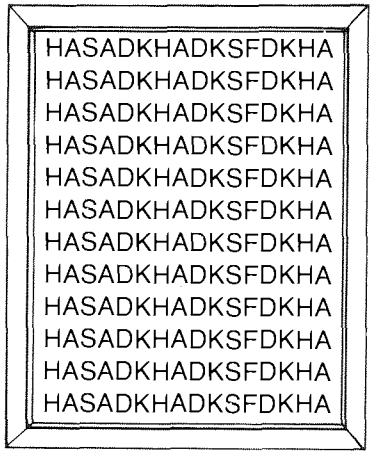



**Figure 10–4.**

The default frame size is 80 columns by 25 lines. However, the frame size and characteristics are programmable, so you can change them. The 8563 has several registers that determine the size and characteristics of the frame. These registers are as follows:

Horizontal Total (R0)
Horizontal Displayed (R1)
Horizontal Sync Position (R2)
Horizontal Sync Width (R3)
Vertical Total (R4)
Vertical Total Adjust (R5)
Vertical Displayed (R6)
Vertical Sync Position (R7)
Vertical Sync Width (R3)
DRAM Refresh/Scan Line (R36)

These registers are covered in the register-by-register description at the end of the chapter.

# UNDERSTANDING THE 80-COLUMN CHARACTER DISPLAY AND ATTRIBUTE MEMORY

As you learned earlier in this chapter, the 8563 has three areas of memory that control the 80-column display: character definitions, character attributes and display screen memory.

In a previous section, the three main elements of programming the 80-column chip were introduced: (1) the address and data registers $D600 and $D601 that map into the C128 RAM memory map, (2) the 16K of independent 8563 RAM, and (3) the 37 internal registers. The 16K of independent 8563 RAM provide the main elements of video display information: display memory, character attributes and character definitions. The display memory is comparable to screen memory in the VIC chip. The character attributes supply color and other information for the characters and parallel the function of VIC color RAM. The 8563 character definitions correspond to the VIC character ROM. Though the VIC and 8563 video memory are similar, the operations of the two are different. This provides a frame of reference to the VIC chip characteristics, so you can understand the comparison between the operation of both the VIC and 8563 video processors.

All these video components are stored in the 16K of independent 8563 RAM. Again, here is the breakdown of the locations where the three sections of video memory are stored in the 8563 RAM:

| TYPE OF MEMORY | 8563 RAM LOCATION |
|---|---|
| **Display Memory** | **$0000–$07FF** |
| **Character Attributes** | **$0800–$0FFF** |
| **Character Definitions** | **$2000–$3FFF** |

All of the 8563 video display memory resides in the 16K section of independent RAM. The 8563 character definitions in RAM require up to 32 bytes per character. If 32 bytes per character are used, the 8563 character definitions use 8K of memory per character set, or a total of 16K. Otherwise, each character set occupies 4K of independent 8563 RAM, or a total of 8K for both character sets. The VIC chip requires a total of 4K of character ROM to represent all the characters, since the VIC chip only requires 8 bytes to represent a character.

The rest of the section describes how the character display area, the character attributes and the character definitions are addressed and interpreted.

## UNDERSTANDING DISPLAY MEMORY

The frame, which was introduced earlier in this chapter, pertains to display memory, the portion of memory you can see on the screen. Display memory resides in the range $0000–$07FF in the 16K independent 8563 RAM. Think of the screen as a frame of 25 rows of 80 characters each. A frame may have different proportions, but for the purposes of this discussion, assume that the frame is 80 by 25.

Each character position (display memory location) within a frame contains a code that acts as a pointer into the character definitions that tells the 8563 which character to display in that particular character position within the frame. For the defined screen of frame size 25 rows by 80 columns, the first 80 pointers define the character codes for the 80-character positions in the top row of the screen (frame). Locations 81 through 160 define which characters are displayed in the second row of the screen (frame), and so on. For the defined screen frame of 80 by 25, 2000 display memory locations are necessary to display the 2000 possible character positions of the screen.

In text mode, the contents of display RAM are screen display codes like the ones the VIC chip uses. These are not ASCII CHR$ codes. Screen codes are actually indexes into the 8563 character RAM definitions, like the screen codes used as indexes into the VIC character ROM. Since the first character in 8563 character RAM is the at-sign character, it is assigned screen code 0. The A is the second character, so it is assigned the screen code 1, and so on. For a complete list of screen codes, see Appendix D.

## ADDRESSING DISPLAY MEMORY

The contents of registers 12 (high-byte) and 13 (low-byte) define the 16-bit start address of the first (top-left) character in the frame (display memory). The default start address of display memory is $0000. The address of the second character position on the frame, that is, the character in column two, row one, is defined by the contents of registers 12 and 13 plus 1. Add an index to the contents of registers 12 and 13 to arrive at the address of subsequent character positions on the frame. For example, to find the address of the character position in the first column of the second row of an 80 by 25 frame, add an offset of 80 to the contents of registers 12 and 13. To find the address of the character position in column 1, row 3, add 160 to the contents of registers 12 and 13.

In general, add an offset of 1 for each character position to the right of the upper-left corner of the frame. To reach the same column in the row below the current character position, add an offset of 80. Although the screen (frame) appears as a two-dimensional matrix of characters, the address references in memory are accessed linearly in sequence. For example, although the character in the top-left corner of the screen is only 24 character rows away from the character in the lower-left corner of the screen, in memory, the locations of those characters are really 1920 (24 × 80) memory locations apart. So a frame, the size of the default screen (80 × 25) requires 2000 memory locations to store the contents of the 2000 character positions on the frame. The contents of each display RAM location (acting as an indexed pointer into the character definitions) tells the 8563 which character, out of the 256 available characters in either character set, to place at any particular position on the 80 by 25 character frame.

## CHARACTER ATTRIBUTES

The character attributes are a set of 8563 RAM locations that assign video enhancement qualities to the characters in the frame. Each character position in the frame is assigned its own attribute byte, so a frame (screen) of 80 columns by 25 lines requires 2000 attribute bytes. The default location of the attribute bytes for the 80 by 25 frame is stored in the range $0800 through $0FCF in the 16K of independent 8563 RAM.

The attributes (and the corresponding bits) that enhance the video qualities of the characters in the frame are as follows:

| BIT | ATTRIBUTE |
|---|---|
| **7** | **Alternate Character Set** |
| **6** | **Reverse Video Attribute** |
| **5** | **Underline Attribute** |
| **4** | **Blink Attribute** |
| **3** | **Red** |
| **2** | **Green** |
| **1** | **Blue** |
| **0** | **Intensity** |

The **ALTERNATE CHARACTER SET** (bit 7), when high (1), enables an additional 256 characters to be displayed within the frame.

The **REVERSE VIDEO ATTRIBUTE** (bit 6), when high (1), reverses the foreground and background colors for the corresponding character in display memory. See also the register-by-register description of the 8563 chip for all the details about register 24, the Reverse Screen Register.

The **UNDERLINE ATTRIBUTE** (bit 5), when high (1), enables the **UNDERLINE SCAN LINE** (as defined by bits 0 through 4 of register 29) in the foreground color for that character within the frame. The underline scan line may be any scan line within the character as determined by the lower four bits of register 29.

When the **BLINK ATTRIBUTE** (bit 4) is equal to 1, the character associated with this attribute byte blinks. The blinking is achieved by repeatedly switching back and forth between the foreground and background colors. The blink rate is determined by bit 5 of register 24.

Attribute bits 3 through 0 determine the foreground color for the **R, G, B** and **I** respectively. The character within the associated display memory appears in the foreground color determined by the R, G, B and I bits. The foreground color of each character cell can be one of 16 colors. The actual colors on the screen can be more than red, green and blue. The RGBI signals coming from each input and the combination of these signals determine the actual screen color. Here are the available colors and associated color codes. The characters' background color is determined by bits 0 through 3 of register 26. These bits in register 26 specify the background color for all the characters on the screen. Note, however, that in machine language these codes are one less than the BASIC codes.

| | | | |
|---|---|---|---|
| **1** | **Black** | **9** | **Dark Purple** |
| **2** | **White** | **10** | **Dark Yellow** |
| **3** | **Dark Red** | **11** | **Light Red** |
| **4** | **Light Cyan** | **12** | **Dark Cyan** |
| **5** | **Light Purple** | **13** | **Medium Gray** |
| **6** | **Dark Green** | **14** | **Light Green** |
| **7** | **Dark Blue** | **15** | **Light Blue** |
| **8** | **Light Yellow** | **16** | **Light Gray** |

**BASIC Color Codes in 80-Column Output**

## ADDRESSING CHARACTER ATTRIBUTES

The contents of registers 20 (high-byte) and 21 (low-byte) define the address of the start of attribute storage. The default start address of attribute storage is $0800. The contents of these two registers form the 16-bit address that marks the start of attribute storage.

Address these attributes in the same way as addressing display memory. To address subsequent character attributes to the right of the character in the **HOME** (upper-left) position of the frame, add an offset of 1 to the contents of registers

20 and 21 for each character attribute to the right of the HOME position. To address the character attribute directly below the HOME position (row 2, column 1), add an offset of 80 to the contents of registers 20 and 21. To address the character attribute of the character in the lower-left corner of the 80 by 25 frame, add a decimal offset of 1920 (24 × 80).

## DISABLING CHARACTER ATTRIBUTES

Many applications programs for the 8563 chip do not need to assign the character attributes. The 8563 has a bit that allows attributes to be disabled, saving a considerable amount of RAM. Bit 6 of register 25 determines whether or not attributes are enabled. If bit 6 is set (1), attributes are enabled: if the value of bit 6 is 0, attributes are disabled. When attributes are disabled, the attribute portion of 8563 RAM may be used for other purposes. If attributes are disabled, only one set of 256 characters is available, and underlined, blinking and reverse video attributes are not available. In addition, all characters are displayed in the same foreground color as determined by bit 4 through 7 of register 26. The background color is still determined by the lower four bits (3–0) of register 26, regardless of whether attributes are enabled or not. The screen border and the vertical and horizontal intercharacter spaces default to the background color.

Display memory may contain more memory locations and attributes than can be displayed at a single time in an 80 by 25 frame. This portion of display memory may be displayed by changing the contents of registers 12 and 13, the Display Start address, and registers 20 and 21, the Attribute Start address, to reflect a new start of display memory and attribute storage. The screen (frame) may be scrolled vertically by incrementing or decrementing the values of the display memory pointer and attribute start addresses. This provides a way to vertically scroll forward or backward through display memory.

In addition to scrolling vertically through display memory, the 8563 has a feature that allows scrolling through display memory in the horizontal direction. Both of these topics are covered later in this chapter.

# CHARACTER DEFINITIONS

The character definitions residing in 8563 RAM specify the character patterns you see on the 80-column screen. The character patterns in memory define the pixel formation of each member of the character set visually represented on the screen. The character definitions can be redefined by your application program.

The default memory range for the character definitions is locations $2000 through $3FFF in 8563 RAM. This range is a total of 8K (8192 bytes). This is enough memory to represent two character sets of 256 characters each. The attribute bit 7 of each attribute byte selects either of the character sets for each character position within a frame.

The 8563 characters are represented by either 16 or 32 bytes, depending on the value of the lower five bits (4–0) of register 9. If the value of the lower five bits of

register 9 is less than 16, each character is defined by 16 bytes in character storage. Therefore, each character set requires a total of 4096 bytes of 8563 RAM. If the value of the lower five bits of register 9 is greater than or equal to 16, each character is defined by 32 bytes of 8563 character RAM. In this case, each character set occupies 8192 bytes (8K) of 8563 character RAM memory and the defined characters are displayed using more than 16 scan lines each. This occupies the full 16K of 8563 RAM and leaves no space for attributes. These taller characters are rarely used, however, because the characters are twice as tall as they are wide and seem out of proportion.

The first byte of each character definition in memory represents the top horizontal scan line of the character displayed on the screen. Depending on the value of the lower five bits of register 9, a character will have either 16 or 32 scan lines, each being 8 bits (pixels) wide.

Although the character definitions in 8563 character RAM are defined by either 16 or 32 8-bit bytes stacked one on top of another, the Display Data Bus is only 8 bits wide, which limits the actual image of the character to 8 horizontal pixels. The total width of the character including the horizontal intercharacter spacing may exceed 8 pixels. See the definition of a character in the beginning of the chapter and the register-by-register description at the end of this chapter for details.

## ADDRESSING THE CHARACTER SET PATTERNS

The default location of the character definition start at address $2000 in 8563 RAM, and since the starting location is programmable, you may change it. The starting location of the character set definitions is determined by bits 5 through 7 of register 28. These are the most significant three bits of the 16-bit address that points to the start of the character definitions in 8563 RAM. Bits 4 through 0 of register 9 make up the five least significant bits for the address that points to the start of the 8563 character definitions. However, if the value of bits 4 through 0 in register 9 is greater than or equal to 16, only bits 7 through 5 are significant, since the characters are made up of 32 bytes each and the full 16K of 8563 RAM memory is allocated for the two character sets, 8K for each set.

## READING 8563 (RAM AND) REGISTERS

As you know, the 8563 chip has its own 16K of RAM and its own set of registers. Neither of these appear in the standard Commodore 128 memory map. The only memory locations that apply to the 8563 chip in Commodore 128 memory are the locations

D600 and $D601. These are the 8563 address and data registers, respectively. In order to read or write to 8563 RAM or registers, you must pass data through the address and data registers. This section explains how to read the 8563 registers. The next section illustrates how to write to the 8563 registers and RAM.

## READING AN 8563 REGISTER

Reading an 8563 register requires five distinct programming steps. Although this may be accomplished in BASIC, it is strongly recommended that you program the 8563 in machine language. Briefly, here are the five machine language programming steps, the *algorithm*, to read an 8563 register. A more detailed explanation and a program to read an 8563 register follows.

1. Load (immediate) the X register with the register number that you want to read.
2. Store the contents of the X register (the 8563 register number) into the 8563 address register ($D600).
3. Check the status bit, bit 7 of address register $D600, until it is high (1).
4. If the status bit is low (0), go back and check it again.
5. Otherwise, load the accumulator with the contents of the 8563 data register $D601. Store the contents of the data register into a C128 RAM location (a variable) for future use and print the variable value.

Since many of the 8563 registers are pointers to a 16-bit address, a register pair is required, one register for the high byte and one for the low byte. In this case, simply increment the X register and repeat the five-step process to read the register contents of the low byte. The high-byte register comes first in the register order, therefore the first read is the high byte of a register pair, increment the X register and repeat the process for the low byte. When the 8502 uses a register pair to point to a 16-bit address, the high byte is the low byte plus 1.

Here's a program that reads the high- (R20) and low- (R21) byte register pair that acts as a pointer to the start of the attribute storage in the 8563 RAM. The value stored in locations $FB and $FC return the default start (high/low byte) of the attribute storage. This pointer can be relocated to point to a section of 8563 RAM in which you define, as is the case for display RAM and the character patterns.

### EXAMPLE:

Reading a register (attribute base address)

```
10 SYS DEC("1800")
20 A$=HEX$(PEEK(251))
30 B$=HEX$(PEEK(252))
40 C$=RIGHT$(A$,2)
50 D$=RIGHT$(B$,2)
60 E$=C$+D$
70 F$="$"
80 G$=F$+E$
90 PRINT "START ADDRESS = ";G$

READY.

MONITOR
    PC  SR AC XR YR SP
; FB000 00 00 00 00 F8

. 01800  A2 14    LDX #$14
. 01802  20 0F 18 JSR $180F
. 01805  85 FB    STA $FB
. 01807  E8       INX
. 01808  20 0F 18 JSR $180F
. 0180B  85 FC    STA $FC
. 0180D  60       RTS
. 0180E  00       BRK
. 0180F  8E 00 D6 STX $D600
. 01812  2C 00 D6 BIT $D600
. 01815  10 FB    BPL $1812
. 01817  AD 01 D6 LDA $D601
. 0181A  60       RTS
```

Here are the details behind each instruction.

- The first instruction ($1800) loads the X register with register number 20 ($14), the high byte of the attribute start address.
- The second instruction jumps to the subroutine at $180F.
- The first instruction of the subroutine stores the contents of the X register, ($14), into the 8563 address register (location $D600 in C128 I/O memory).
- The second instruction of the subroutine (BIT $D600) tests the **UPDATE READY** status bit (7) of the 8563 address register. This is necessary since a read or write operation does not take place as soon as those instructions are issued. The Update Ready bit (bit 7) of the 8563 Address register determines when the read (or write) instruction can be executed. The read operation is pending according to the value of bit 7 of the address register. While the read operation is pending (in prelatch), the value of bit 7 is 0. This indicates that the read (or write) operation data placed in register 31, is not yet valid. Only when the value of bit 7 in the address register is equal to 1, is the data in the data register ($D601) valid.

The behavior of the 8563 chip and the way data is transferred between the 8563 registers and the address and data registers suggests the use of the BIT instruction. The nature of the 8502 BIT instruction is tailor-made for checking the Update Ready bit status of location $D600. When the BIT instruction is encountered by the microprocessor, the value of bit 7 is transferred into the negative flag bit of the 8502 status register. If the value is equal to 0, the negative condition does not exist. If the value is equal to 1, a negative condition does exist. Depending upon the value of this bit, you can conditionally branch to instructions that carry out an operation according to the value of this flag. That's where the third instruction in the above subroutine ($1815) comes into play.

- Instruction 3 (BPL $1812) is a conditional instruction that branches depending upon the value of the negative flag bit in the 8502 status register. In this program, the **BPL (Branch on Result Plus)** instruction branches back to location $1812 and executes the BIT instruction while the negative flag bit is equal to 0. The BIT instruction is again performed, and the value in bit 7 is again transferred to the negative flag bit in the 8502 status register. Then the BPL instruction is encountered in the loop for a second time. If the value of the status bit is still equal to 0, the branch is performed again. The loop continues until the negative status bit is equal to one. Only when the status bit is equal to 1 does the loop fall through to the instruction immediately following the branch instruction.

  Prior to the loop falling through to the instruction after the branch, the Update Ready status bit becomes a 1, it is transferred to the negative flag bit in the 8502 status register, the branch is skipped, and the data can be read (loaded with the fourth instruction of the subroutine ($1817) into $D601. Only after the status bit becomes high (1) is the data in $D601 valid. This is the point where the data is communicated from the 8563 chip to the Commodore 128 memory. Now you can read and act upon the contents of the address within register $14 of the 8563 chip.
- The fourth instruction loads the value stored in the 8563 data register ($D601) into the accumulator.
- The third instruction of the main program stores the value of the data register into location $FB for future use. Subsequent read operations from ($D601) destroy the original value, so it is always a good practice to store the contents of $D601 into a variable or available RAM location.
- The next instruction, INX, increments the contents of the X register, which selects the next register number, $15, so the low byte of the 16-bit address of the start of attribute storage can be read.
- The rest of the program performs the same sequence of steps above for the low byte of the start of attribute storage except for the RTS instruction. The RTS instruction returns program control back to a main program.

The BASIC portion of the program prints the high and low bytes of the 16-bit address that defines the start of attribute storage.

To find the actual address in 8563 RAM of the start of attribute storage, put the high byte and low byte together, and the four-digit hexadecimal number defines the start of attribute storage. Within the monitor, issue this memory command:

```
M FB
```

The contents of $FB and $FC (high/low bytes) mark the start of attribute storage.

# WRITING TO AN 8563 REGISTER

The five-step algorithm to read data from an 8563 register is similar to writing to an 8563 register. In the last section, Step 5 in the read algorithm loaded the accumulator with the value contained in the 8563 data register ($D601). When writing, perform the opposite operation. Instead of loading the accumulator as in a read operation, store the accumulator value into the data register.

Here's an algorithm to write to an 8563 register:

1. Load the X register with the register number in which you want to write.
2. Load the accumulator with a value that you are writing to the 8563 register.
3. Store the contents of the X register into the 8563 address register at $D600.
4. Check bit 7, the status bit, of the Address Register until it is high (1).
5. If the status is low (0), loop back and check it again until it is high.
6. When the status bit (7) is high, store the value contained in the accumulator into the 8563 data register ($D601).

Here's a sample program that writes to register pair R18 and R19, the Update RAM location. Register pair R18 and R19 is the gateway into the independent 8563 RAM. For now we'll illustrate how to write to an 8563 register. Later in this chapter, we'll take it a step farther to explain how to read and write to the 16K of independent 8563 RAM.

```
READY.

MONITOR
    PC  SR AC XR YR SP
; FB000 00 00 00 00 F8

. 01800  A2 12    LDX #$12
. 01802  A9 20    LDA #$20
. 01804  20 1E 18 JSR $181E
. 01807  E8       INX
. 01808  A9 00    LDA #$00
. 0180A  20 1E 18 JSR $181E
. 0180D  EA       NOP
. 0180E  A2 12    LDX #$12
. 01810  20 2C 18 JSR $182C
. 01813  85 FB    STA $FB
. 01815  E8       INX
. 01816  20 2C 18 JSR $182C
. 01819  85 FC    STA $FC
. 0181B  60       RTS
. 0181C  00       BRK
. 0181D  00       BRK
. 0181E  8E 00 D6 STX $D600
. 01821  2C 00 D6 BIT $D600
. 01824  10 FB    BPL $1821
. 01826  8D 01 D6 STA $D601
. 01829  60       RTS
. 0182A  EA       NOP
. 0182B  EA       NOP
. 0182C  8E 00 D6 STX $D600
. 0182F  2C 00 D6 BIT $D600
. 01832  10 FB    BPL $182F
. 01834  AD 01 D6 LDA $D601
. 01837  60       RTS
```

- The first instruction (at $1800) loads the X register with the register number 18 ($12).
- The next instruction loads the accumulator with the high byte of the value you are placing in register 18 ($20).
- Instruction 3 calls the subroutine at location $181E, which is essentially the same routine as in the read register example in the last section, with one exception. The instruction in the subroutine immediately before the RTS is a store instruction to the 8563 data register at $D601. The two instructions before it form a loop that continually checks the Update Ready status bit pending the write operation, just as the read program did. The write operation (STA $D601) is not performed until the status bit 7 in the address register becomes high (1). Once the status bit becomes equal to a 1, the high byte of the address in register 18 is written to the data register ($D601). The subroutine returns control to the main body of the program with the RTS instruction.
- The program resumes with the fourth instruction INX. This increments the register number.
- The fifth instruction (LDA #00) loads the value of the low byte of the address to be written to register 19 ($13) into the accumulator.
- The subroutine at $181E is called again, and the same process is repeated.
- Upon return of the subroutine, the same read routine example from the last section is performed (the subroutine between $182C–$1837) to ensure that the values that were written to registers 18 and 19 are valid. These values are printed by the BASIC routine. To see those values, use the memory command within the *monitor* as follows:

  M FB

  The first two bytes displayed are:

  20 00

This shows that the address in the update location register pair (R18, R19) has been written to successfully and is valid.

# WRITING TO 8563 RAM

Now take the example from the last section one step farther. The last example illustrated how to write to an 8563 register. It just so happens that the register chosen for the example is the update location register pair (R18, R19). This is the register pair that enables you to write to and read from the 16K of independent 8563 RAM. This section expands on the previous example and shows you how to write to the 16K of 8563 RAM.

The program given below fills display memory in 8563 RAM with characters, in the process displaying the characters on the screen. The program fills display RAM located in the default range $0000 through $07CF, with screen code character 102 ($66). See Appendix D for a complete list of screen code characters.

Here's the listing:

```
READY.

MONITOR
     PC  SR AC XR YR SP
; FB000 00 00 00 00 F8

. 01800  A2 12    LDX #$12
. 01802  A9 00    LDA #$00
. 01804  20 21 18 JSR $1821
. 01807  E8       INX
. 01808  20 21 18 JSR $1821
. 0180B  A0 00    LDY #$00
. 0180D  A9 08    LDA #$08
. 0180F  85 FA    STA $FA
. 01811  A9 66    LDA #$66
. 01813  20 1F 18 JSR $181F
. 01816  C8       INY
. 01817  D0 FA    BNE $1813
. 01819  C6 FA    DEC $FA
. 0181B  D0 F6    BNE $1813
. 0181D  00       BRK
. 0181E  00       BRK
. 0181F  A2 1F    LDX #$1F
. 01821  8E 00 D6 STX $D600
. 01824  2C 00 D6 BIT $D600
. 01827  10 FB    BPL $1824
. 01829  8D 01 D6 STA $D601
. 0182C  60       RTS
```

To execute this program, issue the Go (G) command within the Machine Language Monitor. In order for this program to operate, I/O memory must be in context in the currently selected monitor configuration. I/O memory is always in context within configuration 15 ($F), so issue this command to execute the program:

G F1800

The first five instructions (memory $1800–$1808) and the subroutine stored in memory locations $1821 through $182C are the same instructions contained in the program that wrote to registers 18 and 19 in the last section. However, instead of placing the address $2000 in register pair 18 and 19 as in the last section, the address $0000 is placed there to mark the default starting location of display memory in 8563 RAM. This is where the program begins to place screen code 102 in display RAM. The full 2000 bytes (8 pages of memory minus 48 bytes) of display RAM are filled with this character screen code.

Here's how the rest of the program works. The instruction stored at location $180B, LDY #$00, initializes the Y register to 0. The next instruction, LDA #$08 loads the accumulator with the value 8. Then, STA $FA, stores the contents of the accumulator, which initializes this location to 8. Location $FA is used later in the program as a counter for the number of pages to fill with screen code 102. The instruction stored at $1811, LDA #$66, loads the accumulator with the hexadecimal value ($66) of decimal screen code 102.

The instruction starting in location $1813, JSR $181F, calls the subroutine located at address $181F. This subroutine is the crux of the program and actually writes the data to the display memory (screen). This is essentially the same routine as the subroutine located between $181E and $1829, the one from the last section. However, there is one key difference. The first instruction of the subroutine stored starting at $181F, LDX #$1F, loads the X register with the register number 31 ($1F). When writing to 8563

RAM, you must address register 31 because it acts as the internal data register of the 8563 chip. All data that will be written to the 8563 RAM must pass through register 31.

The next three instructions:

```
STX $D600
BIT $D600
BPL $1824
```

check the status bit (7) of the address register until the value becomes equal to 1 (high). These are the same three instructions used in the last two sections when reading or writing to a register and in the first subroutine of this program.

Once the status bit of the 8563 address register becomes high, the data residing in the accumulator (in this case) is written to the external 8563 data register at address $D601, as in the instruction stored at location $1829. In addition, the data in the accumulator is written to the address contained in the update register pair (R18, R19). This occurs since you addressed the internal data register 31 by placing the value 31 in the 8563 address register at $D600. Upon the status bit becoming high, all data in register 31 is placed in the current address of 8563 RAM specified by the update register pair (R18, R19). You placed this address in the update register pair in the very beginning of the program.

At this point, the 8563 hardware takes over. It automatically increments the address contained in the update register pair (R18, R19). You as the programmer do not increment the update register pair's contents; the 8563 chip does this for you automatically. You can say that the update register pair is auto-incrementing. In addition, the 8563 reads the data from the incremented address within 8563 RAM, and places the contents in the internal 8563 register 31. This also saves programming effort. Remember, both of these tasks are performed by the 8563 hardware, not the programmer.

Since the update register pair automatically increments the 8563 RAM addresses, all you have to do is continually call the subroutine that writes the data to each successive display RAM location. That's what the instructions between addresses $1813 and $181B do in the program. These instructions call the subroutine for eight pages of display RAM, or a total of 2048 times. The write subroutine writes only one character to the screen per call, so to fill the entire screen, it must be called 2000 (80 × 25) times.

This sample BASIC program shows how to properly read and write the 8563 registers, and read, modify, write any location in 8563 RAM. The two ML subroutines are the key- and they can be used from either BASIC as shown here, or from your ML program.

```
100 VI=DEC("1800"):VO=DEC("180C"):BANK15
110 READ A$
115 DO UNTIL A$="END"
116 :  POKE VI+I,DEC(A$)                                  (install ML routines)
117 :  I=I+1
118 :  READ A$
119 :  LOOP
120 :  DATA 8E,00,D6,2C,00,D6,10,FB,AD,01,D6,60
130 :  DATA 8E,00,D6,2C,00,D6,10,FB,8D,01,D6,60,END
140 DO
150 :  PRINT                                              (update location HIGH)
160 :  INPUT"ADDRESS (0-16383)";AD                        (update location LOW)
170 :  SYS VO, AD/256,  18                                (READ memory data)
180 :  SYS VO, ADAND255,19
190 :  SYS VI,,31
200 :  RREG A
210 :  PRINT"THE 8563 MEMORY AT";AD;" IS";A               (update location HIGH)
220 :  INPUT"NEW DATA (0-255) ";A                         (update location LOW)
230 :  SYS VO, AD/256,  18                                (WRITE memory data)
240 :  SYS VO, ADAND255,19
250 :  SYS VO, A       ,31
260 :  LOOP
```

The following disassembly shows the two ML routines used in the previous BASIC program and in the Read register, Write register and Write to RAM examples in the preceding sections. They are relocatable.

```
01800  8E 00 D6  STX $D600
01803  2C 00 D6  BIT $D600
01806  10 FB     BPL $1803          (routine to READ 8563 register in .X)
01808  AD 01 D6  LDA $D601
0180B  60        RTS

0180C  8E 00 D6  STX $D600
0180F  2C 00 D6  BIT $D600
01812  10 FB     BPL $180F          (routine to WRITE .A register in .X)
01814  8D 01 D6  STA $D601
01817  60        RTS
```

## BLOCK WRITE AND BLOCK COPY

The 8563 chip's Block Write and Block Copy features are extensions of the write operation described in the last section. The Block Write feature writes the same data contained in the accumulator into a specified number of successive 8563 memory locations. Here's the procedure:

1. Place the initial address, where writing in RAM begins, into registers 18 (high byte) and 19 (low byte).
2. Next, load the accumulator with the data you want to place in successive 8563 RAM locations.
3. Load the X register with the number 31 which specifies the data register within the 8563.
4. Wait for the Update Status Ready bit to become high (1). So far this is the same procedure as writing to 8563 RAM which was described in the Writing to 8563 RAM section.
5. To select the Block Write feature, clear (0) bit 7 of register 24, using the same procedure you use to write to a register.

6. Finally, place the number of successive memory locations you want to write to 8563 RAM in register 30.

At this point, the 8563 writes the data in register 31 starting from the location determined by registers 18 and 19 through the number of successive memory locations specified by register 30. The contents of registers 18 and 19 are automatically incremented to access the next location to be written. Upon completion of the Block Write, registers 18 and 19 contain the last address plus one of the 8563 RAM locations in which it was written.

Note that one write cycle follows the initial write operation to register 31; therefore, the quantity written to register 30 should be one less than the number of 8563 memory locations in which you want to write.

### BLOCK COPY

The Block Copy feature copies (reads and writes) data from one section of 8563 RAM to another. To copy a block of 8563 RAM:

1. Place the destination address, where copying in RAM begins, into registers 18 (high byte) and 19 (low byte).
2. Wait for the Update Status Ready bit to become high (1).
3. Set (1) bit 7 of register 24, which selects the Block Copy feature. At this point, the 8502 writes the initial source address for the block copy to registers 32 (high byte) and 33 (low byte).
4. Place the number of successive memory locations to copy into register 30.

The 8563 reads the contents of the first source address and writes the data to the first destination address. The 8563 RAM addresses are automatically incremented and copying continues until the number of memory locations specified by register 30 are copied.

## READING FROM 8563 RAM

The process of reading from 8563 RAM is the same as reading from a register, except you read from the data register, register 31.

## PROGRAMMING THE 8563 CURSOR

The 80-column cursor is programmable in its image and movement. First, the cursor image may be programmed in four different ways. Bits 6 and 5 of register 10 control the following four characteristics of the cursor:

| BIT VALUE 6 | 5 | CHARACTERISTIC |
|---|---|---|
| 0 | 0 | **Solid cursor** |
| 0 | 1 | **No visible cursor** |
| 1 | 0 | **Slow cursor blink rate (1/16 the frame rate)** |
| 1 | 1 | **Fast cursor blink rate (1/32 the frame rate)** |

Bits 4 through 0 in register 10 determine the bottom scan line in which the cursor definition begins within the 8 scan line cursor character block. Scan line 0 is at the top of the cursor character block, while scan line 8 is at the bottom.

The interrupt-driven screen editor has an indirect location which controls the cursor programming mode. Location $0A2B in C128 RAM is the shadow register which controls the different ways to program the 8563 cursor.

Bits 4 through 0 in register 11 specify the top scan line in which the cursor definition ends within the 8 scan line cursor character block. Bits 4 through 0 in registers 10 and 11 specify the height of the cursor. These registers enable you to program the cursor as a solid block cursor (by default) or as an underlined cursor.

### CURSOR POSITIONING

Register 14 (high byte) and 15 (low byte) determine the position of the cursor within the displayed frame (screen). This register pair indicates the address of the cursor within 8563 display memory. The address of the cursor can be one that is not in the display area of the current frame. If this address is within the address range of the current frame (screen), the character position where the cursor is displayed appears in flashing reverse video, as do most cursors.

The interrupt driven screen editor also has an indirect location that controls the cursor column and row positioning. Location $0A35 in C128 RAM is the shadow register for the 8563 cursor column and row selection.

## BIT-MAPPING THE 80-COLUMN DISPLAY

The 8563 chip supports a bit-map mode. The resolution of the 80-column bit-map screen is 640 by 400 pixels. The 8563 addresses the bytes that make up the bit map across the width of the screen, instead of in 8 by 8 character cells as with the VIC chip.

Bit-map mode is selected by setting bit 7 of register 25. The Kernal initializes bit 7 to 0 to select text mode.

The three main components of an 8563 bit-map display, bit-map data, display memory and color, are similar to the VIC chip in some ways but differ in others. For example, the VIC has an 8000-byte RAM bit map, in which each bit is assigned an individual pixel on the visual screen. Since the 8563 bit map is twice the resolution of the VIC screen, it uses twice the memory: 16,000 bytes (a little less than 16K). The one-to-one correspondence between bits in memory and pixels on the screen is similar, but they are addressed differently. The addresses to access those bits are laid out differently in memory.

Figure 8–25 in chapter 8 illustrates how the VIC bit map is stored in memory. The VIC screen stores 8 bytes of the bit-map in the 8 bytes that would normally make up a character in character mode. For instance, the first 8 bytes of the visual bit map are stored in the first (HOME) character position, which defaults to 8192 ($2000) to 8199 ($2007), or the first eight pixels of the first eight raster rows. The character to the immediate right of the HOME position stores the second eight pixels of the first eight visual raster rows. The 200 visible VIC screen raster rows are 320 pixels wide. The 200

visible 8563 raster rows are 640 pixels wide. Both the VIC and 8563 screens have raster rows which are not visible on the screen. This portion is known as the vertical retrace.

The 8563 screen is stored byte by byte across a raster row. The first eight pixels of the first raster row are stored in the first byte of the 8563 bit-map byte, address $0000 in 8563 RAM. The second eight pixels of the first row are stored in the second byte of the bit map ($0001 in 8563 RAM). The third eight pixels are stored in location $0002 and so on. The first raster row is stored in the first eighty memory locations in 8563 RAM, the second raster row is stored in the second eighty (80–159) 8563 RAM addresses and so on. See Figure 10-5 below for an illustration.

| | 8563 CHARACTER COLUMNS | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RASTER ROW | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | ... | 80 |
| 1 | $0000 | $0001 | $0003 | $40004 | $0005 | $0006 | $0007 | $0009 | $000A | $000B | $000C | ... | $004F |
| 2 | $0050 | $0051 | $0052 | $0053 | $0054 | $0055 | $0056 | $0057 | ..... | | | | $009F |
| 3 | $00A0 | ..... | | | | | | | | | | | |
| 4 | $00F0 | ..... | | | | | | | | | | | |
| . | . | | | | | | | | | | | | |
| . | . | | | | | | | | | | | | |
| . | . | | | | | | | | | | | | |
| 25 | $3E30 | ..... | | | | | | | | | | | $3E7F |

**Figure 10–5. The Relationship Between 8563 Bit Map Data and Screen Pixels**

The collection of the 640 by 200 pixels on the screen form the image in this way: bits that are set (1) enable screen pixels in the foreground color and clear (0) bits take on the background color.

## THE BIT-MAP POINTER

The pointer to the start of the 8563 bit map defaults to the start of display memory at 8563 RAM location $0000. The display memory start address is contained in registers 12 (high) byte and 13 (low byte). The contents of this address mark the address of the first eight pixels of the top raster row on the 8563 screen (frame). Add an index to access successive pixels in successive raster rows.

Since the full 640-by-200 bit map uses almost all of the 16K available in 8563 RAM, there is no room available in memory for attributes such as color.

## COLOR ASSIGNMENTS

The pixel color assignments for the 8563 bit map are based on the value of the bit in memory which corresponds to the pixel on the screen. If the value of the bit in memory is equal to zero, the pixel is colored in the background color according to the value of bits 0 through 3 in 8563 register 26. If the value of the bit in memory corresponding to the screen pixel is equal to one, the pixel is assigned the foreground color determined by bits 4 through 7 in register 26.

The full-sized 640-by-200 pixel bit map bit takes up almost the full 16K of 8563 RAM. You may wonder how you can use the attributes for the corresponding screen locations.

You have two choices: either disable the attributes or reduce the size of your bit map by 2000 bytes, then define and enable the attributes. With a full-size bit map, you cannot implement the attributes, because there is simply not enough independent 8563 RAM to do both. If attributes are disabled, the 8563 bit-map screen uses only two colors, as specified by the background and foreground color codes in register 26.

If you decide to reduce the size of the 8563 bit map and you want to implement the attributes, set bit 6 of register 25 to enable attributes. Then place the high-byte and low-byte addresses in registers 20 and 21, respectively, to define the start of attribute storage. See the Character Attributes section for more information on attributes.

### THE BIT-MAP FRAME SIZE

The size of the bit-map frame defaults to the 640 by 200 raster row dimensions. This frame (screen) size may be reduced. See the Register-by-Register Description for details on altering the size of the visual bit-map image.

The next section provides an example of a bit-map program in machine language containing a program listing and details about the important routines.

## 8563 BIT MAP PROGRAM

As you know, the 8563 supports bit map mode; however, the BASIC graphics commands are not displayed on the 8563 screen. This section contains a program which transfers the 40-column VIC screen to the 8563 display screen. The screen is still 40 columns wide, though it provides the vehicle with which to understand how the 8563 bit map screen is addressed. For easier programming, more graphics features and general functional design use the VIC chip when bit mapping high resolution graphics. The VIC chip is principally designed for these purposes. The 8563 is primarily designed for the 80-character column display capability.

Here's the program listing as it appears in the Machine Language Monitor. Enter the Monitor and start entering the program at address $1800.

```
. 01800  20 B3 18 JSR $18B3
. 01803  A2 19    LDX #$19
. 01805  A9 80    LDA #$80
. 01807  20 77 18 JSR $1877
. 0180A  A9 20    LDA #$20
. 0180C  85 FB    STA $FB
. 0180E  A9 00    LDA #$00
. 01810  85 FA    STA $FA
. 01812  A2 12    LDX #$12
. 01814  20 77 18 JSR $1877
. 01817  E8       INX
. 01818  20 77 18 JSR $1877
. 0181B  A9 00    LDA #$00
. 0181D  85 B1    STA $B1
. 0181F  A9 19    LDA #$19
. 01821  85 9B    STA $9B
. 01823  A9 07    LDA #$07
. 01825  85 9C    STA $9C
. 01827  A9 27    LDA #$27
```

```
. 01829  85 FE    STA $FE
. 0182B  EA       NOP
. 0182C  EA       NOP
. 0182D  A2 00    LDX #$00
. 0182F  A1 FA    LDA ($FA,X)
. 01831  20 75 18 JSR $1875
. 01834  20 83 18 JSR $1883
. 01837  EA       NOP
. 01838  EA       NOP
. 01839  EA       NOP
. 0183A  C6 FE    DEC $FE
. 0183C  D0 EF    BNE $182D
. 0183E  A5 B1    LDA $B1
. 01840  D0 1C    BNE $185E
. 01842  EA       NOP
. 01843  EA       NOP
. 01844  EA       NOP
. 01845  A2 00    LDX #$00
. 01847  A1 FA    LDA ($FA,X)
. 01849  20 75 18 JSR $1875
. 0184C  20 8F 18 JSR $188F
. 0184F  20 9B 18 JSR $189B
. 01852  C6 9C    DEC $9C
. 01854  D0 D1    BNE $1827
. 01856  A9 01    LDA #$01
. 01858  85 B1    STA $B1
. 0185A  4C 27 18 JMP $1827
. 0185D  EA       NOP
. 0185E  A9 00    LDA #$00
. 01860  85 B1    STA $B1
. 01862  A2 00    LDX #$00
. 01864  A1 FA    LDA ($FA,X)
. 01866  20 75 18 JSR $1875
. 01869  20 8F 18 JSR $188F
. 0186C  20 A9 18 JSR $18A9
. 0186F  C6 9B    DEC $9B
. 01871  D0 B0    BNE $1823
. 01873  00       BRK
. 01874  00       BRK
. 01875  A2 1F    LDX #$1F
. 01877  8E 00 D6 STX $D600
. 0187A  2C 00 D6 BIT $D600
. 0187D  10 FB    BPL $187A
. 0187F  8D 01 D6 STA $D601
. 01882  60       RTS
. 01883  18       CLC
. 01884  A5 FA    LDA $FA
. 01886  69 08    ADC #$08
. 01888  85 FA    STA $FA
. 0188A  90 02    BCC $188E
. 0188C  E6 FB    INC $FB
. 0188E  60       RTS
. 0188F  A0 28    LDY #$28
. 01891  A9 00    LDA #$00
. 01893  20 75 18 JSR $1875
. 01896  88       DEY
. 01897  D0 F8    BNE $1891
. 01899  60       RTS
. 0189A  EA       NOP
. 0189B  38       SEC
. 0189C  A5 FB    LDA $FB
. 0189E  E9 01    SBC #$01
. 018A0  85 FB    STA $FB
. 018A2  A5 FA    LDA $FA
. 018A4  E9 37    SBC #$37
. 018A6  85 FA    STA $FA
. 018A8  60       RTS
. 018A9  E6 FA    INC $FA
. 018AB  D0 02    BNE $18AF
. 018AD  E6 FB    INC $FB
```

```
. 018AF  60       RTS
. 018B0  EA       NOP
. 018B1  EA       NOP
. 018B2  EA       NOP
. 018B3  A9 00    LDA #$00
. 018B5  85 FC    STA $FC
. 018B7  A9 3F    LDA #$3F
. 018B9  85 FD    STA $FD
. 018BB  A2 19    LDX #$19
. 018BD  A9 80    LDA #$80
. 018BF  20 77 18 JSR $1877
. 018C2  EA       NOP
. 018C3  EA       NOP
. 018C4  A9 00    LDA #$00
. 018C6  A2 12    LDX #$12
. 018C8  20 77 18 JSR $1877
. 018CB  E8       INX
. 018CC  EA       NOP
. 018CD  EA       NOP
. 018CE  20 77 18 JSR $1877
. 018D1  A9 00    LDA #$00
. 018D3  20 75 18 JSR $1875
. 018D6  C6 FC    DEC $FC
. 018D8  D0 F7    BNE $18D1
. 018DA  C6 FD    DEC $FD
. 018DC  D0 F3    BNE $18D1
. 018DE  60       RTS
```

The instructions in this explanation are referenced by the memory addresses that appear in the left column of the machine language monitor output as in the other machine language program explanations.

The first instruction (J$R $18B3) jumps to a subroutine to clear the bit map (8563) screen.

The next sequence of instructions ($1803–$1809) set bit 7 of 8563 register 25 to select bit map mode. This bit defaults to 0 for text display. The subroutine starting at $1877 is the familiar register write routine you have seen earlier in the chapter. Notice that if the subroutine had started at location $1875, it would have written to 8563 RAM instead of an 8563 register. This routine is called to do so later in the program.

The trick to the success of this program is addressing both the VIC screen and the 8563 screen correctly. The second sequence of instructions ($180A–$1816) define the high byte of the 16-bit address for the start of both the VIC and 8563 screens. First the X register is loaded with the value 18 ($12) which selects 8563 register 18, the 8563 RAM address high pointer. The accumulator is loaded with the value $20, which is stored in location $FB, the VIC bit map pointer (high byte). Now the accumulator is cleared to zero and the write register subroutine places the value zero into the 8563 bit map (high byte) pointer.

The instructions $1817 through $181A perform the same operation for the low byte of both the VIC and 8563 bit map pointers.

The instructions between $1820 and $1827 initialize the variables $FC, $FD, $B1 and $9C to zero. The variables $FC and $FD are the low and high byte pointers for the 8563 bit map RAM addresses. The variable $B1 is the column counter for the VIC screen. The variable $9C is the counter that keeps track of how many addresses to skip in 8563 RAM. This is explained later in the program.

Here's where a major difference between the two video chips becomes important—the way the bit map corresponds to the image on the screen. As you know, the VIC screen stores eight rows of 8 pixels each to make up one character cell. Each row of the eight pixels in a character on the screen are stored consecutively in memory. For example, the HOME character position is stored in default locations $2000 through $2007 for the bit map. The next character position (as if the display was standard text) to the right of the HOME Position is stored between $2008 and $2015. So the first character row is stored between $2000 and $2A00 ($2000 plus (8*8) * 40 = 2560). This is represented graphically in the preceding section (Figure 10–5).

On the other hand, the 8563 stores a bit map across raster rows on the screen. Each set of eight pixels of a raster row is stored in a memory location in 8563 RAM starting at $0000. For example, the first raster row is stored in locations $0000 through $0050, one location for each group of eight pixels. The 8563 screen is 640 pixels across; therefore each row needs 80 bytes of storage in 8563 RAM.

Now you see how the two bit maps are addressed differently. To fully understand how this program gets data from the VIC bit map and places it in the correct spot on the 8563 bit map, this algorithm is provided. The actual instructions to perform these steps are added later. Here's the general scheme you need to follow:

1. Start at the beginning of both the VIC and 8563 bit map locations.
2. Get a VIC bit map byte.
3. Store it in the 8563 bit map.
4. Increment the VIC bit map pointer 8 times (the 8563 pointer increments itself after a write).
5. Do steps 2, 3 and 4 thirty-nine times.

At this point, you have just transferred the first raster row of the VIC bit map onto the 8563 bit map.

6. After each raster row is transferred, subtract 311 from the VIC bit map pointer. This is done 7 times. On the eighth time, the VIC bit map pointer is incremented by 1 to get the to the start of the next VIC character cell.
7. Perform 40 blank write operations (write zeros to the 40 blank columns) to get to the beginning of an 8563 raster row. Remember, the 8563 bit map pointer must be relative to the VIC screen bit map pointer.

If the VIC column counter equals 39, increment the 8563 bit map pointer 41 times to place its relative position at the start of a new character row according to the VIC bit map.

8. Now repeat the process 25 times, once for each character row.

In the program, as discussed, the instructions in $180A through $181A start the VIC and 8563 bit maps at locations $2000 and $0000 respectively. All these instructions encompass step 1 of the algorithm.

The instructions that perform step 2 of the algorithm are stored at locations $182D–$1830.

Step 3 stores the VIC bit map byte at the 8563 bit map position and is performed by the instruction $1831 and the subroutine stored between $1875 and $1882. This subroutine writes the value stored in the accumulator into the appropriate address in 8563 RAM as specified by the contents of 8563 registers 18 ($12) and 19 ($13).

The instructions stored at $183A through $183D perform the fourth step of the algorithm.

The stored in $1883 through $188E update the VIC bit map pointer by 8.

Steps 2, 3 and 4 are performed while the pixel row counter in $9C is less than seven. If it is less than seven, the next byte of the VIC bit map is loaded and stored into the 8563 bit map.

Step 6 is contained in the subroutine starting at ($189B). This decrements the VIC bit map pointer by 311 locations.

Step 7 of the algorithm is contained in the subroutine stored in locations $188F through $1899.

Finally, step 8 of the algorithm is performed by the instructions $186F through $1872.

This program allows you to transfer one bit map to the other, but leaves the VIC bit map as 40 columns. Expand on the algorithm in this section and center the bit map on the 80-column screen. Write a routine that proportionally spaces the 40-column bit map and places it on the entire 80-column screen. Also add a routine that clears the other half of the 80-column bit map. Add these routines to any business application, and you'll have a valuable business tool which you can use completely within the 8563 chip that display your character and bit map data.

## SCROLLING THE 8563 SCREEN

The 8563 has a feature that allows the screen to scroll smoothly either vertically (up or down) or horizontally (left or right). The 8563 scrolling feature can be thought of as a small virtual screen window which is displayed on a portion of a full size 80-by-25 character screen. The virtual screen is only visible through the window you define at the specified position of the full-sized screen. Both screens however do exist in the 8563 character RAM storage locations.

## VERTICAL SCROLLING

Scrolling in the vertical direction is easier than scrolling horizontally. The general algorithm to scroll vertically is as follows:

1. Set the default start of the character RAM storage to $2000 in 8563 RAM. This is done through writing to (or using the default contents of) registers 12 ($0C) and 13 ($0D). Now location $2000 in 8563 RAM marks the start of the first character in the upper left corner of display memory (the HOME position).

   At this point, the size of the screen is 80-by-25 and the screen RAM locations are from $2000 through $27CF in 8563 RAM. Therefore, the first 80 bytes store the first 80 characters in the top row of the 8563 screen.
2. Now add 80 ($50) to the low byte of the display RAM Pointer, register 13. Now the second 80 bytes of the display RAM memory are displayed as the first character row. The screen is now defined by locations $2050 through $281F in 8563 memory. This moves the characters up one character row. The "old" first character row has moved off the top of the screen and the "new" character row has moved onto the last character row of the screen from a previously non-visible range in display area. The new character row should be defined in 8563 RAM before it is moved onto the screen as the last character row.
3. Though this moves an entire character row of display RAM up by one row, the movement is sudden, abrupt and not smooth. The 8563 allows smooth scrolling to occur vertically, through the lower four bits of register 24 ($18) in the 8563.

   To scroll smoothly one scan line at a time, write the value 1 to register 24. This moves the entire screen up by one scan line. The first previously visible scan line is now off screen at the top and a new scan line is scrolled from an off screen location from the bottom onto the last visible scan line of the screen.
4. To scroll subsequent scan lines, increment the value of the lower four bits of register 24. By incrementing this value to 2, the entire screen is smoothly scrolled an additional scan line.
5. Continue to increment the value in register 24 until the value equals the value stored in the lower four bits of register 9, the total number of vertical scan lines of a character. Once these values are equal, only the bottom scan line of the first character row is visible at the top of the screen, and all but the last scan line of the newly scrolled data is visible at the bottom of the screen.
6. To continue to scroll, reset the value in the lower four bits of register 24 to zero and add 80 more locations to the contents of the display RAM pointer as you did in step 2.

   If you plan to use attributes, you must increase the value of the attribute start pointer also to keep up with the display RAM pointer.

```
. 01800  A2 0C     LDX #$0C
. 01802  A9 20     LDA #$20
. 01804  85 FB     STA $FB
. 01806  20 60 18  JSR $1860
. 01809  E8        INX
. 0180A  A9 00     LDA #$00
. 0180C  85 FA     STA $FA
. 0180E  20 60 18  JSR $1860
. 01811  EA        NOP
. 01812  A2 18     LDX #$18
. 01814  A9 01     LDA #$01
. 01816  A8        TAY
. 01817  20 60 18  JSR $1860
. 0181A  EA        NOP
. 0181B  C8        INY
. 0181C  98        TYA
. 0181D  C0 08     CPY #$08
. 0181F  D0 F6     BNE $1817
. 01821  EA        NOP
. 01822  EA        NOP
. 01823  A9 00     LDA #$00
. 01825  20 60 18  JSR $1860
. 01828  A0 00     LDY #$00
. 0182A  E6 FA     INC $FA
. 0182C  D0 02     BNE $1830
. 0182E  E6 FB     INC $FB
. 01830  C8        INY
. 01831  C0 50     CPY #$50
. 01833  D0 F5     BNE $182A
. 01835  A2 0C     LDX #$0C
. 01837  A5 FB     LDA $FB
. 01839  20 60 18  JSR $1860
. 0183C  E8        INX
. 0183D  A5 FA     LDA $FA
. 0183F  20 60 18  JSR $1860
. 01842  A9 00     LDA #$00
. 01844  85 FD     STA $FD
. 01846  A9 00     LDA #$00
. 01848  85 FC     STA $FC
. 0184A  E6 FC     INC $FC
. 0184C  D0 FC     BNE $184A
. 0184E  E6 FD     INC $FD
. 01850  A5 FD     LDA $FD
. 01852  C9 A0     CMP #$A0
. 01854  D0 F0     BNE $1846
. 01856  4C 12 18  JMP $1812
. 01859  EA        NOP
. 0185A  EA        NOP
. 0185B  EA        NOP
. 0185C  EA        NOP
. 0185D  EA        NOP
. 0185E  EA        NOP
. 0185F  EA        NOP
. 01860  8E 00 D6  STX $D600
. 01863  2C 00 D6  BIT $D600
. 01866  10 FB     BPL $1863
. 01868  8D 01 D6  STA $D601
. 0186B  60        RTS
```

## HORIZONTAL SCROLLING

Horizontal scrolling is more complex than vertical scrolling. First, the Display RAM memory must be set up to match the "virtual" screen size. If the "virtual" screen is 132 characters wide, then the Display RAM memory must be organized as if 132 characters were to be displayed, even if only 80 horizontal characters will be displayed

in the 8563 "window." This means that to display a window of 25 rows of 80 characters of a virtual screen of width 132 characters, the Display RAM memory must have at least 3300 (132 × 25) memory locations and the same number of attributes.

To display a horizontal "window" of 80 characters, the 8563 must first read the first 80 display memory locations and attributes from the first 132-character row, and then "skip" the next 52 (132–80) and proceed to read the next 80 pointers and attributes from the second 132-character row. The amount of the "skip" must be written to R27. In this case "52" is written to R27 at system initialization.

When horizontal scrolling is not used, a "zero" written to R27 will cause the 8563 to skip no display memory locations or attributes, allowing the "virtual" screen in Display RAM memory to be the same width as the 8563 window.

Now that the 8563 has been set up to view a window 80 characters wide out of a virtual screen of 132 characters wide, horizontal scrolling may occur by modifying the address in R12/R13 (DISPLAY START ADDRESS) and R20/R21 (ATTRIBUTE START ADDRESS). Increasing these addresses by one will scroll the entire screen "left" one character. The first character of each character row will be scrolled "off" the left side of the screen, all characters on the screen will be moved one space to the left, and one extra character will be scrolled onto the right side of the screen. Simple manipulation of the address in R12/R13 and R20/R21 will scroll the window to the left or right.

## SMOOTH HORIZONTAL SCROLLING

The 8563 has the ability to scroll smoothly in the horizontal direction. Horizontal smooth scrolling moves all characters on the screen the desired number of pixels to the "left," up to the width of one character. Before horizontal smooth scrolling can occur, the 8563 must have something to scroll "onto" the right side of the screen, so R27 must be greater than zero. This sets up a virtual screen at least one character greater than the displayed screen.

The screen is scrolled left by one pixel by changing the contents of R25(3–0) to "one" and increasing the contents of R22(3–0) by one. Both registers must be changed to scroll horizontally.

The number in R25(3–0) and R22(3–0) may be increased, increasing the number of pixels scrolled, until R25(3–0) equals R22(7–4). This is the maximum amount of horizontal smooth scrolling allowed. At this point, only one (the right) pixel will exist for the first character of each character row and the last character will have scrolled onto the row with only the one (the right) pixel still not visible. To scroll further, the CPU must change R25(3–0) back to zero and increase R12/R13 (DISPLAY START ADDRESS) and R20/R21 (ATTRIBUTE START ADDRESS) each by one to scroll horizontally by one character. This sequence will have smooth-scrolled left by one character.

# REGISTER-BY-REGISTER DESCRIPTION

This section describes each 8563 register in detail by register number. The significant bits are isolated and explained separately.

The conventions used in the register descriptions are as follows:

Rn (msb–lsb)

The R stands for "register." The letter n represents the register number, ranging from 0 through 36. The msb stands for "most significant bit" in that particular description. Often the most significant bit is assumed to be bit 7; however, in this discussion, bit 7 is not always the most significant.

The notation lsb stands for "least significant bit" in the particular register description. Generally, the least significant bit is assumed to be 0; however, in this description 0 is not always the least significant.

If a single bit number is enclosed in parentheses, then this is the only bit explained for that particular function.

Consult this section as a quick reference. See the main body of the chapter for additional background on programming the 80-column display.

## R0 HORIZONTAL TOTAL

The number of characters, minus 1, between successive horizontal sync pulses. This number includes the displayed part of a character row, the horizontal border and the blanking interval during horizontal sync pulses.

## R1 HORIZONTAL DISPLAYED

The number of characters displayed on each character row. This number sets the width of the displayed part of the character row.

## R2 HORIZONTAL SYNC POSITION

The number of characters from the beginning of the displayed part of a character row to the start of the horizontal sync pulse. This register controls the horizontal position of the text on the CRT screen. This number should be greater than the number in R1 (horizontal displayed).

## R3(3–0) HORIZONTAL SYNC WIDTH

The width of the horizontal sync pulse in characters, plus 1.

## R3(7–4) VERTICAL SYNC WIDTH

The width of the vertical sync pulse in scan lines. This number sets the duration of the vertical sync pulse, which may extend past the end of the current frame into the next frame. In the interlaced sync and video mode, this register should be programmed with a value equal to twice the number of scan lines desired.

## R4 VERTICAL TOTAL

The number of character rows, minus 1, between successive vertical sync pulses. This number includes the displayed rows, vertical border and the blanking interval during the vertical sync pulse and determines the vertical sync rate.

## R5(4–0) VERTICAL TOTAL ADJUST

The number of scan lines added to the end of the frame for fine adjustment of the vertical sync rate. In the interlaced sync and video mode, this register should be programmed with a value equal to twice the number of scan lines desired.

## R6 VERTICAL DISPLAYED

The number of character rows displayed in a frame. This number sets the height of the displayed part of the frame.

## R7 VERTICAL SYNC POSITION

The number of character rows, plus 1, from the first displayed character row to the start of the vertical sync pulse. This number should be greater than the number in R6 (vertical displayed).

## R8(1–0) INTERLACE MODE CONTROL

There are three raster-scan display modes: non-interlaced, interlaced sync, and interlaced sync and video. The non-interlaced mode has been described above. In this mode, each scan line is refreshed at the vertical sync rate. This mode is selected by setting R8(1–0) equal to 00 or 10.

In the interlaced sync mode [R8(1–0) = 01], even and odd fields alternate to generate frames. The same information is displayed in both odd and even fields but the vertical sync timing causes the scan lines in the odd fields to be displaced from those in the even fields by one-half scan line. The spaces between adjacent scan lines are filled, resulting in a higher-quality character on monitors designed to receive interlaced video.

In the interlaced sync and video mode [R8(1–0) = 11] even and odd fields alternate to generate frames. Odd fields display odd scan lines, and even fields display even scan lines. This doubles the vertical character density on the screen. To display twice as many character rows, the user must re-initialize the 8563 registers. In this mode, the vertical sync width [R3(7–4)] and the vertical total adjust [R5(4–0)] should be programmed with even values. The duration of each of these events (in scan lines) will be one half the value programmed into the registers.

## R9(4–0) CHARACTER TOTAL, VERTICAL

The number of scan lines, minus 1, in a character. This number includes the displayed part of a character and the vertical intercharacter spacing below the displayed part. This register sets the vertical dimension of a character and the vertical dimension of a character row.

## R10(4–0) CURSOR START SCAN LINE

The number of the first (top) scan line of the reverse-video cursor. The number 0 refers to the first (top) scan line of the character. This register and register 11 (below) define the cursor height, and can create a variety of effects, from a block cursor to an underline cursor.

## R10(6–5) CURSOR MODE

The cursor can be programmed to four different modes by programming R10(6–5) to a 00 (solid cursor), 01 (no cursor), 10 (cursor blinking at one sixteenth the frame rate), or 11 (cursor blinking at one thirty-second the frame rate).

## R11(4–0) CURSOR END SCAN LINE

The number of the last (bottom) scan line of the reverse-video cursor, plus 1.

## R12, R13 DISPLAY START ADDRESS (HIGH, LOW)

The address of the first (left) 8 pixels of the top scan line of the frame is defined by R12 and R13. The most significant 8 bits are in R12, the least significant in R13. These set the complete 16-bit address for the first 8 pixels. The address of subsequent sets of 8 pixels on that scan line are incremented from the previous address. The bit-mapped pixel data for a scan line is fetched from adjacent memory locations. A frame of 640 horizontal by 200 vertical pixels will use 16,000 bytes of display 8563 RAM memory locations (R12, R13) through (R12, R13 + 15999).

The bit map attributes are a set of Display RAM bytes, each of which corresponds to a character position on the frame. The 8-bit attribute defines additional characteristics of the character area at that position on the frame. These characteristics (attributes) are as follows: Bits 3–0 are foreground R, G, B and I, respectively. Bits 7–4 are background R, G, B and I, respectively. For a frame of i rows of j characters each, the first j attributes will define the attributes for the first (top) character row, and (i × j) attributes will need to be defined in display 8563 RAM. For a bit-mapped frame of 25 characters high and 80 characters wide, 2000 attributes will be needed.

## R12, R13 DISPLAY START ADDRESS (HIGH, LOW)

The address for the pointer of the first (top-left) character of the frame is defined in R12 and R13. The most significant eight bits is in R12, the least significant in R13. These set the complete 16-bit address for that first pointer. The address of pointers of subsequent characters are incremented from the previous address, so horizontally adjacent characters have pointers in adjacent memory locations. A frame of 25 character rows of 80 characters each will use 2000 pointers of display 8563 RAM memory locations (R12/13) through (R12/13 + 1999).

## R14, R15 CURSOR POSITION

The position of the cursor is set by R14 and R15. These two registers contain the 16-bit address of the cursor. R14 is the most significant byte, R15 the least significant. If the address of R14/R15 is the address of a character within the displayed part of the frame, then that character will be displayed in reverse video for those scan lines determined by

R10(4–0) and R11(4–0). These registers allow the 8563 to be programmed for a reverse-video "block" cursor, or for an "underline" cursor. The location of the cursor can be changed by changing the contents of registers R14 and R15.

### R16, R17 LIGHT PEN VERTICAL, HORIZONTAL

The 8563 supports a light pen function. When the LPEN input pin transitions from a low to high, the 8563 latches the current vertical and horizontal character counts in the two light pen registers. The vertical count is latched in R16. LPEN transitions that occur during the row at the top of the frame will latch a 1 in R16. Transitions on subsequent rows will latch 2, 3, etc. The horizontal count is latched in R17. LPEN transitions that occur during the leftmost character column will latch an 8. Transitions on subsequent columns will latch 9, 10, etc. The condition of the light pen registers is latched in bit 6 of the status register. If 0, the light pen registers have not been latched and contain invalid data. If 1, the registers contain valid latched data. A CPU read of R16 or R17 resets the Status Register bit.

## READ AND WRITE DISPLAY 8563 RAM MEMORY

### R18, R19 UPDATE LOCATION
### R31 CPU DATA

The CPU communicates to the 8563 RAM memory via address and data registers in the 8563. For the CPU to read a memory location, it must place that address in registers R18 (most significant byte) and R19 (least significant). The 8563 responds by executing a read of that memory location and placing that data into R31, which the CPU may then read. During the time that the "read" is pending and data is not yet valid in R31, the Update Ready bit of the status register (bit 7) will be 0. Upon completion of the read cycle, data will be valid in R31 and the bit will be 1. When the CPU reads the data in R31, the 8563 increments the address in R18/R19 and performs a read of that address. This allows the CPU to read successive memory locations without repeatedly changing the addresses in R18/R19.

For the CPU to write data to a memory location, the address must be written to R18/R19 as in a "read" described above. Following the automatic read and after the Update Ready bit is a 1, the CPU should write the desired data to R31. The 8563 then writes the data to 8563 Memory at the address defined by R18/R19, increments the address in R18/R19, reads the data from the incremented memory location and places it in R31. The 8563 then sets the Update Ready bit to a 1. At all times, if the Update Ready bit is a 0, then CPU access to 8563 memory is pending. Additional access of R18, R19 or R31 should be avoided until the Update Ready bit is a 1.

### R20, R21 ATTRIBUTE START ADDRESS (HIGH, LOW)

The address for the attribute of the first (top-left) character of the frame is defined in R20 and R21. The most significant 8 bits is in R20, the least significant in R21. These set the complete 16-bit address for that first attribute. The address of attributes of

subsequent characters are incremented from the previous address, so horizontally adjacent characters have attributes in adjacent memory locations. A frame of 25 character rows of 80 characters each will use 2000 attributes at display 8563 RAM memory locations (R20/R21) through (R20/R21 + 1999).

## R22(3–0) CHARACTER DISPLAYED, HORIZONTAL

This number sets the width of the displayed part of the character, and defines the horizontal intercharacter spacing to the right of the displayed part.

If the defined character is to have no horizontal intercharacter spacing, then R22(3–0) should be initialized with a value equal to (the value of R22(7–4) plus 1). This value may remain constant even with the use of horizontal smooth scrolling.

If the defined character is to have horizontal intercharacter spacing, then R22(3–0) should be initialized with a value equal to the number of pixels (horizontal) in the displayed part of a character, minus 1. In this case, the value of R22(3–0) must be changed along with R25(3–0) to implement horizontal smooth scrolling.

In the pixel double-width mode, the value written to R22(3–0) should be 1 larger than the numbers described above.

## R22(7–4) CHARACTER TOTAL, HORIZONTAL

The number of pixels (horizontal) in a character, minus 1. This number includes the displayed part of a character and the horizontal intercharacter spacing to the right of the displayed part. This register sets the horizontal dimension of a character.

In the pixel double-width mode, the value written to R22(7–4) should be the number of pixels (horizontal) in a character. Do not subtract 1 from the value in this mode.

## R23(4–0) CHARACTER DISPLAYED, VERTICAL

The number of scan lines, minus 1, of the displayed part of a character. This number sets the height of the displayed part of the character, and defines the vertical intercharacter spacing below the displayed part.

R23(4–0) can be equal to or less than R9(4–0). If they are equal, then the displayed part of the character is equal to the total character, and the entire character is displayed, with no vertical intercharacter spacing.

## R24(4–0) VERTICAL SMOOTH SCROLL

The sudden scroll of an entire character row is too abrupt a change for some applications. The 8563 supports a method of vertically scrolling the screen one scan line at a time, up to one full character row. This is called vertical smooth scroll. This operates along with vertical scrolling to allow the screen to be scrolled up or down smoothly for as many character rows as desired.

To scroll the screen one scan line 'up' vertically, the CPU should write a I to R24(4–0). This will skip the first scan line of the first character row. The first character row will then begin with the second scan line, moving it 'up' one scan line. All character rows on the frame will be moved 'up' one scan line also. At the bottom of the Frame, one additional scan line will be displayed of a 'new' character row. This will actually display parts of "N + 1" character rows instead of the standard N.

The number in R24(4–0) can be increased, increasing the number of scan lines scrolled, until R24(4–0) equals R9(4–0). This is the maximum amount of smooth scroll allowed. At this point, only one (the bottom) scan line will exist for the first character row and the last character row will have scrolled onto the frame with only the last scan line not visible. To scroll further, the CPU must change R24(4–0) back to 0, and increase R12/R13 (display start address) and R20/R21 (attribute start address) to scroll vertically by one character row. This sequence will have smooth scrolled by one character row.

## R24(5) CHARACTER BLINK RATE

When attributes are enabled, the attribute byte for each character contains a bit that controls the blinking of that character. The rate at which the characters blink is determined by R24(5). If this bit is a 0, then the character blink rate is one-sixteenth of the frame rate, if 1, then one thirty-second of the frame rate.

## R24(6) REVERSE SCREEN

The complete screen can be reversed (foreground and background colors reversed) by changing R24(6). With this bit 0, the screen will be displayed with normal foreground/background colors. A 1 will reverse the foreground and background colors for each character on the screen. This reversal will re-reverse any character normally reversed by the cursor and/or the individual character reverse attribute.

## R24(7) BLOCK COPY
## R30 WORD COUNT

To further improve the speed at which the CPU can manipulate 8563 memory, two additional features have been added to the 8563: Block Write and Block Copy. Block Write is an extension of the CPU write cycle except that Block Write writes the same data to more than one successive memory location. This operation is set up by writing the initial address to R18/R19, waiting for the Update Ready Status bit to be 1, writing the data to R31 and again waiting for the Update Ready Status bit to be 1. This is the same as CPU write (above). The CPU must then write a 0 to R24(7) selecting the Block Write mode, then write to R30 the number of successive memory locations the 8563 should write. Following the write to R30, the 8563 will initiate one or more write cycles to 8563 memory, writing the data in R31 to successive memory locations. The contents of R18/R19 will increment with the addresses being written. After the Block Writes are completed, R18/R19 will contain the address following the last memory location written.

> **NOTE:** One write cycle will follow the initial write of data to R31, so the quantity written to R30 should be one *less* than the total number of memory locations desired to be written.

## R25(3–0) SMOOTH HORIZONTAL SCROLL

The 8563 has the ability to scroll smoothly in the horizontal direction. Horizontal smooth scrolling moves all characters on the screen to the 'left' the desired number of

pixels, up to the width of one character. Before horizontal smooth scrolling can occur, the 8563 must have something to scroll 'onto' the right side of the screen, so R27 must be greater than 0. This sets up a virtual screen at least one character greater than the displayed screen.

An unscrolled screen is obtained with the value of R25(3–0) equal to the value of R22(7–4) (character total, horizontal). The screen is scrolled left by one pixel by decreasing the contents of R25(3–0) by 1. Additional scrolling occurs with decreasing values programmed. Maximum scrolling is achieved with a value of 0 in R25(3–0). At this point, only one (the right) pixel will exist for the first character of each character row and the last character will have scrolled onto the row with only one (the right) pixel still not visible. To scroll further, the CPU must change R25(3–0) back to its maximum value and increase R12/R13 (Display Start Address) and R20/R21 (Attribute Start Address) each by 1 to scroll horizontally by one character. This sequence will have smooth scrolled left by one character.

If the character has any nondisplayed pixels {R22(3–0) less than [R22(7–4) plus one]} then the value of R22(3–0) (character displayed, horizontal) must be changed along with R25(3–0) for horizontal scrolling. The value of R22(3–0) should be equal to [(the number of horizontal pixels in the displayed part of a character) plus (the value of R25(3–0), horizontal smooth scroll)] modulo [(the value of R22(7–4), character total, horizontal) plus 1]. In this case both registers R25(3–0) and R22(3–0) must be changed to scroll horizontally.

For example, for a character 8 pixels wide with 5 pixels wide displayed:

| | R22(7–4) | R25(3–0) | R22(3–0) |
|---|---|---|---|
| **Unscrolled** | **7** | **7** | **4** |
| **Scroll 1** | **7** | **6** | **3** |
| **Scroll 2** | **7** | **5** | **2** |
| **Scroll 3** | **7** | **4** | **1** |
| **Scroll 4** | **7** | **3** | **0** |
| **Scroll 5** | **7** | **2** | **7** |
| **Scroll 6** | **7** | **1** | **6** |
| **Scroll 7** | **7** | **0** | **5** |

There are two different versions of the 8563 chip (80-column RGBI) in production, requiring slightly different register initialization values.

The difference between the 8563-R7A and subsequent revisions (8563-R8 and 8563-R9) is in the horizontal smooth scroll feature, represented by bits 0–3 of register 25. Even if this feature is not utilized, the correct value must be placed into this register for a normal 80-column display. Software designed specifically for 8563-R7As will exhibit a problem with the leftmost side of the display when run on a system with R8s or R9s, and software designed specifically for 8563-R8s and R9s will exhibit a problem with the rightmost side of the display when run on a system with R7A. To run correctly on any system, the software must initialize register 25 with the correct data for the

particular 8563 version in the system. The 8563 version can be easily ascertained through software by reading the 8563 status register, located at $D600 in I/O memory. Bits 0–2 contain the version number. Refer to the table below for the actual data to be used.

| 8563 rev | version ($D600) | register 25 value |
|---|---|---|
| 7A | 0 | $40 |
| 8 | 1 | $47 |
| 9 | 1 | $47 |

This situation does not directly affect software which utilizes the resident C128 operating system (Kernal). The Kernal correctly determines which version of the 8563 is present and initializes it accordingly. Software which relies on the Kernal IOINIT routine to accomplish this initialization will not have any problems. Software which performs the initialization itself, or alters the contents of the affected register, may exhibit the 80-column display anomalies described above, but will otherwise function normally.

## R25(5) PIXEL DOUBLE WIDTH

A change from an 80-column frame to a 40-column frame requires the registers of the 8563 to be initialized differently and requires a different pixel rate. To simplify this change, a control bit on the 8563 causes the DCLK input signal to be divided by 2. If R25(4) is a 0, then the pixel width is one DCLK period. If it is a 1, then the pixel width is two DCLK periods and the internal character clock period is twice the normal period. The AC timing is the same as normal. The 8563 RAM bus cycles occur at half the normal rate, with the "idle" time spent with RAS and CAS "high." Note that the contents of the horizontal initialization registers, and the contents of R22(7–4) and R22(3–0) must be initialized with different values in this mode.

## R25(5) SEMIGRAPHIC MODE

The displayed part of a character is limited to 8 pixels wide owing to the width of the Display Data Bus (DD0–DD7). Low-resolution graphics regularly use a "block-graphics" character set that may exceed 8 pixels in width. The 8563 allows this in a limited way by allowing the last displayed pixel of a character to be repeated in the horizontal intercharacter space, which is normally forced to the background color. If the control bit R25(5) (semigraphic mode) is a 0, then normal character operation occurs. If it is a 1, then semigraphics operation occurs, allowing characters to extend into the horizontal intercharacter space, touching the next character.

## R25(6) ATTRIBUTE ENABLE

In some applications, attributes are not necessary. In these cases, the amount of 8563 RAM memory used by the attributes may be excessive, so the 8563 has a control bit to

disable attribute usage. If R25(6) is a 0, then attributes will not be fetched by the 8563 and the RAM memory may be used for other purposes. In this case, no character may be underlined, individually reversed or blinked. Only one character set of 256 characters may be used. The foreground color of all characters will be the same and will be determined by R26(7–4). The background color will still be set by R26(3–0). If R25(6) is a 1, then attributes will be used normally.

## R25(7) MODE SELECT (Text/Bit Map)

The 8563 is primarily a text display device, but it does support a limited bit-mapped graphics mode. In the text mode, the displayed frame is an array of characters, each with a unique pointer to reference the character, and an attribute to define additional display characteristics. In the bit-mapped mode each pixel is controlled by a unique bit in 8563 RAM memory. For text operation, bit 7 is initialized to a 0; for bit-mapped operation, set bit 7. Refer to the Bit-Mapping the 80-column Display section in this chapter for more information.

## R26(3–0) BACKGROUND R, G, B, I

In all cases, the background color is determined by R26(3–0). The borders surrounding the screen and the vertical and horizontal intercharacter spaces are displayed at the background color.

## R26(7–4) FOREGROUND R, G, B, I

When attributes are disabled, the foreground color for all characters is determined by R26(7–4).

## R27 ADDRESS INCREMENT PER ROW

An additional feature of the 8563 is the ability to scroll horizontally, both left and right through display 8563 RAM memory. To scroll a window of i horizontal bytes by j vertical scan lines across a virtual screen of k horizontal by l vertical ($i < k$) there must be ($k \times l$) bytes of bit-mapped data in display 8563 RAM memory. The 8563 must "skip" (k–i) bytes on every scan line, because only i bytes are displayed of a virtual line of k bytes in display 8563 RAM memory. The 8563 has the ability to skip a number of bytes, set by writing to R27. If R27 is 0, then no bytes will be skipped. A nonzero value in R27 is necessary to allow horizontal smooth-scrolling.

The value in R27 is used to increment the address of the bit-mapped data from one scan line to the next and to increment the address of the attributes from one character row to the next. Both the bit-mapped data and the attributes will be incremented by the same amount, the value of R27.

## R28(4) 8563 RAM TYPE (4416/4164)

The data for the displayed text is stored in a dynamic RAM dedicated to the display and external to the 8563. The CPU accesses this 8563 RAM display memory indirectly through the 8563 internal registers. The 8563 internally calculates 16-bit RAM addresses, so can address up to 64K bytes of 8563 RAM. The 8563 RAM addresses may

be configured to be compatible with 4164 8563 RAMs (64K by 1) or with 4416 8563 RAMs (16K × 4). Register 28, bit 4 should be initialized to a 0 for operation with 4416s a 1 for operation with 4164s.

4416: Row/Col = A7/A15, A6/A13, A5/A12, A4/A11, A3/A10, A2/A9, A1/A8, A0/A8

4164: Row/Col = A7/A15, A6/A14, A5/A13, A4/A12, A3/A11, A2/A10, A1/A9, A0/A8

## R28(7–5) CHARACTER SET START ADDRESS

The starting location of the character set in display 8563 RAM is determined by R28(7–5). These bits form the most significant 3 bits of the 16-bit address used to access character data. If R9(4–0) is greater than 15, then only bits 7–6 of R28 will be used, because each character set will occupy 8192 bytes of 8563 RAM memory.

The character pointers correspond to a displayed character position on the frame. The 8-bit pointer selects one of the 256 characters in the character set to be displayed at that position on the frame. For a frame of i rows of j characters each, the first j pointers will define the characters in the first (top) character row, and (i × j) pointers will need to be defined in display 8563 RAM. For 25 character rows of 80 characters each, 2000 pointers will be needed.

## R29(4–0) UNDERLINE SCAN LINE COUNT

The number of the scan line used for underline. The underline is defined as a single scan line. Scan line 0 refers to the scan line at the top of the character.

## R32, R33 BLOCK COPY SOURCE ADDRESS

Block Copy is similar to Block Write, except that the data written to 8563 memory is obtained from other successive memory locations. The Block Copy actually 'copies' one part of 8563 memory to another. To set this operation up, the CPU writes the initial Destination Address to R18/R19, waits for the Update Ready Status bit to be 1 and writes a 1, to R24(7) selecting the Block Copy mode. The CPU then writes the initial source address to R32 (most significant byte) and R33 (least significant), then the number of successive memory locations to be copied should be written to R30. The 8563 will then read the contents of the first source address and write that data to the first destination address. Additional copies will occur at addresses incremented from the source and destination addresses, for as many words as are defined in R30.

## R34, R35 DISPLAY ENABLE BEGIN, END

During the horizontal and vertical synchronization pulses, the R, G, B and I signals must be blanked (brought to a low level) to prevent the display of the scanning beam during retrace times. Two registers allow the user to adjust the beginning (R34) and end (R35) of the horizontal blanking interval.

Register R34 is programmed with the number of characters from the first displayed character of a row to the first blanked character in that row. Register R35 is programmed with the number of characters from the first displayed character of a row to the last blanked character in that row. Blanking occurs on all scan lines of a frame.

## R36(3–0) 8563 RAM REFRESH/SCAN LINE

The number of Dynamic RAM refresh cycles every scan line. These refresh cycles occur on displayed and nondisplayed scan lines, during the displayed part of the screen and nondisplayed parts (blanked scan lines, vertical borders and vertical sync). Each refresh address is incremented from the previous refresh address, incrementing through all 65,536 addresses. The least significant byte of the 16-bit Display 8563 RAM memory address is used as the row address to the 8563 RAM. As the 16-bit refresh address is incremented, successive rows of the 8563 RAM are refreshed.

# 11

# SOUND AND MUSIC ON THE COMMODORE 128

# INTRODUCTION

The Commodore 128 has one of the most sophisticated built-in sound synthesizers available in a personal computer. The synthesizer, called the Sound Interface Device (SID), is a chip dedicated solely to generating sound and music. The SID chip is capable of producing three independent **voices** (sounds) simultaneously. Each of the voices can be played in one of four types of sounds, called waveforms. The SID chip also has programmable Attack, Decay, Sustain and Release (ADSR) parameters. These parameters define the quality of a sound. In addition, the synthesizer has a filter you can use to choose certain sounds, eliminate others, or modify the characteristics of a sound or sounds. All these features add up to a powerful and versatile synthesizer.

To make it easy for you to select and manipulate the many capabilities of the SID chip, Commodore has developed new and powerful BASIC music statements.

Here are the new sound and music statements available on the Commodore 128:

SOUND
ENVELOPE
VOL
TEMPO
PLAY
FILTER

The beginning of this section explains these sound statements, in an encyclopedia format. The second half describes how to program the SID chip in machine language.

# ENVELOPE

Define a musical instrument envelope

**ENVELOPE n,[,atk] [,dec] [,sus] [rel][,wf] [,pw]**

where:

| | |
|---|---|
| **n** | Envelope number (0–9) |
| **atk** | Attack rate (0–15) |
| **dec** | Decay rate (0–15) |
| **sus** | Sustain (0–15) |
| **rel** | Release rate (0–15) |
| **wf** | Waveform: 0 = triangle<br>1 = sawtooth<br>2 = variable pulse (square)<br>3 = noise<br>4 = ring modulation |
| **pw** | Pulse width (0–4095) |

A parameter that is not specified will retain its predefined or currently redefined value. Pulse width applies to the width of the variable pulse waveform (wf = 2) only. The Commodore 128 has initialized the following ten envelopes:

| | N | A | D | S | R | wf | pw | instrument |
|---|---|---|---|---|---|---|---|---|
| **ENVELOPE** | **0,** | **0,** | **9,** | **0,** | **0,** | **2,** | **1536** | **piano** |
| **ENVELOPE** | **1,** | **12,** | **0,** | **12,** | **0,** | **1** | | **accordion** |
| **ENVELOPE** | **2,** | **0,** | **0,** | **15,** | **0,** | **0** | | **calliope** |
| **ENVELOPE** | **3,** | **0,** | **5,** | **5,** | **0,** | **3** | | **drum** |
| **ENVELOPE** | **4,** | **9,** | **4,** | **4,** | **0,** | **0** | | **flute** |
| **ENVELOPE** | **5,** | **0,** | **9,** | **2,** | **1,** | **1** | | **guitar** |
| **ENVELOPE** | **6,** | **0,** | **9,** | **0,** | **0,** | **2,** | **512** | **harpsichord** |
| **ENVELOPE** | **7,** | **0,** | **9,** | **9,** | **0,** | **2,** | **2048** | **organ** |
| **ENVELOPE** | **8,** | **8,** | **9,** | **4,** | **1,** | **2,** | **512** | **trumpet** |
| **ENVELOPE** | **9,** | **0,** | **9,** | **0,** | **0,** | **0** | | **xylophone** |

## FILTER

Define sound (SID chip) filter parameters

**FILTER [freq], [,lp] [,bp] [,hp] [,res]**

where:

**freq** Filter cut-off frequency (0–2047)
**lp** Low-pass filter on (1), off (0)
**bp** Band-pass filter on (1), off (0)
**hp** High-pass filter on (1), off (0)
**res** Resonance (0–15)

Unspecified parameters result in no change to the current value.

You can use more than one type of filter at a time. For example, both low-pass and high-pass filters can be used together to produce a notch-(or band-reject) filter response. For the filter to have an audible effect, at least one type of filter must be selected and at least one voice must be routed through the filter.

**EXAMPLES:**

| | |
|---|---|
| FILTER 1024,0,1,0,2 | Set the cutoff frequency at 1024, select the band pass filter and a resonance level of 2. |
| FILTER 2000,1,0,1,10 | Set the cutoff frequency at 2000, select both the low pass and high pass filters (to form a notch reject) and set the resonance level at 10. |

# PLAY

Defines and plays musical notes and elements

**PLAY "Vn,On,Tn,Un,Xn,elements"**

**Vn** Voice (n = 1–3)
**On** Octave (n = 0–6)
**TN** Tune Envelope Defaults (n = 0–9)
0 = piano
1 = accordion
2 = calliope
3 = drum
4 = flute
5 = guitar
6 = harpsichord
7 = organ
8 = trumpet
9 = xylophone
**Un** Volume (n = 0–15)
**Xn** Filter on (n = 1), off (n = 0)
**Notes:** A,B,C,D,E,F,G

**Elements:**

| | |
|---|---|
| **#** | Sharp |
| **$** | Flat |
| **W** | Whole note |
| **H** | Half note |
| **Q** | Quarter note |
| **I** | Eighth note |
| **S** | Sixteenth note |
| **.** | Dotted |
| **R** | Rest |
| **M** | Wait for all voices currently playing to end current measure |

The PLAY statement gives you the power to select voice, octave and tune envelope (including ten predefined musical instrument envelopes), the volume and the notes you want to PLAY. All these controls are enclosed in quotes.

All elements except R and M precede the musical notes in a PLAY string.

**EXAMPLES:**

| | |
|---|---|
| PLAY "V1O4T0U5X0CDEFGAB" | Play the notes C,D,E,F,G,A and B in voice 1, octave 4, tune envelope 0 (piano), at volume 5, with the filter off. |
| PLAY "V3O5T6U7X1#B$AW.CHDQEIF" | Play the notes B-sharp, A-flat, a whole dotted-C note, a half D-note, a quarter E-note and an eighth F-note. |

## SOUND

Create sound effects and musical notes

**SOUND v,f,d[,dir][,m][,s][,w][,p]**

where:

**v** voice (1–3)
**f** frequency value (0–65535)
**d** duration (0–32767)
**dir** step direction (0(up), 1(down) or 2(oscillate)) default = 0
**m** minimum frequency (if sweep is used) (0–65535) default = 0
**s** step value for sweep (0–32767) default = 0
**w** waveform (0 = triangle, 1 = sawtooth, 2 = variable, 3 = noise) default = 2
**p** pulse width (0–4095) default = 2048

The SOUND command is a fast and easy way to create sound effects and musical tones. The three required parameters v,f and d select the voice, frequency and duration of the sound. The duration is in units called jiffies. Sixty jiffies equals 1 second.

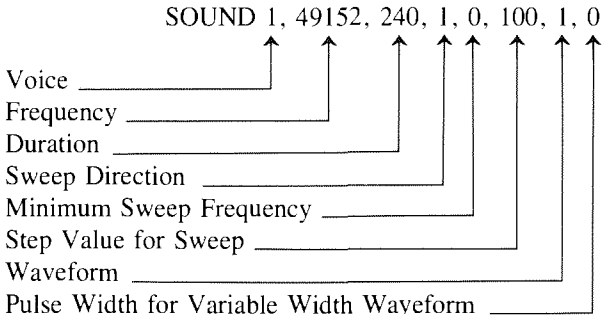
The SOUND command can sweep through a series of frequencies which allows sound effects to pass through a range of notes. Specify the direction of the sweep with the DIR parameter. Set the minimum frequency of the sweep with M and the step value of the sweep with S. Select the appropriate waveform with W and specify P as the width of the variable pulse waveform if selected in W.

**EXAMPLES:**

| | |
|---|---|
| SOUND 1,40960,60 | Play a SOUND at frequency 40960 in voice 1 for 1 second. |
| SOUND 2,20000,50,0,2000,100 | Output a sound by sweeping through frequencies starting at 2000 and incrementing upward in units of 100. Each frequency is played for 50 jiffies. |

SOUND 3,5000,90,2,3000,500,1 This example outputs a range of sounds starting at a minimum frequency of 3000, through 5000, in increments of 500. The direction of the sweep is back and forth (oscillating). The selected waveform is sawtooth and the voice selected is 3.

**EXAMPLES:**




```
10 DO
20 PRINT"VC  FREQ  DIR  MIN   SV   WF   PW ":PRINT
30 V=INT(RND(1)*3)+1    :REM VOICE
40 F=INT(RND(1)*65535)  :REM FREQ
50 D=INT(RND(1)*32767)  :REM DURATION
60 DIR=INT(RND(1)*3)    :REM STEP DIR
70 M=INT(RND(1)*65535)  :REM MIN FREQ
80 S=INT(RND(1)*32767)  :REM STEP VAL
90 W=INT(RND(1)*4)      :REM WAVEFORM
100 P=INT(RND(1)*4095)  :REM PULSE W
110 PRINT V; F;DIR;M;S;W;P:PRINT:PRINT
120 SOUND V, F, D, DIR, M, S, W, P
130 SLEEP 4
140 SOUND V, 0, 0, DIR, 0, 0, W, P
150 LOOP
```

# TEMPO

Define the speed of the song being played

**TEMPO n**

where n is a relative duration between (0 and 255)

The actual duration for a whole note is determined by using the formula given below:

whole note duration = 19.22/n seconds

The default value is 8, and note duration increases with n.

**EXAMPLES:**

TEMPO 16 Defines the TEMPO at 16.
TEMPO 1 Defines the TEMPO at the slowest speed.
TEMPO 250 Defines the TEMPO at 250.

## VOL

Define output level of sound

**VOL volume level**

This statement sets the volume for SOUND and PLAY statements. VOLUME level can be set from 0 to 15, where 15 is the maximum volume, and 0 is off. VOL affects all voices.

**EXAMPLES:**

VOL 0 Sets volume to its lowest level.

VOL 15 Sets volume for SOUND and PLAY statements to its highest output.

# CODING A SONG FROM SHEET MUSIC

This section provides a sample piece of sheet music and illustrates how to decode notes from a musical staff and translate them into a form the Commodore 128 can understand. This exercise is substantially faster and easier if you know how to read music. However, you don't have to be a musician to be able to play the tune on your Commodore 128. For those of you who cannot read music, Figure 11–1 shows how a typical musical staff is arranged and how the notes on the staff are related to the keys on a piano.

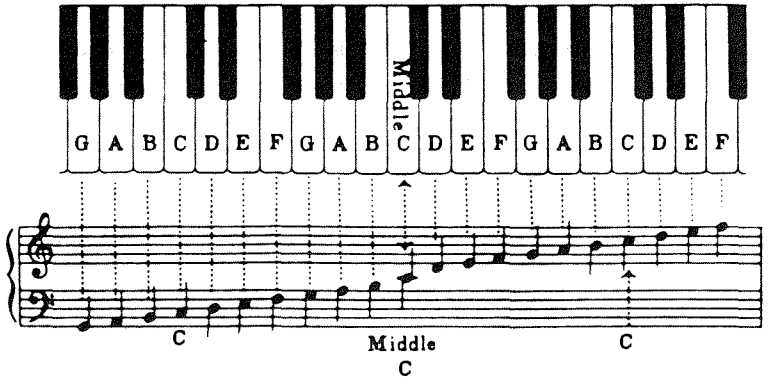



**Figure 11–1. Musical Staff**

Figure 11–2 is an excerpt from a composition titled *Invention 13* (*Inventio 13 in Italian*), by Johann Sebastian Bach. Although this composition was written a few hundred years ago, it can be played and enjoyed on the most modern of computer synthesizers, such as the SID chip in the Commodore 128. Here are the opening measures of *Invention 13*.

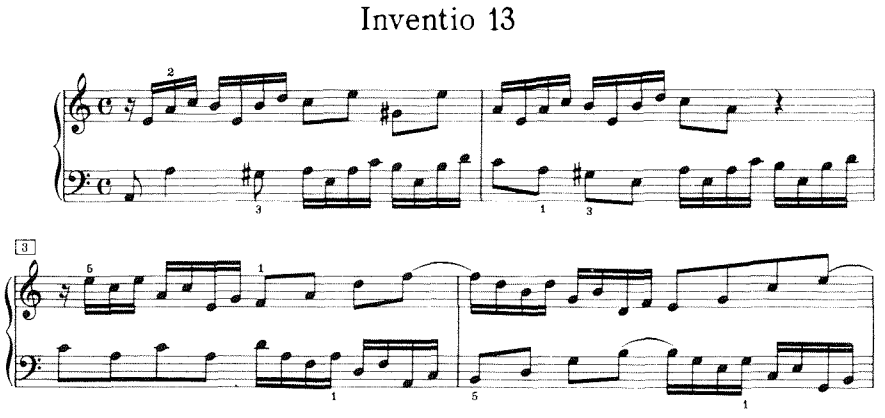



**Figure 11–2. Part of Bach's *Invention 13***

The best way to start coding a song on your Commodore 128 is to break the notes down into an intermediate code. Write down the upper staff notes on a piece of paper. Now write down the notes for the lower staff. Precede the note values with a duration code. For instance, precede an eighth note with an 8, precede a sixteenth note with a 16, and so on. Next, separate the notes so the notes on the upper staff for one measure are proportional in time with the notes for one measure on the lower staff.

If the musical composition had a third staff, you would separate it so the duration would be proportional to the two other upper staffs. Once the notes for all the staffs are separated into equal durations, a separate and dedicated voice plays each note for a particular staff. For example, voice 1 plays the upper staff, voice 2 plays the second staff and voice 3 plays the lowest staff.

Let's say the upper staff begins with a string of four eighth notes. In addition, say the lower staff begins with a string of eight sixteenth notes. Since an eighth note is proportional in time to two sixteenth notes, separate the notes as shown in Figure 11–3.

```
V1 =  8A       8B       8C       8D
V2 =  16D 16E  16F 16G  16A 16B  16C 16D
```

**Figure 11–3. Synchronizing Notes for Two Voices**

The synchronization and timing in a musical composition are critical so you must make sure the notes in the upper staff for voice 1, for example, are in time agreement with the notes in the lower staff for voice 2. The first note in the upper staff in Figure 11-3 is an A eighth note. The first two notes for voice 2 are D and E sixteenth notes. In this case, you must enter the voice 1 eighth note in the PLAY string first, then follow the voice 2 sixteenth notes immediately after it. To continue the example, the second note in Figure 11–3 for voice 1 (the upper staff) is a B eighth note. The B eighth note is equal in time to the two sixteenth notes, F and G, which appear in the bottom staff for voice 2. In order to coordinate the time, enter the B eighth note in the string for voice I and follow it with the two sixteenth notes, F and G, for voice 2.

As a rule, always start with the note with the longer duration. For example, if a bar starts with a series of two sixteenth notes on the lower staff for voice 2 and the upper staff starts with an eighth note for voice 1, enter the eighth note in the string first since it must play for the duration while the two sixteenth notes are being fetched by the Commodore 128. You must give the computer time to play the longer note first, and then PLAY the notes of shorter duration, or else the composition will not be synchronized.

Here's the program that plays *Invention 13*. Spaces are omitted in the PLAY string to conserve space. For readability in your programs, add space between elements in the string. Enter it into your C128 and SAVE it for future use. Now RUN it and sit back and enjoy the music!

```
10 REM INVENTION 13 BACH
20 TEMPO 6
30 PLAY"V1O4T7U8X0":REM VOICE 1=ORGAN
40 PLAY"V2O4T0U8X0":REM VOICE 2=PIANO
50 REM FIRST MEASURE
60 A$="V2O1IAV1O3IEV2O2QAV1O3SAO4CO3BEV2O2I#GV1O3SBO4DV1O4ICV2O2SAEM"
70 B$="V1O4IEV2O2SAO3CV1O3I#GV2O2SBEV1O4IEV2O2SBO3D"
80 REM SECOND MEASURE
90 C$="V2O3ICV1O3SAEV2O2IAV1O3SAO4CV2O2I#GV1O3SBEV2O2IEV1O3SBO4D"
100 D$="V1O4ICV2O2SAEV1O3IAV2O2SAO3CV1O4QRV2O2SBEBO3D"
110 REM   REM THIRD MEASURE
120 E$="V2O3ICV1O4SREV2O2IAV1O4SCEV2O3ICV1O3SAO4CV2O2IAV1O2SEG"
130 F$="V1O3IFV2O3SDO2AV1O3IAV2O2SFAV1O4IDV2O2SDFV1O4IFV2O1SAO2C"
140 REM  FOURTH MEASURE
150 G$="V2O1IBV1O4SFDV2O2IDV1O3SBO4DV2O2IGV1O3SGBV2O2IBV1O3SDF"
160 H$="V1O3IEV2O2SGEV1O3IGV2O2SEGV1O4ICV2O2SCEV1O4IEV2O1SGB"
170 REM  FIFTH MEASURE
180 I$="V2O1IAV1O4SECV2O2ICV1O3SAO4CV1O3IFV2O2SDFV1O4IDV2O1SBO2D"
190 J$="V2O1IGV1O3SDBV2O1IBV1O3SGBV1O3IEV2O2SCEV1O4ICV2O1SAO2C"
200 REM SIXTH MEASURE
210 K$ = "V2O1IFV1O4SCO3AV2O1IDV1O3SFAV1O3IDV2O1SGO2GV1O3IBV2O2SFG"
220 M$="V2O1IAV1O4SCO3AV2O2I#FV1O4SCEV2O1IBV1O4SDO3BV2O2I#GV1O4SDF"
230 REM SEVENTH MEASURE
240 N$="V2O2ICV1O4SECV2O2IAV1O4SEGV2O2IDV1O4SFEV2O2I$BV1O4SDC"
250 O$="V2O2I#GV1O3SBO4CV2O2IFV1O4SDEV2O2IDV1O4SFDV2O1IBV1O4S#GD"
260 REM EIGHTH MEASURE
270 P$="V2O2I#GV1O4SBDV2O2IAV1O4SCAV2O2IDV1O4SFDV2O2IEV1O3SBO4D"
280 Q$="V2O2IFV1O3S#GBV2O2I#DV1O4SCO3AV2O2IEV1O3SEAV2O2IEV1O3SB#G"
290 REM NINTH MEASURE
300 R$="V2O1HAV1O3SAECEO2QA"
310 PLAY A$:PLAY B$:PLAY C$:PLAY D$:PLAY E$
320 PLAY F$:PLAY G$:PLAY H$:PLAY I$:PLAY J$
330 PLAY K$:PLAY M$:PLAY N$:PLAY O$:PLAY P$
340 PLAY Q$:PLAY R$
```

Here are two sample sounds programs that you can try:

```
10 REM SOUND LOOPS
20 REM CHANGES STEP VALUE, DIRECTION AND WAVEFORM
30 SV=1000:REM STEP VALUE
40 DO
50 PRINT:PRINT "SV=";SV:PRINT
60 :   DIR=0 :REM SWEEP DIRECTION
70 :     DO
80 :       WF=0:REM WAVEFORM
90 :          DO
100 :             SOUND 1,40000,120,DIR,2000,SV,WF
110 :           PRINT "WF=";WF;:WF=WF+1
120 :           LOOP UNTIL WF=4
130 :     PRINT "DIR=";DIR:DIR=DIR+1
140 :     LOOP UNTIL DIR = 3
150 SV=SV+2000
160 LOOP UNTIL SV > 20000
```

```
10 REM C128-MODE JOYSTICK MUSICIAN
20 REM    PLAYS NOTES C THROUGH A IN EACH OCTAVE
30 REM    TO INCREASE THE NOTE VALUE, PUSH JOYSTICK FORWARD
40 REM    TO DECREASE NOTE VALUE, PULL BACK
50 REM    TO INCREASE OCTAVE VALUE, PUSH JOYSTICK TO THE RIGHT
60 REM    TO DECREASE OCTAVE VALUE, PUSH JOYSTICK TO THE LEFT
70 A$(1)="C":A$(2)="D":A$(3)="E":A$(4)="F":A$(5)="G":A$(6)="A":A$(7)="B"
80 Z=4:REM INITIALIZE TO FOURTH OCTAVE
90 SCNCLR :REM CLEAR SCREEN
100 INPUT"ENTER VOICE 1 INSTRUMENT NUMBER (0-9)";N$
110 INPUT"ENTER VOICE 2 INSTRUMENT NUMBER (0-9)";O$
120 INPUT"ENTER VOICE 3 INSTRUMENT NUMBER (0-9)";P$
130 PRINT:PRINT:PRINT:PRINT:PRINT:PRINT"             INCREASE NOTE"
140 PRINT:PRINT"                          ^":PRINT
150 PRINT"DECREASE OCTAVE <-- --> INCREASE OCTAVE"
160 PRINT:PRINT"                          V":PRINT
170 PRINT:PRINT"             DECREASE NOTE":PRINT
180 PRINT: PRINT"TO INCREASE OR DECREASE NOTE OR OCTAVE";
190 PRINT: PRINT"WITHOUT PLAYING NOTES,HOLD DOWN THE ";
200 PRINT"FIRE BUTTON WHILE POSITIONING THE";
210 PRINT:PRINT"JOYSTICK IN THE APPROPRIATE DIRECTION";
220 Q$="V1"+"T"+N$:REM VOICE 1 INSTRUMENT
230 R$="V2"+"T"+O$:REM VOICE 2 INSTRUMENT
240 S$="V3"+"T"+P$:REM VOICE 3 INSTRUMENT
250 DO
260 : DO
270 :     A=JOY(1):REM GET JOY VALUE
280 :     IF A=0 THEN LOOP:REM IF ZERO LOOP AND GET ANOTHER
290 :     B=A     :REM STORE THE VALUE FOR LATER CHECKING
300 :       Z$="O"+STR$(Z):REM UPDATE OCTAVE
310 :       T$=Z$+Q$+A$(I):REM CONSTRUCT VOICE 1 PLAY STRING
320 :       U$=Z$+R$+A$(I):REM CONSTRUCT VOICE 2 PLAY STRING
330 :       V$=Z$+S$+A$(I):REM CONSTRUCT VOICE 3 PLAY STRING
340 :
350 :       IF (A AND 128)=128 THEN 400
360 :         PLAY T$:REM PLAY VOICE 1
370 :         PLAY U$:REM PLAY VOICE 2
380 :         PLAY V$:REM PLAY VOICE 3
390 :
400 :       IF (A=1 OR A=129) AND (I<8) THEN  I=I+1:REM INCREMENT NOTE VALUE
410 :       IF (A=5 OR A=133) AND (I>0) THEN I=I-1 :REM DECREMENT NOTE VALUE
420 :       IF (A=3 OR A=131) AND (Z<6) THEN Z=Z+1:REM INCREMENT OCTAVE
430 :       IF (A=7 OR A=135) AND (Z>0) THEN Z=Z-1:REM DECREMENT OCTAVE
440 :    A=JOY(1):REM GET NEXT JOY VALUE
450 : LOOP WHILE A=B
460 LOOP:REM DO IT AGAIN
```

# SOUND AND MUSIC IN C128 MODE

## BACKGROUND: THE CHARACTERISTICS OF SOUND

Every sound you hear is actually a sound wave traveling through the air. Like any wave, a sound (sine) wave can be represented graphically and mathematically (see Figure 11–4).

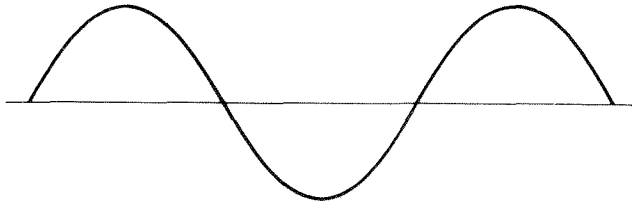


**Figure 11–4. Sine Wave**

The sound wave moves (oscillates) at a particular rate (frequency) which determines the overall pitch (the highness or lowness of the sound).

The sound is also made up of harmonics, which are accompanying multiples of the overall frequency of the sound or note. The combination of these harmonic sound waves give the note its qualities, called timbre. Figures 11–5 shows the relationship of basic sound frequencies and harmonics.

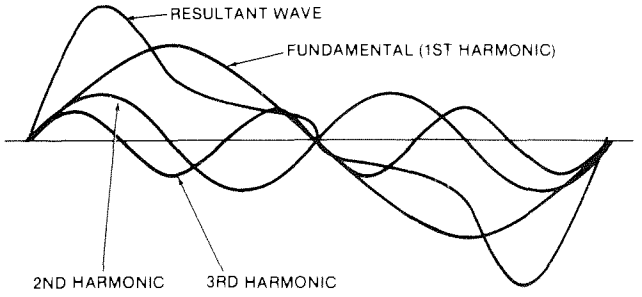



**Figure 11–5. Frequency and Harmonics**

The timbre of a musical tone (i.e., the way a tone sounds) is determined by the tone's waveform. The Commodore 128 can generate four types of waveforms: triangle, sawtooth, variable pulse and noise. See Figure 11–6 for a graphic representation of these four waveforms.

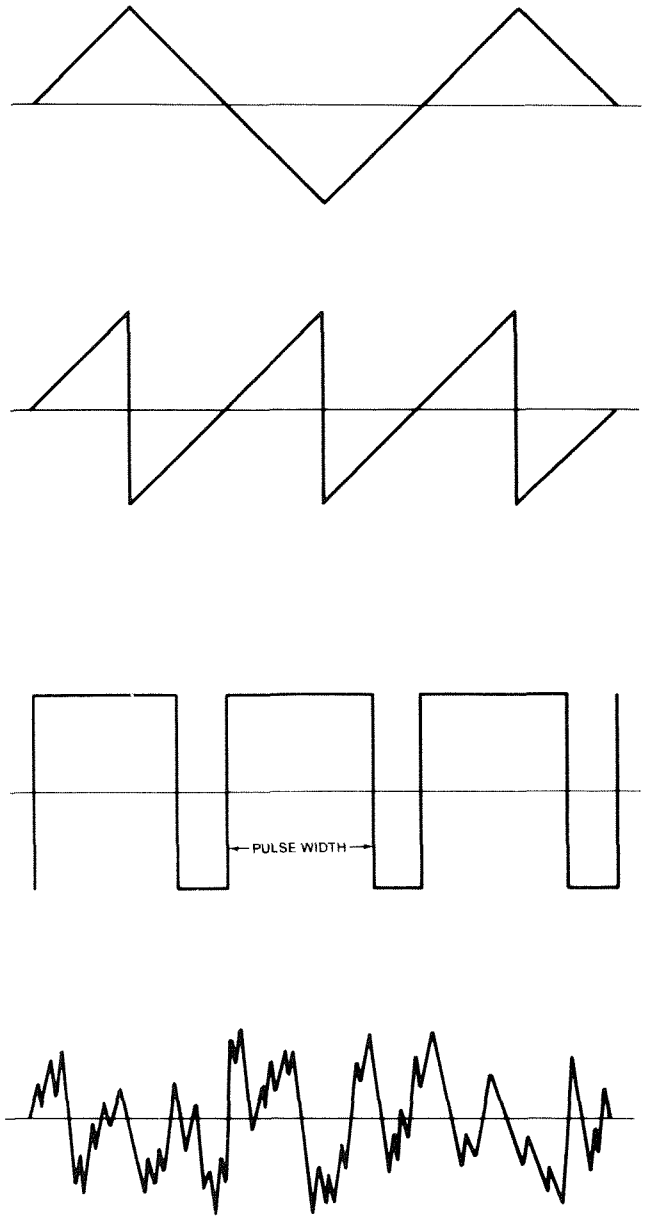



**Figure 11–6. Sound Waveform Types**

The volume of a sound changes throughout the duration of the note, from when you first hear it until it is no longer audible. These volume qualities are referred to as Attack, Decay, Sustain and Release (ADSR). *Attack* is the rate at which a musical note reaches its peak volume. *Decay* is the rate at which a musical note decreases from its peak volume to its midranged (sustain) level. *Sustain* is the level at which a musical note is played at its midrange volume. *Release* is the rate at which a musical note decreases from its sustain level to zero volume. The ENVELOPE generator controls the ADSR parameters of sound. See Figure 11–7 for a graphical representation of ADSR. The Commodore 128 can change each ADSR parameter to sixteen different levels. This gives you absolute flexibility over the ENVELOPE generator and the resulting properties of the volume of the sound.

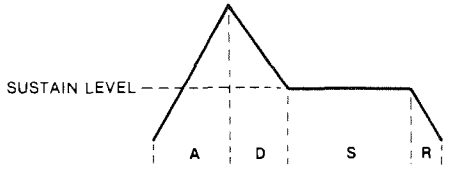



**Figure 11–7. ADSR Phases**

One of the most powerful Commodore 128 sound statements—the one that controls the ADSR and waveform—is the ENVELOPE statement. The ENVELOPE statement sets the different controls in the synthesizer which make each sound unique. The ENVELOPE gives you the power to manipulate the SID synthesizer.

Here are the definitions of the parameters within the Envelope statement:

**Envelope** The properties of a musical note specified by the waveform and the attack, decay, sustain and release settings of the note. For example, the envelope for a guitar note has a different ADSR and waveform than that for a flute.

**Waveform** The type of sound wave created by the combination of accompanying musical harmonics of a tone. The accompanying harmonic sound waves are multiples of, and are based on the overall frequency of the tone. The qualities of the tone generated by each waveform are recognizably different from one another and are represented graphically in Figure 11–6.

**Pulse Width** The length of time between notes, generated by the pulse waveform.

The Commodore 128 has ten predefined envelopes for ten different musical instruments. In using the predefined envelopes you do not have to specify the ADSR parameters, waveform and pulse width settings—this is already done for you. All you have to do is specify the envelope number. The rest of the parameters are chosen

automatically by the Commodore 128. Here are the preselected envelopes for different types of musical instruments:

| ENVELOPE NUMBER | INSTRUMENT | ATTACK | DECAY | SUSTAIN | RELEASE | WAVEFORM | WIDTH |
|---|---|---|---|---|---|---|---|
| 0 | Piano | 0 | 9 | 0 | 0 | 2 | 1536 |
| 1 | Accordion | 12 | 0 | 12 | 0 | 1 | |
| 2 | Calliope | 0 | 0 | 15 | 0 | 0 | |
| 3 | Drum | 0 | 5 | 5 | 0 | 3 | |
| 4 | Flute | 9 | 4 | 4 | 0 | 0 | |
| 5 | Guitar | 0 | 9 | 2 | 1 | 1 | |
| 6 | Harpsichord | 0 | 9 | 0 | 0 | 2 | 512 |
| 7 | Organ | 0 | 9 | 9 | 0 | 2 | 2048 |
| 8 | Trumpet | 8 | 9 | 4 | 1 | 2 | 512 |
| 9 | Xylophone | 0 | 9 | 0 | 0 | 0 | |

**Figure 11–8. Default Parameters for ENVELOPE Statement**

## THE SID FILTER

Once you have selected the ENVELOPE, ADSR, VOLume and TEMPO, use the FILTER to perfect your synthesized sounds. In your program, the FILTER statement will precede the PLAY statement. First you should become comfortable with generating the sound and worry about FILTERing last. Since the SID chip has only one filter, it applies to all three voices. Your computerized tunes will play without FILTERing, but to take full advantage of the music synthesizer, use the FILTER statement to increase the sharpness and quality of the sound.

In the first paragraph of this section, *The Characteristics of Sound*, we defined a sound as a wave traveling (oscillating) through the air at a particular rate. The rate at which a sound wave oscillates is called the wave's frequency. Recall that a sound wave is made up of an overall frequency and accompanying harmonics, which are multiples of the overall frequency. See Figure 11–5. The accompanying harmonics give the sound its timbre, the qualities of the sound which are determined by the waveform. The filter within the SID chip gives you the ability to accent and eliminate the harmonics of a waveform and change its timbre.

The SID chip filters sounds in three ways: low-pass, band-pass and high-pass filtering. These filtering methods are additive, meaning you can use more than one filter at a time. Low-pass filters out frequencies above a certain level you specify, called the cutoff frequency. The cutoff frequency is the dividing line that marks the boundary of which frequency level will be played and which will not. In low-pass filtering, the SID chip plays all frequencies below the cutoff frequency and filters out the frequencies above it. As the name implies, the low frequencies are allowed to pass through the filter and the high ones are not. The low-pass filter produces full, solid sounds. See Figure 11–9.

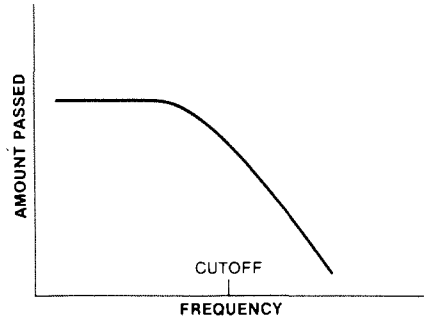



**Figure 11–9. Low-pass Filter**

Conversely, the high-pass filter allows all the frequencies above the cutoff frequency to pass through the chip. All the ones below it are filtered out. See Figure 11–10. The high-pass filter produces tinny, hollow sounds.

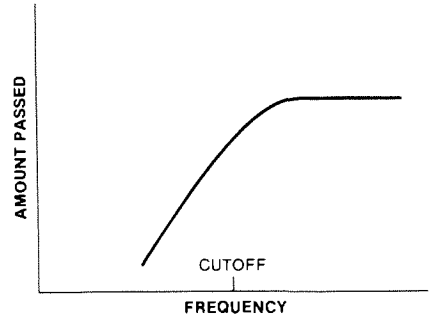



**Figure 11–10. High-pass Filter**

The band-pass filter allows a range of frequencies partially above and below the cutoff frequency to pass through the SID chip. All other frequencies above and below the band surrounding the cutoff frequency are filtered out. See Figure 11–11.

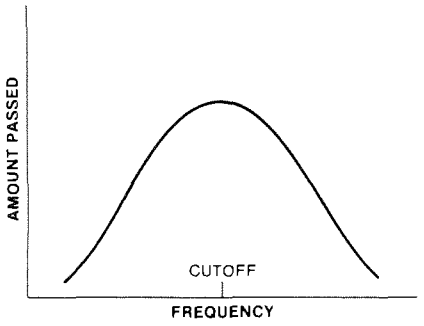



**Figure 11–11. Band-pass Filter**

## ADVANCED FILTERING

Each of the previous FILTERing examples used only one type of filtering at a time. You can combine the SID chip's three methods of filtering with each other to achieve different filtering effects. For example, you can enable the low-pass and high-pass filters at the same time to form a notch reject filter. A notch reject filter allows the frequencies below and above the cutoff to pass through the SID chip, while the frequencies close to the cutoff frequency are filtered out. See Figure 11–12 for a graphic representation of a notch reject filter.

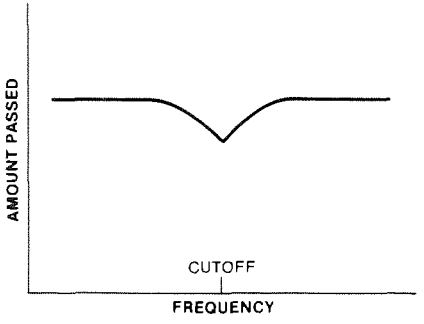



**Figure 11–12. Notch Reject Filter**

You can also add either the low-pass or high-pass filter to the band-pass filter to obtain interesting effects. By mixing the band-pass filter with the low-pass filter, you can select the band of frequencies beneath the cutoff frequency and below. The rest are filtered out.

By mixing the band-pass and the high-pass filters, you can select the band of frequencies above the cutoff frequency and higher. All the frequencies below the cutoff are filtered out.

Experiment with the different combinations of filters to see all the different types of accents you can place on your musical notes and sound effects. The filters are designed to perfect the sounds created by the other components of the SID chip. Once you have created the musical notes or sound effects with the SID chip, go back and add the FILTERing to your programs to make them as crisp and clean as possible.

Now you have all the information you need to write your own musical programs in Commodore 128 BASIC. Experiment with the different waveforms, ADSR settings, TEMPOs and FILTERing. Look in a book of sheet music and enter the notes from a musical scale in sequence within a play string. Accent the notes in the string with the SID control characters. You can combine your Commodore 128 music synthesizer with C128 mode graphics to make your own videos or "movies" complete with sound tracks.

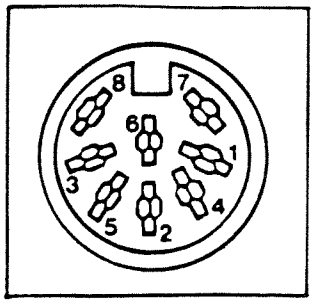
## AUDIO INPUT TO THE SID CHIP

The SID chip has a little known additional feature that most computers do not offer: audio input. This allows you to channel your own music, such as an electric guitar, your voice through a microphone or any other (high impedence, line level) instrument—you can actually "jam" with the C128.

In order to connect properly to the computer, you must devise your own cable or buy one with the correct connections. Get a cable with an 8-pin DIN connector on the end you are plugging into the C128. Connect pins 2 (ground), 3 (audio out), and 5 (audio in) on the 8-pin end.

Here's the pinout:

Video Connector—This DIN connector supplies direct audio and composite video signals. These can be connected to the Commodore monitor or used with separate components. This is the 40-column output connector.



| PIN | TYPE | NOTE |
|---|---|---|
| 1 | LUM/SYNC | Luminance/SYNC output |
| 2 | GND | |
| 3 | AUDIO OUT | |
| 4 | VIDEO OUT | Composite signal output |
| 5 | AUDIO IN | |
| 6 | COLOR OUT | Chroma signal output |
| 7 | NC | No connection |
| 8 | NC | No connection |

Run the three connecting wires inside the cable to attach to the audio in, audio out and ground on the end of your component.

You can connect the C128 to your stereo system or VCR and channel sound through your speakers. Be careful, however, not to exceed the standard electrical restrictions. See the SID chip specs in Chapter 16 under the description EXT IN (Pin 26).

Using the C128 in these ways opens your eyes to how flexible a personal computer can be and how it can communicate with a wide range of devices in your home, including your telephone (with an additional modem), VCR, stereo, TV—and don't forget the electric guitar!

## PROGRAMMING THE SID CHIP IN MACHINE LANGUAGE

This section provides an algorithm and the coded program procedure to play music on the C128.

First you need to know the key SID locations in the C128 memory map. Here's an abbreviated SID memory map, which is exactly the same in both C128 and C64 modes. Register 0 is located at address 54272 ($D400).

| | ADDRESS | | | | | REG # | DATA | | | | | | | | REG NAME | REG TYPE |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $A_4$ | $A_3$ | $A_2$ | $A_1$ | $A_0$ | (HEX) | $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ | | |
| | | | | | | | | | | | | | | | **Voice 1** | |
| 0 | 0 | 0 | 0 | 0 | 0 | 00 | $F_7$ | $F_6$ | $F_5$ | $F_4$ | $F_3$ | $F_2$ | $F_1$ | $F_0$ | FREQ LO | WRITE-ONLY |
| 1 | 0 | 0 | 0 | 0 | 1 | 01 | $F_{15}$ | $F_{14}$ | $F_{13}$ | $F_{12}$ | $F_{11}$ | $F_{10}$ | $F_9$ | $F_8$ | FREQ HI | WRITE-ONLY |
| 2 | 0 | 0 | 0 | 1 | 0 | 02 | $PW_7$ | $PW_6$ | $PW_5$ | $PW_4$ | $PW_3$ | $PW_2$ | $PW_1$ | $PW_0$ | PW LO | WRITE-ONLY |
| 3 | 0 | 0 | 0 | 1 | 1 | 03 | — | — | — | — | $PW_{11}$ | $PW_{10}$ | $PW_9$ | $PW_8$ | PW HI | WRITE-ONLY |
| 4 | 0 | 0 | 1 | 0 | 0 | 04 | NOISE |  |  |  | TEST | RING MOD | SYNC | GATE | CONTROL REG | WRITE-ONLY |
| 5 | 0 | 0 | 1 | 0 | 1 | 05 | $ATK_3$ | $ATK_2$ | $ATK_1$ | $ATK_0$ | $DCY_3$ | $DCY_2$ | $DCY_1$ | $DCY_0$ | ATTACK/DECAY | WRITE-ONLY |
| 6 | 0 | 0 | 1 | 1 | 0 | 06 | $STN_3$ | $STN_2$ | $STN_1$ | $STN_0$ | $RLS_3$ | $RLS_2$ | $RLS_1$ | $RLS_0$ | SUSTAIN/RELEASE | WRITE-ONLY |
| | | | | | | | | | | | | | | | **Voice 2** | |
| 7 | 0 | 0 | 1 | 1 | 1 | 07 | $F_7$ | $F_6$ | $F_5$ | $F_4$ | $F_3$ | $F_2$ | $F_1$ | $F_0$ | FREQ LO | WRITE-ONLY |
| 8 | 0 | 1 | 0 | 0 | 0 | 08 | $F_{15}$ | $F_{14}$ | $F_{13}$ | $F_{12}$ | $F_{11}$ | $F_{10}$ | $F_9$ | $F_8$ | FREQ HI | WRITE-ONLY |
| 9 | 0 | 1 | 0 | 0 | 1 | 09 | $PW_7$ | $PW_6$ | $PW_5$ | $PW_4$ | $PW_3$ | $PW_2$ | $PW_1$ | $PW_0$ | PW LO | WRITE-ONLY |
| 10 | 0 | 1 | 0 | 1 | 0 | 0A | — | — | — | — | $PW_{11}$ | $PW_{10}$ | $PW_9$ | $PW_8$ | PW HI | WRITE-ONLY |
| 11 | 0 | 1 | 0 | 1 | 1 | 0B | NOISE |  |  |  | TEST | RING MOD | SYNC | GATE | CONTROL REG | WRITE-ONLY |
| 12 | 0 | 1 | 1 | 0 | 0 | 0C | $ATK_3$ | $ATK_2$ | $ATK_1$ | $ATK_0$ | $DCY_3$ | $DCY_2$ | $DCY_1$ | $DCY_0$ | ATTACK/DECAY | WRITE-ONLY |
| 13 | 0 | 1 | 1 | 0 | 1 | 0D | $STN_3$ | $STN_2$ | $STN_1$ | $STN_0$ | $RLS_3$ | $RLS_2$ | $RLS_1$ | $RLS_0$ | SUSTAIN/RELEASE | WRITE-ONLY |
| | | | | | | | | | | | | | | | **Voice 3** | |
| 14 | 0 | 1 | 1 | 1 | 0 | 0E | $F_7$ | $F_6$ | $F_5$ | $F_4$ | $F_3$ | $F_2$ | $F_1$ | $F_0$ | FREQ LO | WRITE-ONLY |
| 15 | 0 | 1 | 1 | 1 | 1 | 0F | $F_{15}$ | $F_{14}$ | $F_{13}$ | $F_{12}$ | $F_{11}$ | $F_{10}$ | $F_9$ | $F_8$ | FREQ HI | WRITE-ONLY |
| 16 | 1 | 0 | 0 | 0 | 0 | 10 | $PW_7$ | $PW_6$ | $PW_5$ | $PW_4$ | $PW_3$ | $PW_2$ | $PW_1$ | $PW_0$ | PW LO | WRITE-ONLY |
| 17 | 1 | 0 | 0 | 0 | 1 | 11 | — | — | — | — | $PW_{11}$ | $PW_{10}$ | $PW_9$ | $PW_8$ | PW HI | WRITE-ONLY |
| 18 | 1 | 0 | 0 | 1 | 0 | 12 | NOISE |  |  |  | TEST | RING MOD | SYNC | GATE | CONTROL REG | WRITE-ONLY |
| 19 | 1 | 0 | 0 | 1 | 1 | 13 | $ATK_3$ | $ATK_2$ | $ATK_1$ | $ATK_0$ | $DCY_3$ | $DCY_2$ | $DCY_1$ | $DCY_0$ | ATTACK/DECAY | WRITE-ONLY |
| 20 | 1 | 0 | 1 | 0 | 0 | 14 | $STN_3$ | $STN_2$ | $STN_1$ | $STN_0$ | $RLS_3$ | $RLS_2$ | $RLS_1$ | $RLS_0$ | SUSTAIN/RELEASE | WRITE-ONLY |
| | | | | | | | | | | | | | | | **Filter** | |
| 21 | 1 | 0 | 1 | 0 | 1 | 15 | — | — | — | — | — | $FC_2$ | $FC_1$ | $FC_0$ | FC LO | WRITE-ONLY |
| 22 | 1 | 0 | 1 | 1 | 0 | 16 | $FC_{10}$ | $FC_9$ | $FC_8$ | $FC_7$ | $FC_6$ | $FC_5$ | $FC_4$ | $FC_3$ | FC HI | WRITE-ONLY |
| 23 | 1 | 0 | 1 | 1 | 1 | 17 | $RES_3$ | $RES_2$ | $RES_1$ | $RES_0$ | FILTEX | FILT 3 | FILT 2 | FILT 1 | RES/FILT | WRITE-ONLY |
| 24 | 1 | 1 | 0 | 0 | 0 | 18 | 3 OFF | HP | BP | LP | $VOL_3$ | $VOL_2$ | $VOL_1$ | $VOL_0$ | MODE/VOL | WRITE-ONLY |
| | | | | | | | | | | | | | | | **Misc.** | |
| 25 | 1 | 1 | 0 | 0 | 1 | 19 | $PX_7$ | $PX_6$ | $PX_5$ | $PX_4$ | $PX_3$ | $PX_2$ | $PX_1$ | $PX_0$ | POT X | READ-ONLY |
| 26 | 1 | 1 | 0 | 1 | 0 | 1A | $PY_7$ | $PY_6$ | $PY_5$ | $PY_4$ | $PY_3$ | $PY_2$ | $PY_1$ | $PY_0$ | POT Y | READ-ONLY |
| 27 | 1 | 1 | 0 | 1 | 1 | 1B | $O_7$ | $O_6$ | $O_5$ | $O_4$ | $O_3$ | $O_2$ | $O_1$ | $O_0$ | $OSC_3$/RANDOM | READ-ONLY |
| 28 | 1 | 1 | 1 | 0 | 0 | 1C | $E_7$ | $E_6$ | $E_5$ | $E_4$ | $E_3$ | $E_2$ | $E_1$ | $E_0$ | $ENV_3$ | READ-ONLY |

**Figure 11–13. SID Memory Map**

Here's the general algorithm to output sound from the SID chip in voice 1.

1. Clear the sound chip (For $D400–$D41C)
2. Select ATTACK and DECAY for each voice ($D405)v1
3. Select SUSTAIN and release for each voice ($D406) v1
4. Select a waveform ($D404 for v1)
5. Set the volume ($D418)
6. Set the frequency of the note with a low byte ($D400) and high byte ($D401) from the Musical Note Table in Figure 11–15 at the end of this chapter.

If you want to output a certain frequency level, use this equation:

$F_n = F\ out/.06097$

Round off the ($F_n$) frequency with the integer function and divide the result into the low byte and high byte.
In decimal:

$F_{HI} = F_n/256$ is the high byte

and:

$F_{low} = F_n - (256 * F_{hi})$ is the low byte

To get a feel for the ranges of notes, experiment with the SOUND command in BASIC.

7. Place the high byte in Frequency Control Register (in $D401) for the appropriate voice.
8. Place the low byte in the (low byte) frequency control register for the appropriate voice ($D400).
9. Initiate (GATE) the note or tone. *Gating* means to start the ATTACK, DECAY and SUSTAIN cycle of the tone. Set bit 0 of $D404 (V1), $D40B(V2) and $D412(V3), respectively.
10. Leave bit 0 set for the duration you wish to play the note or tone.
11. Clear the GATE bit.

The durations of the standard musical notes are as follows:

| NOTE TYPE | DURATION |
|---|---|
| 1/16 | **128** |
| 1/8 | **256** |
| **DOTTED** 1/8 | **384** |
| 1/4 | **512** |
| 1/4 + 1/16 | **640** |
| **DOTTED** 1/4 | **768** |
| 1/2 | **1024** |
| 1/2 + 1/16 | **1152** |
| 1/2 + 1/8 | **1280** |
| **DOTTED** 1/2 | **1536** |
| **WHOLE** | **2048** |

To synchronize three voices, divide each measure into sixteen equal parts. Play (Gate) the notes with longer duration first, then gate the values of smaller duration, but of equal proportion, next. For example, one staff has an eighth note. The staff below it has two sixteenth notes. Play the note of longer duration so they can be gated while the other faster notes are being fetched.

The following machine language program plays musical notes in one voice. The musical note data begins at location $1890. The data for each note is stored in the four byte format starting at $1890 as follows:

Frequency (low byte/high byte)
Duration (low byte/high byte)

For example, the data for the second note starts at location $1894, data for the third note begins at location $1898 and so on.

For now, sample data has been supplied. Eventually, place your own data beginning at location $1890. See Figure 11–15 at the end of this chapter for the low- and high-byte frequency values to place in the SID frequency control registers. Refer to the duration table on page 354 for the byte values for the note duration.

The NOP instructions are added between instruction sequences for readability.

Here's the program listing as it appears in the Machine Language Monitor:

```
READY.

MONITOR
    PC  SR AC XR YR SP
; FB000 00 00 00 00 F8

. 01800  A2 00     LDX #$00
. 01802  8A        TXA
. 01803  9D 00 D4  STA $D400,X
. 01806  E8        INX
. 01807  E0 19     CPX #$19
. 01809  D0 F8     BNE $1803
. 0180B  EA        NOP
. 0180C  EA        NOP
. 0180D  EA        NOP
. 0180E  A2 05     LDX #$05
. 01810  A9 F5     LDA #$F5
. 01812  9D 00 D4  STA $D400,X
. 01815  EA        NOP
. 01816  EA        NOP
. 01817  EA        NOP
. 01818  A2 06     LDX #$06
. 0181A  A9 A3     LDA #$A3
. 0181C  9D 00 D4  STA $D400,X
. 0181F  EA        NOP
. 01820  EA        NOP
. 01821  EA        NOP
. 01822  EA        NOP
. 01823  A2 18     LDX #$18
. 01825  A9 0F     LDA #$0F
. 01827  9D 00 D4  STA $D400,X
. 0182A  EA        NOP
. 0182B  EA        NOP
. 0182C  EA        NOP
. 0182D  A0 00     LDY #$00
. 0182F  A2 00     LDX #$00
. 01831  B9 90 18  LDA $1890,Y
```

```
. 01834  F0 3F     BEQ $1875
. 01836  9D 00 D4  STA $D400,X
. 01839  E8        INX
. 0183A  C8        INY
. 0183B  B9 90 18  LDA $1890,Y
. 0183E  9D 00 D4  STA $D400,X
. 01841  EA        NOP
. 01842  EA        NOP
. 01843  EA        NOP
. 01844  C8        INY
. 01845  B9 90 18  LDA $1890,Y
. 01848  85 FA     STA $FA
. 0184A  C8        INY
. 0184B  B9 90 18  LDA $1890,Y
. 0184E  85 FB     STA $FB
. 01850  A2 04     LDX #$04
. 01852  A9 21     LDA #$21
. 01854  9D 00 D4  STA $D400,X
. 01857  EA        NOP
. 01858  EA        NOP
. 01859  EA        NOP
. 0185A  EA        NOP
. 0185B  C6 FA     DEC $FA
. 0185D  D0 FC     BNE $185B
. 0185F  C6 FB     DEC $FB
. 01861  D0 F8     BNE $185B
. 01863  EA        NOP
. 01864  A2 04     LDX #$04
. 01866  A9 20     LDA #$20
. 01868  9D 00 D4  STA $D400,X
. 0186B  EA        NOP
. 0186C  EA        NOP
. 0186D  EA        NOP
. 0186E  C8        INY
. 0186F  4C 2F 18  JMP $182F
. 01872  EA        NOP
. 01873  EA        NOP
. 01874  EA        NOP
. 01875  A2 18     LDX #$18
. 01877  A9 00     LDA #$00
. 01879  9D 00 D4  STA $D400,X
. 0187C  00        BRK


>01890 1E 19 80 00 1C 41 80 00
>01898 1E 19 80 00 1C 41 80 00
>018A0 1E 19 80 00 1C 41 00 00
>018A8 1E 19 80 00 1C 41 80 00
>018B0 1E 19 80 00 1E 19 80 00
>018B8 1E 19 80 01 1E 19 80 01
>018C0 1E 19 80 04 1C 19 80 04
>018C8 1E 19 80 04 1C 41 80 00
>018D0 1E 19 80 00 1C 41 00 00
>018D8 21 21 00 00 00 00 00 00
>018E0 0D 54 08 D0 D7 AD 7B 08
>018E8 8D 6C 08 AD 7C 08 8D 6D
```

Enter the program in memory with the Machine Language Monitor and save it. Run the program with the following command:

G F1800

The program plays a series of notes in a single voice (voice 1). Here's a line by line description of the instructions in the program.

The instructions stored in locations $1800 through $180A clear the SID registers ($D400–$D418) to zero. This is a recommended programming practice, so you can assume that all the SID registers are initialized to zero.

The instructions stored in locations $180E through $1814 assign the attack/decay values in the SID envelope generator. In this program, attack is set at (15:$0F) and the decay is set at 5. This attack setting makes the sounds seem as if they are far away and traveling closer, similar to the way a train sounds as it approaches from far away.

The instructions stored in locations $1818 through $181E select the sustain/release values for the voice 1 envelope generator. In this case the sustain duration is set on 10 ($0A) and the release rate is set to 3. For more information on the ADSR settings, see the beginning of the chapter.

The next sequence of instructions ($1823–$1829) sets the volume to maximum output (15:$0F). Notice that the last three instruction sequences loaded the X register with the register number, loaded the accumulator with the appropriate value and stored the value in the location of the start of the SID plus an offset in the X register. This programming style is easy to follow and is standardized throughout the program.

The instructions stored in locations $182D through $183E get the low- and high-byte frequency values for the musical note data, and place them into the low- and high-byte frequency control registers for voice 1. First the X and Y registers are cleared to zero. The Y register is used as an index to the musical note data in memory starting at location $1890. To access successive bytes for frequency and duration, the Y index register is incremented. The X register is used as an index to access the low- and high-byte frequency control registers at locations $D400 and $D401. In this program segment, the X register can only have one of two values; 0 to access location $D400 and 1 to access location $D401.

The first time through the loop, the instruction starting at location $1831 (LDA $1890,Y) loads the musical data value (low byte frequency) into the accumulator. The next instruction (BEQ $1875) checks to see if this value is equal to zero. If it is, control passes to location $1875, the volume is cleared to zero, and the program ends. This suggests that to end the program, place a zero in the low byte data element. This mechanism acts as a data terminator, since no additional data is read if the low-byte value of a musical note equals zero. If the data value does not equal zero, it is stored in the low byte frequency control register for voice 1.

Both the X and Y registers are incremented, and the next musical data value, this time at $1891, is loaded into the accumulator with the instruction beginning at location $183B. The next instruction (STA $D400,X) stores the data value in the high-byte frequency control register ($D401) for voice 1.

The instruction at location $1844 increments the Y register. The next load instruction starting at location $1845 loads the accumulator with the next data element in memory, this time the low byte for the duration of the note. The duration note table is found on page 252. The low-byte duration is stored in zero page location $FA. The Y

register is incremented again and the high byte duration is loaded into the accumulator and stored in zero page location $FB. These two locations are decremented in the instructions stored in location $185B through $1862.

The instructions stored in locations $1850 through $1857 GATE the note, in other words play the note. To output audible notes, each voice has a gate bit which initiates the sound from the envelope generator for a particular voice. Bit 0 of location $D404 is the gate bit for voice 1. To gate a bit, set bit 0 and also set the bit for the desired waveform at the same time. In this case, the sawtooth waveform is gated on (32 + 1 = $21).

At this point, the sawtooth waveform is gated on and is playing in the voice 1 envelope generator. The selected frequency is output until you turn off the gate bit. Here's where the duration of the note comes into play. The instructions stored in locations $185B through $1862 decrement the low and high byte duration value stored in zero page locations $FA and $FB respectively. These instructions act as a time delay which count through the loops using the duration values from page 354 that are stored in the musical data region starting at $1890 in memory. When both $FA and $FB are equal to zero, program control drops through to the instructions stored at $1864 through $186A. These instructions clear the voice 1 gate bit and stop the envelope generator from outputting sound.

At this point, the Y register is incremented ($186E) and the program jumps to $182F where the X register is cleared to zero. Since the Y register was already incremented before the JMP instruction in location $186F–71, the low byte frequency value for the second note is already pointed to by the Y register plus the base address of $1890. The program then loads that low-byte value and stores it in the location $D400 plus the offset of the X register, in this case 0. That gives the appropriate address $D400 for the voice 1 (low byte) frequency control register. These instructions are executed repetitively until a low-byte frequency of zero is encountered. When the low-byte frequency equals zero, control is passed to location $1875, the volume is cleared to zero, and the program breaks.

This program plays 19 notes, and on the 20th a zero is detected as the low-byte frequency value (in location $18DC), so volume is set to zero and the program ends. This data is entered as an example. Place your data there instead according to the note table in Figure 11–15 at the end of the chapter and the duration table on page 354.

Expand on this example. Add multiple voices using different waveforms. Utilize the filter to perfect the quality of the musical notes. Use this program as a basis for your own full-featured musical program.

## SYNCHRONIZATION AND RING MODULATION

The 6581 SID chip lets you create more complex harmonic structures through synchronization or ring modulation of two voices.

The process of synchronization is basically a logical ANDing of two wave forms. When either is zero, the output is zero.

Here's a synchronization algorithm:

```
1    Clear sound chip registers.
2    Set high frequency voice 1.
3    Set Attack/Decay for voice 1 (A=13, D=11).
4    Set high frequency voice 3.
5    Set volume 15.
6    Set start triangle, sync waveform control for voice 1.
7    Timing loop.
8    Clear triangle, sync waveform control for voice 1.
9    Wait, then turn off volume.
```

The synchronization feature is enabled (turned on) in step 6, where bits 0, 1, and 4 of register $D404 are set. Bit 1 enables the syncing function between voice 1 and voice 3. Bits 0 and 4 have their usual functions of gating voice 1 and setting the triangular waveform.

Ring modulation (accomplished for voice 1 by setting bit 2 of register $D404 in step 6 of the algorithm) replaces the triangular output of oscillator 1 with a "ring modulated" combination of oscillators 1 and 3. This produces non-harmonic overtone structures for use in mimicking bell or gong sounds.

# SID REGISTER DESCRIPTION

Following is a register-by-register description of the SID chip in numerical order.

# VOICE 1

## FREQ LO/FREQ HI (Registers 00,01)

Together these registers form a 16-bit number which linearly controls the frequency of Oscillator 1. The frequency is determined by the following equation:

$$F_{out} = (F_n \times F_{clk}/16777216) \text{ Hz}$$

Where $F_n$ is the 16-bit number in the Frequency registers and $F_{clk}$ is the system clock applied to the ø2 input (pin 6). For a standard 1.0-MHz clock, the frequency is given by:

$$F_{out} = (F_n \times 0.06097) \text{ Hz}$$

A complete table of values for generating eight octaves of the equally tempered musical scale with concert A (440 Hz) tuning is provided in Figure 11–15. It should be noted that the frequency resolution of SID is sufficient for any tuning scale and allows sweeping from note to note (portamento) with no discernable frequency steps.

## PW LO/PW HI (Registers 02,03)

Together these registers form a 12-bit number (bits 4–7 of PW HI are not used) which linearly controls the Pulse Width (duty cycle) of the Pulse waveform on Oscillator 1. The pulse width is determined by the following equation:

$$PW_{out} = (PW_n/40.95)\ \%$$

Where $PW_n$ is the 12-bit number in the Pulse Width registers.

The pulse width resolution allows the width to be smoothly swept with no discernable stepping. Note that the Pulse waveform on Oscillator 1 must be selected in order for the Pulse Width registers to have any audible effect. A value of 0 or 4095 ($FFF) in the Pulse Width registers will produce a constant DC output, while a value of 2048 ($800) will produce a square wave.

## CONTROL REGISTER (Register 04)

This register contains eight control bits which select various options on Oscillator 1.

**GATE (Bit 0):** The GATE bit controls the Envelope Generator for Voice 1. When this bit is set to a one, the Envelope Generator is Gated (triggered) and the ATTACK/DECAY/SUSTAIN cycle is initiated. When the bit is reset to a zero, the RELEASE cycle begins. The Envelope Generator controls the amplitude of Oscillator 1 appearing at the audio output, therefore, the GATE bit must be set (along with suitable envelope parameters) for the selected output of Oscillator 1 to be audible. A detailed discussion of the Envelope Generator can be found in this book in the chapter on Hardware Overview.

**SYNC (Bit 1):** The SYNC bit, when set to a one, synchronizes the fundamental frequency of Oscillator 1 with the fundamental frequency of Oscillator 3, producing "Hard Sync" effects.

Varying the frequency of Oscillator 1 with respect to Oscillator 3 produces a wide range of complex harmonic structures from Voice 1 at the frequency of Oscillator 3. In order for sync to occur, Oscillator 3 must be set to some frequency other than zero but preferably lower than the frequency of Oscillator 1. No other parameters of Voice 3 have any effect on sync.

**RING MOD (Bit 2):** The RING MOD bit, when set to a one, replaces the Triangle waveform output of Oscillator 1 with a "Ring Modulated" combination of Oscillators 1 and 3. Varying the frequency of Oscillator 1 with respect to Oscillator 3 produces a wide range of non-harmonic overtone structures for creating bell or gong sounds and for special effects. In order for ring modulation to be audible, the Triangle waveform of Oscillator 1 must be selected and Oscillator 3 must be set to some frequency other than zero. No other parameters of Voice 3 have any effect on ring modulation.

**TEST (Bit 3):** The TEST bit, when set to a one, resets and locks Oscillator 1 at zero until the TEST bit is cleared. The Noise waveform output of Oscillator 1 is also reset and the Pulse waveform output is held at a DC level. Normally this bit is used for testing purposes, however, it can be used to synchronize Oscillator 1 to external events, allowing the generation of highly complex waveforms under real-time software control.

**(Bit 4):** When set to a one, the Triangle waveform output of Oscillator 1 is selected. The Triangle waveform is low in harmonics and has a mellow, flute-like quality.

**(Bit 5):** When set to a one, the Sawtooth waveform output of Oscillator 1 is selected. The Sawtooth waveform is rich in even and odd harmonics and has a bright, brassy quality.

**(Bit 6):** When set to a one, the Pulse waveform output of Oscillator 1 is selected. The harmonic content of this waveform can be adjusted by the Pulse Width registers, producing tone qualities ranging from a bright, hollow square wave to a nasal, reedy pulse. Sweeping the pulse width in real-time produces a dynamic "phasing" effect which adds a sense of motion to the sound. Rapidly jumping between different pulse widths can produce interesting harmonic sequences.

**NOISE (Bit 7):** When set to a one, the Noise output waveform of Oscillator 1 is selected. This output is a random signal which changes at the frequency of Oscillator 1. The sound quality can be varied from a low rumbling to hissing white noise via the Oscillator 1 Frequency registers. Noise is useful in creating explosions, gunshots, jet engines, wind, surf and other unpitched sounds, as well as snare drums and cymbals. Sweeping the oscillator frequency with Noise selected produces a dramatic rushing effect.

One of the output waveforms must be selected for Oscillator 1 to be audible, however, it is NOT necessary to de-select waveforms to silence the output of Voice 1. The amplitude of Voice 1 at the final output is a function of the Envelope Generator only.

> **NOTE:** The oscillator output waveforms are NOT additive. If more than one output waveform is selected simultaneously, the result will be a logical ANDing of the waveforms. Although this technique can be used to generate additional waveforms beyond the four listed above, it must be used with care. If any other waveform is selected while Noise is on, the Noise output can "lock up." If this occurs, the Noise output will remain silent until reset by the TEST bit or by bringing RES (pin 5) low.

## ATTACK/DECAY (Register 05)

Bits 4–7 of this register (ATK0–ATK3) select 1 of 16 ATTACK rates for the Voice 1 Envelope Generator. The ATTACK rate determines how rapidly the output of Voice 1 rises from zero to peak amplitude when the Envelope Generator is gated. The 16 ATTACK rates are listed in Figure 11–14.

Bits 0–3 (DCY0–DCY3) select 1 of 16 DECAY rates for the Envelope Generator. The DECAY cycle follows the ATTACK cycle and the DECAY rate determines how rapidly the output falls from the peak amplitude to the selected SUSTAIN level. The 16 DECAY rates are listed in Figure 11–14.

## SUSTAIN/RELEASE (Register 06)

Bits 4–7 of this register (STN0–STN3) select 1 of 16 SUSTAIN levels for the Envelope Generator. The SUSTAIN cycle follows the DECAY cycle and the output of Voice 1 will remain at the selected SUSTAIN amplitude as long as the gate bit remains set. The SUSTAIN levels range from zero to peak amplitude in sixteen linear steps, with a SUSTAIN value of 0 selecting zero amplitude and a SUSTAIN value of 15 ($F) selecting the peak amplitude. A SUSTAIN value of 8 would cause Voice 1 to SUSTAIN at an amplitude one-half the peak amplitude reached by the ATTACK cycle.

Bits 0–3 (RLS0–RLS3) select 1 of 16 RELEASE rates for the Envelope Generator. The RELEASE cycle follows the SUSTAIN cycle when the Gate bit is reset to zero. At this time, the output of Voice 1 will fall from the SUSTAIN amplitude to zero amplitude at the selected RELEASE rate. The 16 RELEASE rates are identical to the DECAY rates as listed in Figure 11–14.

**NOTE:** The cycling of the Envelope Generator can be altered at any point via the Gate bit. The Envelope Generator can be gated and released without restriction. For example, if the gate bit is reset before the envelope has finished the ATTACK cycle, the RELEASE cycle will immediately begin, starting from whatever amplitude had been reached. If the envelope is then gated again (before the RELEASE cycle has reached zero amplitude), another ATTACK cycle will begin, starting from whatever amplitude had been reached. This technique can be used to generate complex amplitude envelopes via real-time software control.

| VALUE | | ATTACK RATE | DECAY/RELEASE RATE |
|---|---|---|---|
| DEC | (HEX) | (TIME/CYCLE) | TIME/CYCLE) |
| 0 | (0) | 2 ms | 6 ms |
| 1 | (1) | 8 ms | 24 ms |
| 2 | (2) | 16 ms | 48 ms |
| 3 | (3) | 24 ms | 72 ms |
| 4 | (4) | 38 ms | 114 ms |
| 5 | (5) | 56 ms | 168 ms |
| 6 | (6) | 68 ms | 204 ms |
| 7 | (7) | 80 ms | 240 ms |
| 8 | (8) | 100 ms | 300 ms |
| 9 | (9) | 250 ms | 750 ms |
| 10 | (A) | 500 ms | 1.5 s |
| 11 | (B) | 800 ms | 2.4 s |
| 12 | (C) | 1 s | 3 s |
| 13 | (D) | 3 s | 9 s |
| 14 | (E) | 5 s | 15 s |
| 15 | (F) | 8 s | 24 s |

**NOTE:** Envelope rates are based on a 1.0-MHz ø2 clock. For other ø2 frequencies, multiply the given rate by 1 MHz/ø2. The rates refer to the amount of time per cycle. For example, given an ATTACK value of 2, the ATTACK cycle would take 16 ms to rise from zero to peak amplitude. The DECAY/RELEASE rates refer to the amount of time these cycles would take to fall from peak amplitude to zero.

**Figure 11–14. Envelope Rates**

# VOICE 2

Registers $07–$0D control Voice 2 and are functionally identical to registers $00–$06 with these exceptions:

1. When selected, SYNC synchronizes Oscillator 2 with Oscillator 1.
2. When selected, RING MOD replaces the Triangle output of Oscillator 2 with the ring modulated combination of Oscillators 2 and 1.

# VOICE 3

Registers $0E–$14 control Voice 3 and are functionally identical to registers $00–$06 with these exceptions:

1. When selected, SYNC synchronizes Oscillator 3 with Oscillator 2.
2. When selected, RING MOD replaces the Triangle output of Oscillator 3 with the ring modulated combination of Oscillators 3 and 2.

Typical operation of a voice consists of selecting the desired parameters: frequency, waveform, effects (SYNC, RING MOD) and envelope rates, then gating the voice whenever the sound is desired. The sound can be sustained for any length of time and terminated by clearing the gate bit. Each voice can be used separately, with independent parameters and gating, or in unison to create a single, powerful voice. When used in unison, a slight detuning of each oscillator or tuning to musical intervals creates a rich, animated sound.

# FILTER

## FC LO/FC HI (REGISTERS $15,$16)

Together these registers form an 11-bit number (bits 3–7 of FC LO are not used) which literally controls the Cutoff (or Center) Frequency of the programmable Filter. The approximate Cutoff Frequency ranges from 30 Hz to 12 KHz.

## RES/FILT (REGISTER $17)

Bits 4–7 of this register (RES0–RES3) control the resonance of the filter. Resonance is a peaking effect which emphasizes frequency components at the Cutoff Frequency of the Filter, causing a sharper sound. There are sixteen resonance settings ranging linearly from no resonance (0) to maximum resonance (15 or $F). Bits 0–3 determine which signals will be routed through the Filter:

**FILT 1 (Bit 0):** When set to a zero, Voice 1 appears directly at the audio output and the Filter has no effect on it. When set to a one, Voice 1 will be processed through the Filter and the harmonic content of Voice 1 will be altered according to the selected Filter parameters.

**FILT 2 (Bit 1):** Same as bit 0 for Voice 2.
**FILT 3 (Bit 2):** Same as bit 0 for Voice 3.
**FILTEX (Bit 3):** Same as bit 0 for External audio input (pin 26).

## MODE/VOL (REGISTER $18)

Bits 4–7 of this register select various Filter mode and output options:

**LP (Bit 4):** When set to a one, the Low-Pass output of the Filter is selected and sent to the audio output. For a given Filter input signal, all frequency components below the Filter Cutoff Frequency are passed unaltered, while all frequency components above the Cutoff are attenuated at a rate of 12 dB/Octave. The Low-Pass mode produces full-bodied sounds.

**BP (Bit 5):** Same as bit 4 for the Bandpass output. All frequency components above and below the Cutoff are attenuated at a rate of 6 dB/Octave. The Bandpass mode produces thin, open sounds.

**HP (Bit 6):** Same as bit 4 for the High-Pass output. All frequency components above the Cutoff are passed unaltered, while all frequency components below the Cutoff are attenuated at a rate of 12 dB/Octave. The High-Pass mode produces tinny, buzzy sounds.

**3 OFF (Bit 7):** When set to a one, the output of Voice 3 is disconnected from the direct audio path. Setting Voice 3 to bypass the Filter (FILT 3 = 0) and setting 3 OFF to a one prevents Voice 3 from reaching the audio output. This allows Voice 3 to be used for modulation purposes without any undesirable output.

> **NOTE:** The Filter output modes ARE additive and multiple Filter modes may be selected simultaneously. For example, both LP and HP modes can be selected to produce a Notch (or Band Reject) Filter response. In order for the Filter to have any audible effect, at least one Filter output must be selected and at least one Voice must be routed through the Filter. The Filter is, perhaps, the most important element in SID as it allows the generation of complex tone colors via subtractive synthesis (the Filter is used to eliminate specific frequency components from a harmonically rich input signal). The best results are achieved by varying the Cutoff Frequency in real-time.

**Bits 0–3** (VOL0–VOL3) select 1 of 16 overall Volume levels for the final composite audio output. The output volume levels range from no output (0) to maximum volume (15 or $F) in sixteen linear steps. This control can be used as a static volume control for balancing levels in multi-chip systems or for creating dynamic volume effects, such as Tremolo. Some Volume level other than zero must be selected in order for SID to produce any sound.

# MISCELLANEOUS

## POTX (REGISTER $19)

This register allows the microprocessor to read the position of the potentiometer tied to POTX (pin 24), with values ranging from 0 at minimum resistance, to 255 ($FF) at maximum resistance. The value is always valid and is updated every 512 ø2 clock cycles. See the Pin Description section for information on pot and capacitor values.

## POTY (REGISTER $1A)

Same as POTX for the pot tied to POTY (pin 23).

## OSC 3/RANDOM (REGISTER $1B)

This register allows the microprocessor to read the upper 8 output bits of Oscillator 3. The character of the numbers generated is directly related to the waveform selected. If the Sawtooth waveform of Oscillator 3 is selected, this register will present a series of numbers incrementing from 0 to 255 ($FF) at a rate determined by the frequency of Oscillator 3. If the Triangle waveform is selected, the output will increment from 0 up to 255, then decrement down to 0. If the Pulse waveform is selected, the output will jump between 0 and 255. Selecting the Noise waveform will produce a series of random numbers, therefore, this register can be used as a random number generator for games. There are numerous timing and sequencing applications for the OSC 3 register; however, the chief function is probably that of a modulation generator. The numbers generated by this register can be added, via software, to the Oscillator or Filter Frequency registers or the Pulse Width registers in real-time. Many dynamic effects can be generated in this manner. Siren-like sounds can be created by adding the OSC 3 Sawtooth output to the frequency control of another oscillator. Synthesizer "Sample and Hold" effects can be produced by adding the OSC 3 Noise output to the Filter Frequency control registers. Vibrato can be produced by setting Oscillator 3 to a frequency around 7 Hz and adding the OSC 3 Triangle output (with proper scaling) to the Frequency control of another oscillator. An unlimited range of effects are available by altering the frequency of Oscillator 3 and scaling the OSC 3 output. Normally, when Oscillator 3 is used for modulation, the audio output of Voice 3 should be eliminated (3 OFF = 1).

## ENV 3 (Register $1C)

Same as OSC 3, but this register allows the microprocessor to read the output of the Voice 3 Envelope Generator. This output can be added to the Filter Frequency to produce harmonic envelopes, WAH-WAH, and similar effects. "Phaser" sounds can be created by adding this output to the frequency control registers of an oscillator. The Voice 3 Envelope Generator must be Gated in order to produce any output from this register. The OSC 3 register, however, always reflects the changing output of the oscillator and is not affected in any way by the Envelope Generator.

# EQUAL-TEMPERED MUSICAL SCALE VALUES

The table in Figure 11–15 lists the numerical values which must be stored in the SID Oscillator frequency control registers to produce the notes of the equal-tempered musical scale. The equal-tempered scale consists of an octave containing twelve semitones (notes): C,D,E,F,G,A,B and C#,D#,F#,G#,A#. The frequency of each semitone is exactly the 12th root of 2 ($\sqrt[12]{2}$) times the frequency of the previous semitone. The table is based on a ø2 clock of 1.02 MHz. Refer to the equation given in the Register Description for use of other master clock frequencies. The scale selected is concert pitch, in which A–4 = 440 Hz. Transpositions of this scale and scales other than the equal-tempered scale are also possible.

Although the table in Figure 11–15 provides a simple and quick method for generating the equal-tempered scale, it is very memory inefficient as it requires 192 bytes for the table alone. Memory efficiency can be improved by determining the note value algorithmically. Using the fact that each note in an octave is exactly half the frequency of that note in the next octave, the note look-up table can be reduced from ninety-six entries to twelve entries, as there are twelve notes per octave. If the twelve entries (24 bytes) consist of the 16-bit values for the eighth octave (C-7 through B-7), then notes in lower octaves can be derived by choosing the appropriate note in the eighth octave and dividing the 16-bit value by two for each octave of difference. As division by two is nothing more than a right-shift of the value, the calculation can easily be accomplished by a simple software routine. Although note B-7 is beyond the range of the oscillators, this value should still be included in the table for calculation purposes (the MSB of B-7 would require a special software case, such as generating this bit in the CARRY before shifting). Each note must be specified in a form which indicates which of the twelve semitones is desired, and which of the eight octaves the semitone is in. Since 4 bits are necessary to select one of twelve semitones and 3 bits are necessary to select one of eight octaves, the information can fit in one byte, with the lower nybble selecting the semitone (by addressing the look-up table) and the upper nybble being used by the division routine to determine how many times the table value must be right-shifted.

# MUSIC NOTE VALUES

Figure 11–15 contains a complete list of Note#, actual note, and the values to be stored into the HI FREQ and LOW FREQ registers of the sound chip to produce the indicated note.

**Figure 11–15. Music Note Values**

| MUSICAL NOTE | | OSCILLATOR FREQ | | |
|---|---|---|---|---|
| NOTE | OCTAVE | DECIMAL | HI | LOW |
| 0 | C–0 | 268 | 1 | 12 |
| 1 | C#–0 | 284 | 1 | 28 |
| 2 | D–0 | 301 | 1 | 45 |
| 3 | D#–0 | 318 | 1 | 62 |
| 4 | E–0 | 337 | 1 | 81 |
| 5 | F–0 | 358 | 1 | 102 |
| 6 | F#–0 | 379 | 1 | 123 |
| 7 | G–0 | 401 | 1 | 145 |
| 8 | G#–0 | 425 | 1 | 169 |
| 9 | A–0 | 451 | 1 | 195 |
| 10 | A#–0 | 477 | 1 | 221 |
| 11 | B–0 | 506 | 1 | 250 |
| 16 | C–1 | 536 | 2 | 24 |
| 17 | C#–1 | 568 | 2 | 56 |
| 18 | D–1 | 602 | 2 | 90 |
| 19 | D#–1 | 637 | 2 | 125 |
| 20 | E–1 | 675 | 2 | 163 |
| 21 | F–1 | 716 | 2 | 204 |
| 22 | F#–1 | 758 | 2 | 246 |
| 23 | G–1 | 803 | 3 | 35 |
| 24 | G#–1 | 851 | 3 | 83 |
| 25 | A–1 | 902 | 3 | 134 |
| 26 | A#–1 | 955 | 3 | 187 |
| 27 | B–1 | 1012 | 3 | 244 |
| 32 | C–2 | 1072 | 4 | 48 |
| 33 | C#–2 | 1136 | 4 | 112 |
| 34 | D–2 | 1204 | 4 | 180 |
| 35 | D#–2 | 1275 | 4 | 251 |
| 36 | E–2 | 1351 | 5 | 71 |
| 37 | F–2 | 1432 | 5 | 152 |
| 38 | F#–2 | 1517 | 5 | 237 |
| 39 | G–2 | 1607 | 6 | 71 |
| 40 | G#–2 | 1703 | 6 | 167 |
| 41 | A–2 | 1804 | 7 | 12 |
| 42 | A#–2 | 1911 | 7 | 119 |
| 43 | B–2 | 2025 | 7 | 233 |
| 48 | C–3 | 2145 | 8 | 97 |
| 49 | C#–3 | 2273 | 8 | 225 |
| 50 | D–3 | 2408 | 9 | 104 |
| 51 | D#–3 | 2551 | 9 | 247 |
| 52 | E–3 | 2703 | 10 | 143 |
| 53 | F–3 | 2864 | 11 | 48 |
| 54 | F#–3 | 3034 | 11 | 218 |
| 55 | G–3 | 3215 | 12 | 143 |
| 56 | G#–3 | 3406 | 13 | 78 |

**Figure 11–15. Music Note Values (continued)**

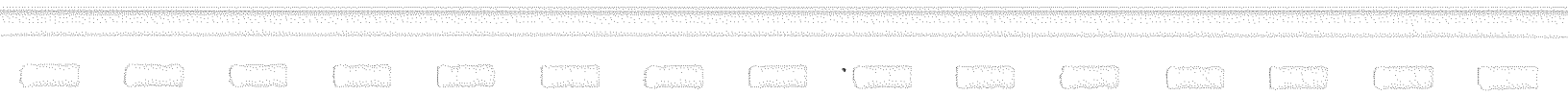
| MUSICAL NOTE | | OSCILLATOR FREQ | | |
|---|---|---|---|---|
| NOTE | OCTAVE | DECIMAL | HI | LOW |
| 57 | A–3 | 3608 | 14 | 24 |
| 58 | A#–3 | 3823 | 14 | 239 |
| 59 | B–3 | 4050 | 15 | 210 |
| 64 | C–4 | 4291 | 16 | 195 |
| 65 | C#–4 | 4547 | 17 | 195 |
| 66 | D–4 | 4817 | 18 | 209 |
| 67 | D#–4 | 5103 | 19 | 239 |
| 68 | E–4 | 5407 | 21 | 31 |
| 69 | F–4 | 5728 | 22 | 96 |
| 70 | F#–4 | 6069 | 23 | 181 |
| 71 | G–4 | 6430 | 25 | 30 |
| 72 | G#–4 | 6812 | 26 | 156 |
| 73 | A–4 | 7217 | 28 | 49 |
| 74 | A#–4 | 7647 | 29 | 223 |
| 75 | B–4 | 8101 | 31 | 165 |
| 80 | C–5 | 8583 | 33 | 135 |
| 81 | C#–5 | 9094 | 35 | 134 |
| 82 | C–0 | 9634 | 37 | 162 |
| 83 | C#–0 | 10207 | 39 | 223 |
| 84 | D–0 | 10814 | 42 | 62 |
| 85 | F–5 | 11457 | 44 | 193 |
| 86 | F#–5 | 12139 | 47 | 107 |
| 87 | G–5 | 12860 | 50 | 60 |
| 88 | G#–5 | 13625 | 53 | 57 |
| 89 | A–5 | 14435 | 56 | 99 |
| 90 | A#–5 | 15294 | 59 | 190 |
| 91 | B–5 | 16203 | 63 | 75 |
| 96 | C–6 | 17167 | 67 | 15 |
| 97 | C#–6 | 18188 | 71 | 12 |
| 98 | D–6 | 19269 | 75 | 69 |
| 99 | D#–6 | 20415 | 79 | 191 |
| 100 | E–6 | 21629 | 84 | 125 |
| 101 | F–6 | 22915 | 89 | 131 |
| 102 | F#–6 | 24278 | 94 | 214 |
| 103 | G–6 | 25721 | 100 | 121 |
| 104 | G#–6 | 27251 | 106 | 115 |
| 105 | A–6 | 28871 | 112 | 199 |
| 106 | A#–6 | 30588 | 119 | 124 |
| 107 | B–6 | 32407 | 126 | 151 |
| 112 | C–7 | 34334 | 134 | 30 |
| 113 | C#–7 | 36376 | 142 | 24 |
| 114 | D–7 | 38539 | 150 | 139 |
| 115 | D#–7 | 40830 | 159 | 126 |
| 116 | E–7 | 43258 | 168 | 250 |

| MUSICAL NOTE | | OSCILLATOR FREQ | | |
|---|---|---|---|---|
| NOTE | OCTAVE | DECIMAL | HI | LOW |
| 117 | F–7 | 45830 | 179 | 6 |
| 118 | F#–7 | 48556 | 189 | 172 |
| 119 | G–7 | 51443 | 200 | 243 |
| 120 | G#–7 | 54502 | 212 | 230 |
| 121 | A–7 | 57743 | 225 | 143 |
| 122 | A#–7 | 61176 | 238 | 248 |
| 123 | B–7 | 64814 | 253 | 46 |

**Figure 11–15. Music Note Values (continued)**

| FILTER SETTINGS | |
|---|---|
| LOCATION | CONTENTS |
| 54293 | Low cutoff frequency (0–7) |
| 54294 | High cutoff frequency (0–255) |
| 54295 | Resonance (bits 4–7)<br>Filter voice 3 (bit 2)<br>Filter voice 2 (bit 1)<br>Filter voice 1 (bit 0) |
| 54296 | High pass (bit 6)<br>Bandpass (bit 5)<br>Low pass (bit 4)<br>Volume (bits 0–3) |

**Figure 11–16. Filter Settings**



# 12

# INPUT/OUTPUT GUIDE

# INTRODUCTION

Besides being a calculator and a data manipulator, the Commodore 128 is a communicator. Through the various connecting ports, it can transmit and receive information to many different kinds of peripheral equipment. Because of the variety of devices, each must have its unique access codes and specific instructions (software). This chapter includes the details of each connection and the associated technique of communicating with various devices.

## INPUT/OUTPUT PORTS

Most of the connections are considered as input and output because information is transmitted in both directions. These connections include the Commodore Serial Port for the disk drive(s) and printer(s), the User Port for external communication with modems, an expansion port for additional memory, a Datassette connector for a tape storage device, two monitor ports for both the composite video and the RGBI monitors, a TV connector, and game ports for various control devices such as joysticks, paddles and graphics tablets. The port pinout details begin on page 394.

# DISK DRIVE

The disk drive is the most popular information storage device today. It stores programs and data as magnetic signals on a floppy disk of flexible plastic coated with iron oxide. Information (characters, letters and numbers) represented by a series of binary digits is magnetically impressed upon the disk in one of thirty-five concentric ring tracks. Each ring is divided into 17 to 21 sectors each. Each ring can store about 5000 bytes per track. A 1541 disk drive has a capacity of about 170K. The 1571 disk drive is double-sided, so a disk formatted on the 1571 holds twice as much data, about 340K.

## FORMATTING A DISK

Blank disks from the factory are not assigned tracks or sectors. A disk must be formatted by your computer/disk drive according to the disk operating system. You *cannot* store anything on a brand-new disk until this process, called *Formatting* or *newing*, is completed. The Commodore disk drive has built-in microprocessors that properly organize a blank disk when instructed to do so by the computer. The eighteenth track is used as a directory for the disk drive. All the addresses for all the blocks (over 600) are stored on track 18 as the Block Allocation Map, or more simply the directory.

Prior to formatting a disk, decide on an identification name and a two-digit code. Select a name of up to 16 characters that will help identify your disk. For example, the disk name could be your name, or a topic such as games. The two-digit code can be almost any letter-number combination. This code immediately alerts the disk drive whenever

you change disks—so, for your own protection, try to assign each disk a unique code.

The format command for C128 mode is as follows:

HEADER "disk name", Ixy,D0,ON U8

For C64 or C128 modes, use:

OPEN 15,8,15, "N0:diskname, xy"

Note that xy is any two-digit code you select, such as 01, 45, W3, 40, or WD. The 0 after the D or N represents the default drive number for a single disk drive or the first drive of a dual drive unit where both share a common device number. The 8 in each statement represents the device number. A second drive would be set to number 9, a third drive to 10, etc. With a single drive, the HEADER command can simply be:

HEADER"disk name",Ixy

The formatting process typically takes about 80 seconds on the 1541 disk drive, and considerably less on the 1571 drive. Once this operation is completed, this disk can be used for storing programs and data. Do *not* format it again—otherwise everything saved on it will be erased. To simply "wipe" an already formatted disk, use the above instructions but do not include the comma followed by an identification number. This process will maintain the original format, use the new name and wipe the directory clean—a process considerably faster than total formatting.

The OPEN command establishes a communication link between the computer and a specific device number. Disk drives are factory set as device number 8 and can be converted to device 9, 10, 11 or 12. See your disk drive manual for details.

To verify you have properly formatted the disk, see the section below on reading the directory.

## SAVING PROGRAMS

The saving process for storing programs on a disk does *not* remove the program from the computer. It simply "copies" the program or file from the computer onto the disk in the disk drive. Nothing is lost in the process. BASIC programs are saved to a formatted disk with the following commands:

| | |
|---|---|
| C128 or C64 | SAVE"program name",8 |
| C128 only | DSAVE"program name" |
| | DSAVE"program name",D0,U9 |

(if the drive is device number 9).

In C128 mode, machine language programs or binary files can be saved through these BASIC commands:

BSAVE"filename",Bb,Ps TO Pe
or BSAVE"filename",D0,U9,Bb,Ps TO Pe
(if the drive is device number 9).

In these formats:

"b" is the memory bank number (0 through 15)
"s" is the starting address
"e" is the ending address

**EXAMPLE:**

BSAVE"filename",B2,P3584 TO P4096

This saves the binary file named "filename" from memory bank 2, in the range of memory between 3584 to 4096 decimal.

The following command saves a BASIC program in either C128 or C64 modes:

SAVE "filename",8

where 8 represents the device number.

## REPLACING A FILE OR PROGRAM

When re-saving a program under the same filename, you *must* use the at sign (@) in the SAVE command format, as follows:

In BASIC 7.0: DSAVE"@program name"
In BASIC 2.0: SAVE"@0:program name",8

The @ symbol tells the computer to store this program in place of the one of the same name. Without this symbol, the disk drive will not allow the new program to be stored. Note that during the replace process, the new program is saved first, then the old one is automatically deleted and the directory updated. Use **VERIFY** to make sure you are saving correctly.

## VERIFYING A PROGRAM OR FILE

You may want to verify that a program was accurately saved on the disk. The VERIFY command compares the program in memory with the program on the disk. If the programs are identical byte for byte, an OK message is displayed.

In BASIC 7.0, use the following:

DVERIFY"filename"

or

DVERIFY"filename",D0,U9

The first command above verifies a program using a single disk drive assigned as Unit 8, and the second command above verifies a program using a single disk drive assigned as unit 9.

In 2.0 BASIC, use:

VERIFY"filename",8

or VERIFY"filename",8,1

(to verify the program from the memory location from which it was SAVEd, primarily machine language programs).

After your program has been saved and verified, it is stored on your disk for future use.

## THE DIRECTORY

Once a disk is formatted, it has a directory, or catalog listing of the files stored on the disk. Prior to loading a program, you may want to view the directory to find out exactly what is stored on the disk. When you call for the directory, the screen displays the disk name and identification number followed by a list of programs or files presently stored. Of course, a newly formatted disk will display only its name and number. To see the directory, use the commands listed below.

In BASIC 7.0, use the following:

DIRECTORY
or CATALOG
or CATALOG D0,U9
or DIRECTORY D0,U9 (for a dual disk drive, device number 9)

The program in memory is not affected by this command.

In BASIC 2.0, use the following command:

LOAD"$0",8:LIST

This command erases the resident program. The 0 is optional with a single-drive unit.

To print a hard copy of a directory, you must use the BASIC 2.0 method. You must first open a channel to the printer. This is covered below in the section Output to a Printer.

## RETRIEVING PROGRAMS OR FILES FROM DISKS

Once you have files or programs stored on a disk, you retrieve them with an appropriate version of the LOAD command.

In BASIC 7.0, use the following command:

DLOAD"filename"

or DLOAD"filename",D0,U9
(for disk drive unit 9, drive 0)

To retrieve a program or binary file saved by the BSAVE command or by the machine language monitor, use the following BASIC 7.0 command:

BLOAD"filename",Bb,Ps

or BLOAD"filename",D0,U9,Bb,Ps
(for disk drive unit 9, drive 0)

where b is the bank number and s is the starting address.

In BASIC 2.0, use the following command formats:

To load a BASIC program: LOAD"filename",8
To load a machine language program: LOAD"filename",8,1

The relocate flag is either a 1 or a 0, which is the default value. The 1 tells the computer to LOAD from the point where it was SAVEd.

## CREATING AND STORING FILES

A *file* is a series of related data: characters, numbers and letters. Anything stored on the disk is considered a file. A *program* is a specific type of file that can be executed or run. Data files are files that contain raw data used by a program. These files are classified as either *sequential, relative, user* or *random*. The directory lists all the files and indicates the type of file: PRG for program, SEQ for sequential file, REL for relative file and USR for user files.

A program is a file that contains operating instructions for the computer. A sequential file is a series of data stored in sequence and can be read only in sequential order. A relative file is a file stored in sequential order and can be accessed directly at any location using a pointer.

The creation of a sequential file is initiated with an OPEN statement. A file is OPENed with the following information:

- File number: 0 through 255; 15 is reserved for special commands and error checking.
- Device number: default is 8 for a disk drive; 4 for a printer (4–30 are available for serial bus devices).
- Channel number: 0 through 255; the line of communication established between devices (also known as secondary address)

Additional (optional) information may also be specified with quotes:

- Drive number followed by a colon: default is 0 for a single drive; 1 may be specified on a dual drive.
- Filename: the name of the file, up to 16 characters.
- Filetype: SEQ = sequential, PRG = program, REL = relative, USR = user
- Operation: (R = read, W = write)

The following command:

OPEN 111, 8, 12,"0:NAME, S,W"

opens file number 111 on device number 8, using channel 12. The file is opened to default drive 0, with the filename "NAME," as a sequential file. The W specifies that a write operation will take place. To store data, it is "written" on the disk. Data is retrieved by "reading" the disk.

Enter, save and run the following programs as sample routines to create, save and read a sequential file. See your disk drive manual for more information.

```
10  REM STORE DATA
11  REM NOTE PROGRAM STOPS WHEN YOU RESPOND WITH "END".
20  OPEN 5,8,5,"EXAMPLE,S,W"
25  DO
30  INPUT "NAME OF A FRIEND";N$
40  PRINT#5,N$
50  IF N$="END"THEN CLOSE 5:END
60  LOOP
```

> **NOTE:** In C128 mode, the Commodore 128 has a set of disk input/output commands. These include the DLOAD and DSAVE commands already described, as well as the DOPEN, DCLOSE and other commands. See the BASIC 7.0 Encyclopedia in Chapter 2 or your disk drive manual for details on any of these commands.

Line 20 tells the disk drive to open a sequential file for writing data. Line 30 obtains the information from the keyboard by requesting it on the screen. The PRINT# statement in line 40 specifies the file number where the information is to be written. Line 40 is the statement that actually writes the data to the channel and creates the contents of the file. Line 50 also puts a recognizable end-of-file marker—the word "end" at the end of the file. Line 50 closes the file after all data has been written.

Here's a program that reads the data from the sequential file you just created:

```
10  REM RETRIEVE FILES
20  OPEN 6,8,6,"EXAMPLE,S,R"
25  DO
30  INPUT#6,L$
40  IF L$="END"THEN CLOSE6:END
50  PRINT L$
60  LOOP
```

Line 20 opens a channel to the sequential file called EXAMPLE for reading. Once the end item is found, the file is properly closed. As each item is retrieved using the INPUT# statement, it is printed to the screen with a simple PRINT statement. Any sequential file in which all writing has been completed and which is already closed can be opened only as a read file; it cannot be written to unless an APPEND operation is specified or the @ prefix is used.

Files can be expanded in size with the BASIC 7.0 APPEND command as follows:

APPEND#7,"EXAMPLE"

This command opens the file called "EXAMPLE." Any data sent to it in a PRINT#7 statement will be appended. Any file number may be specified as long as the PRINT file number matches the APPEND file number.

Once all the data is sent, the channel is closed with

DCLOSE#7

In BASIC 2.0, use this command:

OPEN7,8,7, "0:EXAMPLE,A"

The letter A allows subsequent PRINT# statements to enter data at the end of the file "Example." This BASIC 2.0 technique also requires a CLOSE statement: CLOSE7.

## OTHER FILES

C128 BASIC includes several direct disk drive housekeeping commands including **SCRATCH, RENAME, CONCAT, COPY** and **COLLECT** as well as a relative file command **RECORD.** Note that these operations are also available in the C64 mode if combined with an OPEN statement on channel 15. See your C128 System Guide Encyclopedia or disk drive manual for details.

In addition, see your disk drive manual for details on relative files, random files and direct program access to the disk drive microprocessor via channel 15.

## CHANGING THE DISK DRIVE DEVICE NUMBER

The disk drive device number is usually 8, but it can be changed to be device 9, 10, or 11 by either a hardware or software change. The following command changes your drive to unit 9 through software until power is turned off:

```
OPEN15,8,15:PRINT#15,"M-W"CHR$(119)CHR$(0)CHR$(2)
CHR$(41)CHR$(73):CLOSE15
```

The 1571 has a device number switch on the back.

# OUTPUT TO A PRINTER

Your Commodore 128 computer transmits data over the serial port to many different styles of printers. The most popular Commodore printers today are the daisy wheel and dot matrix types. The daisy wheel system uses metal or plastic molded letters similar to those on a typewriter and produces the finest impressions. The fingers that hold these characters are mounted on a center post, and the wheel resembles a daisy flower. The printer head spins the wheel until the correct letter is in position behind a hammer, which presses it against a ribbon onto the paper.

The more versatile system is the dot matrix. Using a column of metal pins, a

series of dots is impressed on paper through an inked ribbon as the printer head moves across the page. The pattern of dots is based upon the binary code of each character. The infinite number of potential dot patterns makes it possible to print any style character imaginable from italics to Japanese depending only upon the printer's internal software (firmware).

Although many printer manufacturers claim their system is compatible with the Commodore 128 computer, if Commodore does not authorize the use, a special printer interface may be required. Printers cited as Centronics parallel or Serial do not directly match any of the Commodore computer ports. Commodore does not support any third-party printers or printer interfaces. The Commodore Serial Port is unique and not the same as the RS232 Serial Port, which many printer manufacturers specify.

The Commodore computer can display 512 characters. The second set of 256 includes both upper and lower case letters; the first includes upper case only with graphics. This is possible by using identical codes accessed separately. Consequently, only Commodore dot matrix printers are preprogrammed to print all the available Commodore characters. Owing to these differences, Commodore's ASCII codes are not identical to the true ASCII standard. Therefore, special modifications are required when interfacing with non-Commodore equipment.

In order to operate a printer, a line of communication with the computer must be opened using the OPEN statement. The printer is usually device number 4, though often switch selectable to 5. The command:

```
OPEN 3,4
```

establishes a link to the printer as device number 4. The first number can be any number between 1 and 255. Numbers beyond 128 force a line feed after a carriage return.

## PRINTING A PROGRAM LISTING

The following commands will produce a printed listing:

```
OPEN3,4:CMD3:LIST
```

The CMD command redirects all screen output to the printer. Any subsequent output is sent to the printer until turned off with:

```
PRINT#3:CLOSE3
```

These statements are necessary to clear the channel and ensure that all commands have been properly sent to the printer.

> **NOTE:** To print a listing with double spacing, a line feed can be forced using a file number greater than 127 such as:

```
OPEN131,4:CMD131:LIST:PRINT#131:CLOSE131
```

## PRINTING A TYPED LINE

A PRINT# command to an opened print file number outputs the associated data to the printer. The following commands output whatever is typed:

```
OPEN3,4:PRINT#3,"TYPE ANYTHING"
```

After **RETURN** is depressed, the line is sent to the printer and anything under quotes is printed on paper.

## PRINTING THROUGH A PROGRAM

Instructions to a printer require a **PRINT#** statement and can be mixed with the PRINT statement for video display. The following program prints five lines of text for each PRINT# statement while using the screen display to obtain the input:

```
10  OPEN 3,4,7
20  DO UNTIL L=5
25  IF L=0THEN30 :ELSE INPUTA$:GOTO40
30  INPUT"PLEASE TYPE";A$
40  PRINT#3,A$
50  L=L+1
60  LOOP
70  CLOSE3
```

This is the beginning of a simple word processor and could be expanded into a complex system through printer control.

## CONTROLLING YOUR PRINTER

Each printer capability (such as lower case characters, line feed, double width characters, reverse characters, dot graphics and special print fonts) is controlled by sending the proper control code from the computer. Depending upon the printer being used, there are two ways of sending code—either via the CHR$ function or the secondary address.

Standard codes are sent via the BASIC CHR$(X) function. ASCII codes under 27 are generally reserved for printer codes (such as 14 for the double width feature). Codes over 27 represent specific keyboard characters. Printer control codes over 27 (e.g., 145) are preceded with the ESCape code 27. Some systems will accept CHR$(27) followed by either CHR$(X) or the equivalent character within quotes. Thus, boldface type represented by Code 33 could be turned on with:

```
    PRINT#3, CHR$(27)CHR$(33)
or  PRINT#3, CHR$(27)"!"
```

and turned off with:

```
PRINT#3, CHR$ (27)CHR$(34)
```

assuming channel 3 was previously opened.

Many word processors can accommodate special printer controls with at least one of these methods. Refer to your printer and word processing manuals for the technique of sending control characters to your printer.

The alternate method of sending control codes to a printer is via the secondary address in the OPEN statement. An OPEN statement without a secondary address defaults to 0. Hence OPEN3,4 is the same as OPEN3,4,0. Generally a secondary address of 7 will cause the printer to print with upper case and lower case characters:

```
OPEN3,4,7
```

Other secondary addresses are often available and will perform various duties in accordance with your printer's design. See your manual for specifics. Most word processors allow the use of secondary addresses as embedded codes.

The following program will open a channel to a sequential file on disk, read data and output to a printer:

```
10  OPEN 3,8,3,''EXAMPLE,S,R''
20  OPEN 4,4,0
25  DO
30  INPUT#3,L$
40  IF L$=''END''THEN80ELSE50
50  PRINT#4,L$
60  PRINT L$
70  LOOP
80  CLOSE4:DCLOSE3:END
```

# OUTPUT TO A MODEM

A *modem* is a device used to transmit and receive data across telephone lines.

Generally, a computer uses an RS232 port to communicate with the external world. Commodore's RS232C Port (also known as the User Port) is a variation of the RS232 standard port. The RS232 has 0 to 5 TTL voltage levels rather than the standard plus/minus 12 volts.

The OPEN statement for the modem (device 2) sets up the parameters to match the speed (baud rate) and format (number of data and stop bits) of the host computer by sending a Control Register code. A second required code, the Command Register code, defines the parity, duplex and handshake. For example:

```
OPEN3,2,0,CHR$(6)CHR$(112)
```

opens a channel to the User Port (device #2) and establishes a 300-baud rate, 8-bit word, single stop bit (Control Register (6)), even parity and half duplex (Command Register (112)). See Figure 12–1, the Control Register Map, and Figure 12–2, the Command Register Map.

# THE RS-232 CHANNEL

The RS-232 capabilities can be accessed from either BASIC or machine language. The BASIC commands **INPUT#** and **GET#** are used to fetch data from the receiving buffer while **PRINT#** and **CMD** place data into the transmitting buffer.

The RS-232 Kernal routines run under the control of the 6526 CIA #2 device timers and interrupts. The 6526 chip generates NMI (Non-Maskable Interrupt) requests for RS-232 processing. This allows background RS-232 processing to take place during BASIC and machine language programs. There are built-in hold-offs in the Kernal, cassette and serial bus routines to prevent the disruption of data storage or transmission by the NMIs that are generated by the RS-232 routines. During cassette or serial bus activities, data can *not* be received from RS-232 devices. But because these hold-offs are only local, (assuming you're careful about your programming) no interference should result.

There are two buffers in the Commodore 128 memory to help prevent the loss of data when transmitting or receiving RS-232 information.

The Commodore 128 RS-232 I/O buffers consist of two first-in/first-out (FIFO) buffers, each 256 bytes long. In C128 mode, the RS-232 I/O buffers are permanently allocated. The input buffer is located between 3072 and 3327 ($0C00-$0CFF). The output buffer is located between 3328 and 3583 ($0D00-$0DFF). You cannot harm program storage by opening and closing the RS-232 channel as can happen with the Commodore 64. By permanently allocating the buffers in a fixed location, no "OUT OF MEMORY" error can occur, and your program in memory will not get "stepped on."

# OPENING THE RS-232 CHANNEL

Only one RS-232 channel should be open at any time; a second OPEN statement will cause the buffer pointers to be reset. Any characters in either the transmit buffer or the receive buffer will be lost.

Up to 4 characters can be sent in the filename field. The first two are the control and command register characters; the other two are used for user-defined, non-standard baud rates. Baud rate, parity and other options can be selected through this feature.

No error checking is done on the control word to detect a nonimplemented baud rate. Any illegal control word will cause the system output to operate at a very slow rate (below 50 baud).

## BASIC SYNTAX

OPEN lfn,2,0"<control reg.> <command reg.> <opt baud low> <opt baud high>"

The logical file number (lfn) then can be any number from 1 through 255. But be aware of the fact that if you choose a logical file number that is greater than 127, then a line feed will follow all carriage returns.

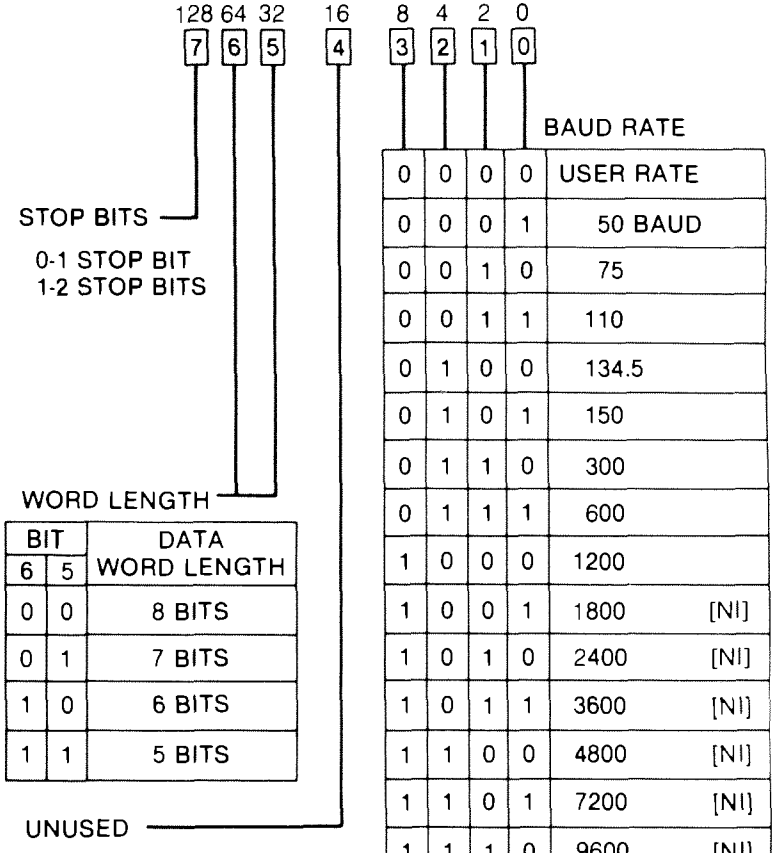
## THE CONTROL REGISTER

The **Control Register** is a single-byte required to specify the baud rates and other RS-232

options. Select the parameters you want from the Control Register Map (Figure 12-1) by placing the proper bit value in binary form. A selection, for example, from left to right of two stop bits, 7-bit word and 1200-baud rate yields:

1010 1000 (in binary), which is decimal 168 ($A8) and results in
OPEN 3,2,0,CHR$(168)

defining these parameters.




| BIT 6 | BIT 5 | DATA WORD LENGTH |
|---|---|---|
| 0 | 0 | 8 BITS |
| 0 | 1 | 7 BITS |
| 1 | 0 | 6 BITS |
| 1 | 1 | 5 BITS |

| | | | | | |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | USER RATE | |
| 0 | 0 | 0 | 1 | 50 BAUD | |
| 0 | 0 | 1 | 0 | 75 | |
| 0 | 0 | 1 | 1 | 110 | |
| 0 | 1 | 0 | 0 | 134.5 | |
| 0 | 1 | 0 | 1 | 150 | |
| 0 | 1 | 1 | 0 | 300 | |
| 0 | 1 | 1 | 1 | 600 | |
| 1 | 0 | 0 | 0 | 1200 | |
| 1 | 0 | 0 | 1 | 1800 | [NI] |
| 1 | 0 | 1 | 0 | 2400 | [NI] |
| 1 | 0 | 1 | 1 | 3600 | [NI] |
| 1 | 1 | 0 | 0 | 4800 | [NI] |
| 1 | 1 | 0 | 1 | 7200 | [NI] |
| 1 | 1 | 1 | 0 | 9600 | [NI] |
| 1 | 1 | 1 | 1 | 19200 | [NI] |

**Figure 12–1. Control Register Map**

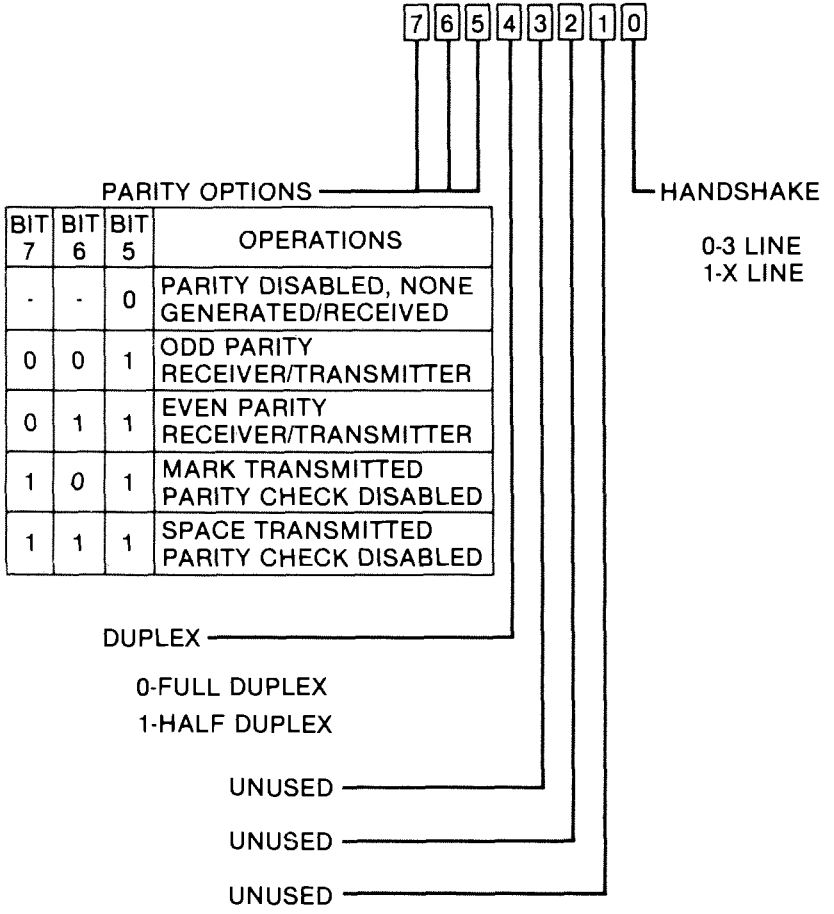
## THE COMMAND REGISTER

The **Command Register** is a single byte that defines parity, duplex and handshaking. Select the parameters from the Command Register Map in Figure 12–2 by creating the binary code and converting it to a decimal value. Thus:

OPEN3,4,0,CHR$(168)CHR$(32)

specifies odd parity, full duplex. A baud rate other than those listed in the Control Register Map is also selectable by choosing 0000 for bits 0 through 3 in the Control Register and including this rate in the OPEN statement as two bytes following the Command Register.

The "opt baud low" and high byte values are calculated from the desired baud rate as a function of the system frequency (1.022727E6 for the USA NTSC system or 0.985248E6 PAL European system). Divide the frequency by the desired rate. Divide this in half and subtract 100. This represents a two-byte decimal number. The high value is the number of times it can be divided by 256. The remainder is the low byte.




| BIT 7 | BIT 6 | BIT 5 | OPERATIONS |
|---|---|---|---|
| - | - | 0 | PARITY DISABLED, NONE GENERATED/RECEIVED |
| 0 | 0 | 1 | ODD PARITY RECEIVER/TRANSMITTER |
| 0 | 1 | 1 | EVEN PARITY RECEIVER/TRANSMITTER |
| 1 | 0 | 1 | MARK TRANSMITTED PARITY CHECK DISABLED |
| 1 | 1 | 1 | SPACE TRANSMITTED PARITY CHECK DISABLED |

**Figure 12–2. Command Register Map**

**EXAMPLE:**

1000 Baud: Divide the system frequency by the baud rate:
1022727/1000 = 1022.727

Divide by 2 and subtract 100: 1022.727/2 – 100 = 411.3635 (or rounded up to 412)

Convert this into two byte values: Division by 256 results in 1 with a remainder of 156. The following statement therefore defines a 1000-baud, 7-bit, single stop bit, odd parity, full duplex:

OPEN3,2,0,CHR$(32)CHR$(32)CHR$(156)CHR$(1)

where the Control Register value in binary is 00100000 and the Command Register in binary is 00100000.

> **NOTE:** In a C64 BASIC program, the RS-232 OPEN command should be performed before creating any variables or arrays because an automatic CLR is performed when an RS-232 channel is OPENed owing to the allocation of 512 bytes at the top of memory. This is not the case in C128 mode, since the RS-232 I/O buffers are permanently allocated.

## GETTING DATA FROM THE RS-232 CHANNEL

The Commodore 128 receiver buffer ($0C00–$0CFF) will hold up to 255 characters before overflowing. This is indicated in the RS-232 status word **ST.** If an overflow occurs, subsequent characters are lost. Keep the buffer as clear as possible. To receive data at high speeds, use machine language routines or use the FAST command to process at 2MHz with the 80-column screen (in C128 mode only).

Use GET# with the file number corresponding to the OPEN statement. Machine code Kernal entries include BASIN and GETIN.

## SENDING DATA TO THE RS-232 CHANNEL

When sending data, the output buffer ($0D00–$0DFF) can hold 255 characters before a full buffer hold-off occurs. The system will wait in the CHROUT routine until transmission is allowed or the **RUN/STOP** and **RESTORE** keys are used to recover the system.

Use CMD or PRINT# with the proper file number to send data. Acceptable Kernal entries include BSOUT (CHROUT).

## CLOSING THE RS-232 CHANNEL

Use **CLOSE** followed by the file number to close the RS-232 channel. Closing the file discards all data in the buffers at the time of execution, stops all transmitting and receiving and sets the RTS and transmitted data lines high. In C64 mode (only) CLOSE removes both buffers.

**NOTE:** Care should be taken to ensure all data is transmitted before closing the channel. A way to check this from BASIC is:

```
10 DO
20 LOOP WHILE PEEK (2575) and 3
30 CLOSE lfn
```

Location 2575 is the RS-232 "ENABL" location.

Table 12–1 defines the User Port RS-232 channel lines.

(6526 DEVICE #2 Loc. $DD00–$DD0F)

| PIN ID | 6526 ID | DESCRIPTION | EIA | ABV | IN/ OUT | MODES |
|---|---|---|---|---|---|---|
| C | PB0 | Received Data | (BB) | $S_{in}$ | IN | 1 2 |
| D | PB1 | Request to Send | (CA) | RTS | OUT | 1*2 |
| E | PB2 | Data Terminal Ready | (CD) | DTR | OUT | 1*2 |
| F | PB3 | Ring Indicator | (CE) | RI | IN | 3 |
| H | PB4 | Received Line Signal | (CF) | DCD | IN | 2 |
| J | PB5 | Unassigned | ( ) | XXX | IN | 3 |
| K | PB6 | Clear to Send | (CB) | CTS | IN | 2 |
| L | PB7 | Data Set Ready | (CC) | DSR | IN | 2 |
| B | FLAG2 | Received Data | (BB) | $S_{in}$ | IN | 1 2 |
| M | PA2 | Transmitted Data | (BA) | $S_{out}$ | OUT | 1 2 |
| A | GND | Protective Ground | (AA) | GND | | 1 2 |
| N | GND | Signal Ground | (AB) | GND | | 1 2 3 |

MODES:
1. 3-Line Interface ($S_{in}$,$S_{out}$,GND).
2. X-Line Interface.
3. User Available Only (Unused/unimplemented in code).

* These lines are held high during 3-Line mode.

**Table 12–1. User Port Lines**

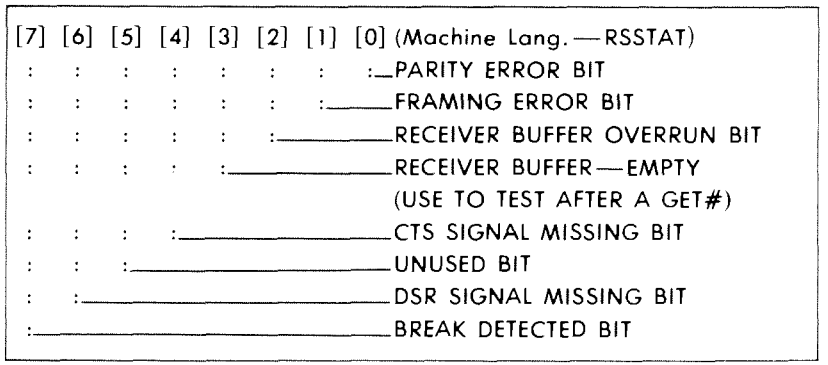



**Figure 12–3. RS-232 Status Register**

**NOTES to Figure 12–3:**
The RS-232 Status Register can be read with these statements.
If the STATUS = 0, then no error has been detected.

S = PEEK (2580): PRINT S : POKE 2580,0

If the RS-232 status is read by BASIC with the above statements or by using the KERNAL READST routine, the RS-232 status word is cleared when you exit. If multiple uses of the STATUS word are necessary, the S variable above should be assigned to another variable. For example:

SR = S: REM ASSIGNS S TO SR

The RS-232 status is read (and cleared) only when the RS-232 channel was the last external I/O used.

## SAMPLE BASIC PROGRAM USING RS-232 CHANNEL

This simple terminal program works in both C128 and C64 modes:

```
10 REM SIMPLE DUMB TERMINAL EMULATOR
20 REM THIS PROGRAM REQUIRES A MODEM
30 OPEN5,2,3,CHR$(6)+CHR$(0)
40 DIMF%(255),T%(255)
50 FORJ=32TO64:T%(J)=J:NEXT
60 T%(13)=13:T%(20)=8:RV=18:CT=0
70 FORJ=65TO90:K=J+32:T%(J)=K:NEXT
80 FORJ=91TO95:T%(J)=J:NEXT
90 FORJ=193TO218:K=J-128:T%(J)=K:NEXT
100 T%(146)=16:T%(133)=16
110 T%(137)=3:T%(134)=17:T%(138)=19
120 FORJ=0TO255
130 K=T%(J)
140 IFK<>0THENF%(K)=J:F%(K+128)=J
150 NEXT
160 PRINT" "CHR$(147)
170 GET#5,A$
180 IFA$=""ORST<>0THEN220
190 PRINT" "CHR$(157);CHR$(F%(ASC(A$)));
200 IFF%(ASC(A$))=34THENPOKE212,0
210 GOTO170
220 PRINTCHR$(RV)" "CHR$(157);CHR$(146);:GETA$
230 IFA$<>""THENPRINT#5,CHR$(T%(ASC(A$)));
240 CT=CT+1
250 IFCT=8THENCT=0:RV=164-RV
260 GOTO170
```

## ZERO-PAGE MEMORY LOCATIONS AND USAGE FOR RS-232 SYSTEM INTERFACE

The zero-page RS-232 storage locations for both C128 and C64 modes are as follows:

**$00A7–INBIT**—Receiver input bit temp storage.
**$00A8–BITCI**—Receiver bit count in.
**$00A9–RINONE**—Receiver flag start bit check.

**$00AA–RIDATA**—Receiver byte buffer/assembly location.
**$00AB–RIPRTY**—Receiver parity bit storage.
**$00B4–BITTS**—Transmitter bit count out.
**$00B5–NXTBIT**—Transmitter next bit to be sent.
**$00B6–RODATA**—Transmitter byte buffer/disassembly location.

All the above zero-page locations are used locally and are given only as a guide to understand the associated routines. These cannot be used directly by the BASIC or Kernal-level programmer to do RS-232–type programming functions. The system RS-232 routines must be used.

## NONZERO-PAGE MEMORY LOCATIONS AND USAGE FOR RS-232 SYSTEM INTERFACE

The general (nonzero-page) RS-232 storage locations for C128 mode are as follows: (The addresses in parentheses are for C64 mode.)

**$0A10($0293)–M51CTR**—Pseudo 6551 Control Register (see Figure 12–1).
**$0A11($0294)–M51CDR**—Pseudo 6551 Command Register (see Figure 12–2).
**$0A12($0295)–M51AJB**—Two bytes following the Control and Command Registers in the file name field. These locations contain the user-defined baud rate for the start of the bit test during the interface activity, which, in turn, is used to calculate baud rate.
**$0A14($0297)—RSSTAT**—The RS-232 Status Register (see Figure 12–3).
**$0A15($0298)–BITNUM**—The number of bits to be sent/received.
**$0A16($0299)–BAUDOF**—Two bytes that are equal to the time of one bit cell (based on system clock/baud rate).
**$0A18($029B)–RIDBE**—The byte index to the end of the receiver FIFO buffer.
**$0A19($029C)–RIDBS**—The byte index to the start of the receiver FIFO buffer.
**$0A1A($029D)–RODBS**—The byte index to the start of the transmitter FIFO buffer.
**$0A1B($029E)–RODBE**—The byte index to the end of the transmitter FIFO buffer.
**$0A0F($02A1)–ENABL**—Holds current active interrupts in the CIA #2 ICR. When bit 4 is turned on, the system is waiting for the Receiver Edge. When bit 1 is turned on, then the system is receiving data. When bit 0 is turned on, then the system is transmitting data.

## OUTPUT TO A SCREEN

The most common form of computer output (called *soft copy*) is a display tube that displays luminescent dots on a screen controlled by electrical signals. Although the television is a specialized monitor and limited by the dot density prescribed by FCC standards for proper reception, it is available in most homes and becomes a ready-to-use output device for the home computer. The C128 computer sends radio frequency signals through the TV port to the television's antenna and tuner.

Monitors for computers produce better-quality displays since they are direct-wired

and involve no tuners. The C128 includes a port to accept the traditional monitors as well as Commodore's 40-column 1702 color monitor. The C128 80-column color capability is enhanced with the new 40/80-column color monitor Model 1902. In this RGBI monitor, signals separately control each of the three color guns (red, green, blue) for maximum clarity and detail.

The screen can be accessed as an input or output device through the OPEN command as device 3.

```
OPEN 1,3
```

The screen is the default output device on the C128. Typing on the keyboard (device 0) results in a screen character display output whenever the monitor is powered up. No special commands are normally required. However, once control is sent to another device, reversion to the screen may be desired.

Control of the screen display is available primarily with the PRINT command; associated punctuation including the quote and the semicolon and common, BASIC functions including the TAB(X), SPC(X), and CHR$(X) as well as the POKE command. In addition, there are color controls, sprite animation and graphics keys.

If using the dual 1902 monitor or two separate 40- and 80-column monitors, it is possible to swap control between the two screens via the **40/80** switch or pressing the **ESC** and X keys. If one is used for text and the other for graphics, use the GRAPHIC command.

The following command will also swap the display output device:

```
PRINT CHR$(27)"X"
```

In addition to a screen display, sound commands can be used through a Commodore monitor or the television speakers. See Chapter 11, Sound and Music.

# OUTPUT TO THE DATASSETTE

Commodore programs and files can also be stored on tape cassette using a special recorder called the *Datassette*. Unlike most tape recorders, which store analog sinusoidal signals, the Datassette stores digital pulse signals that are more distinct, resulting in a reliable storage format. Compared to a disk storage system, tape is somewhat slower and limited to program and sequential files. Access to the files on tape is sequential.

Just as **DLOAD** automatically loads a program from disk, **LOAD** loads a program from tape. Similarly **SAVE** and **VERIFY** automatically default to tape. A LOAD for machine code requiring a secondary address would be:

```
LOAD"NAME",1,1
```

Access to tape for sequential files is made with an OPEN statement to device 1:

```
OPEN1,1
```

A secondary address of 0 (default value) assumes the file is opened for reading (using INPUT# or GET#). A secondary address for writing (PRINT# statements) requires a 1 or 2, where 2 includes an end-of-tape marker.

OPEN1,1,2"FILENAME"

The end-of-tape marker prevents searching for files beyond the data file you are looking for. It is written when the CLOSE command is given.

> **NOTE:** Do not connect the Datassette and the 1571 disk drive to the C128 at the same time. If they are used at the same time, the data transfers may not be reliable.

# INPUT FROM THE CONTROLLER PORTS

The C128 has two controller ports (numbered 1 and 2) that allow the use of control devices such as a mouse, joystick, paddle or light pen. Each game port will accept either one joystick, one mouse, or one paddle pair. A light pen can be plugged only into Port 1.

## JOYSTICKS

A digital joystick has five distinct switches. Four of the switches are used for direction and one is used for the fire button. The joystick switches are arranged as shown in Figure 12–4.

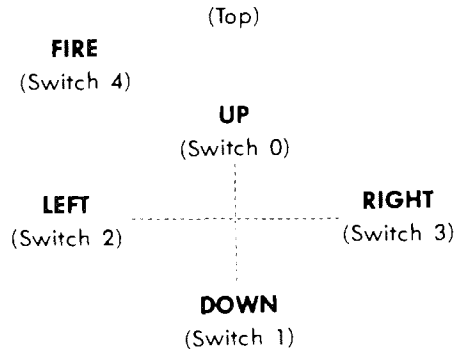



**Figure 12–4. Joystick Switch Arrangement**

These switches correspond to the lower 5 bits of the data in location 56320 or 56321. Normally the bit is set to a 1 if a direction is *not* chosen or the fire button is *not* pressed. When the fire button is pressed, the bit (bit 4 in this case) changes to a 0. To read the joystick from C64 BASIC, the following subroutine should be used:

```
10 REM JOYSTICK READ ROUTINE FOR C128 OR C64 MODES
20 JV=PEEK(56321)
30 IF (JV AND 15)<15 THEN PRINT 15-(JV AND 15)
40 FIRE=JV AND 16
50 IF FIRE<>16 THEN PRINT"FIRE"
60 GOTO 20
```

**NOTE:** For the first joystick, set JV = PEEK (56320).

The values for JV correspond to the joystick directions shown in Table 12–2.

| JV EQUAL TO | DIRECTION |
|---|---|
| 0 | NONE |
| 1 | UP |
| 2 | DOWN |
| 3 | — |
| 4 | LEFT |
| 5 | UP & LEFT |
| 6 | DOWN & LEFT |
| 7 | — |
| 8 | RIGHT |
| 9 | UP & RIGHT |
| 10 | DOWN & RIGHT |

**Table 12–2. Joystick Direction Values**

A machine language routine that accomplishes the same task is given in the section on shift and rotate instructions in Chapter 5, Machine Language on the Commodore 128.

BASIC 7.0 includes special joystick functions. To determine the position of joystick N, **PRINT JOY(N)** returns the position of the joystick. A number between 0 and 8 inclusive specifies the relative position where 0 is stationary and 1 through 8 represent clockwise positions. Values between 128 and 136 signify the fire button is pressed as well. Thus if **JOY(2)** returns 131, it means joystick number 2 fires to the right (position 3).

```
10 REM C128-MODE SPRITE-MOVE JOYSTICK ROUTINE
20 SPRITE 1,1,7:REM ENABLE SPRITE 1
30 MOVSPR 1,160,150:REM PLACE SPRITE 1 ON CENTER OF THE SCREEN
40 DO
50 :  DO
60 :     A=JOY(1):REM GET JOYSTICK DIRECTION VALUE
70 :     IF A=0 THEN LOOP:REM LOOP IF ZERO - THIS KEEPS SPRITE STATIONARY
80 :     B=A:REM SET B EQUAL TO A
90 :     Z=(A-1)*45:REM CALCULATE ANGLE
100 :    MOVSPR 1,5;Z:REM MOVE SPRITE 5 UNITS AT THE CALCULATED ANGLE
110 :    A=JOY(1):REM CHECK JOYSTICK VALUE AGAIN
120 : LOOP WHILE A=B:REM MOVE IN THE SAME DIRECTION WHILE NEW VAL=OLD JOY VALUE
130 LOOP:REM OTHERWISE CHECK JOY VALUE AGAIN AND MOVE IN DIFFERENT DIRECTION
```

Using Boolean logic and the masking technique, a comparison can be made to determine whether the fire button was depressed:

IF (JOY(N)AND 128)=128 THEN PRINT"FIRE"

## THE 1350 MOUSE

The Commodore 128 makes use of an additional new peripheral device known as a mouse. The Commodore mouse is a user interface that allows you to control the operations of a program, such as selecting menu or icon options, without the need to type on the keyboard. It attaches to the controller ports like a joystick, paddle or light pen. Programs that support a mouse usually have a pointer indicator to let you know where the mouse is pointing on the screen. To move the pointer on the screen, place the mouse on a flat surface and move it in the direction you want to move the pointer. Lift the mouse off the surface and the screen indicator remains stationary. The left hand button responds like a joystick fire button. The right hand button may have additional functions as specified by a particular software package.

As far as BASIC is concerned, the mouse is supported by the JOY function, the same way a joystick is controlled. The previous joystick program example, which moves a sprite on the screen, also works with the mouse. The Joystick Musician program in Chapter 11 also works with the mouse. Any BASIC program using the JOY joystick function should operate with the Commodore mouse.

The mouse also works in C64 mode, the same way a joystick routine would in C64 mode. The previous C64 mode joystick example program supports the mouse in C64 mode.

## PADDLES

A paddle is connected to both CIA-1 and the SID chip (MOS 6581 Sound Interface Device) through a controller port. The paddle value is read via the SID registers 54297 ($D419) and 54298 ($D41A). *Paddles are not reliable when read from BASIC alone!* The best way to use paddles, from BASIC or machine code, is to use a machine language routine (SYS to it from BASIC then PEEK the memory locations used by the subroutine).

The following BASIC program moves a sprite with paddles:

```
10 REM C128-MODE SPRITE-MOVE PADDLE ROUTINE
20 REM PLACE DUAL PADDLES IN CONTROLLER PORT A
30 REM PADDLE 1 CONTROLS HORIZONTAL SPRITE MOVEMENT
40 REM PADDLE 2 CONTROLS VERTICAL MOVEMENT
50 SPRITE 1,1,7
60 DO
70 A=POT(1):REM GET POSITION (X) OF PADDLE 1
80 B=POT(2):REM GET POSITION (Y) OF PADDLE 2
90 PRINT A;B:REM PRINT VALUES
100 MOVSPR 1,A,B:REM MOVE SPRITE 1 ACCORDING TO PADDLE VALUES
110 IF (A>255) OR (B>255) THEN BEGIN:REM CHECK FOR FIRE BUTTON
120 :       SOUND 2,64000,60,0,1000, 16000,3:REM FIRE
130 :       COLOR 0,3:REM CHANGE SCREEN COLOR
140 :       SLEEP 1  :REM DELAY
150 :       COLOR 0,1:REM CHANGE SCREEN COLOR BACK
160 BEND:REM END THE BEGIN/BEND STRUCTURE
170 LOOP:REM DO IT AGAIN
```

## LIGHT PEN

The light pen input latches the current screen position into a pair of registers (LPX, LPY) on a low-going edge. The X position register 53267 ($D013) will contain the 8 MSB of the X position at the time of transition. Since the X position is defined by a 512-state counter (9 bits), resolution to two horizontal dots is provided. Similarly, the Y position is latched in its register 53268 ($D014), but here 8 bits provide single raster resolution within the visible display. The light pen latch may be triggered only once per frame, and subsequent triggers within the same frame will have no effect. Therefore, you must take several samples before turning the pen to the screen (three or more samples average), depending upon the characteristics of your light pen. The following program is a light pen draw routine in C128 mode:

```
10 REM C128 MODE LIGHT PEN DRAW ROUTINE
20 REM PLACE LIGHT PEN IN CONTROLLER PORT 1
30 COLOR 0,2:COLOR 4,2:COLOR 1,1:REM SELECT SCREEN COLORS
40 GRAPHIC 1,1:REM SELECT BIT MAP MODE
50 DO
60 :    X=PEN(0)    :REM GET HORIZONTAL COORDINATE
70 :    Y=PEN(1)    :REM GET VERTICAL COORDINATE
80 :    DRAW 1,X,Y  :REM PLOT POINT
90 LOOP :REM DO IT AGAIN
100 :
110 REM   IF YOU LIKE THE DRAWING,
120 REM   BREAK THE PROGRAM AND
130 REM   SAVE IT AS A BINARY FILE
140 REM   FROM $2000-$3FFF.
```

## OUTPUT CONTROL TO ALL DEVICES

Each peripheral device is identified by a device number, as listed in Table 12–3.

| DEVICE | DEVICE NUMBER | SECONDARY ADDRESSES |
|---|---|---|
| **Keyboard** | **0** | |
| **Datasette** | **1** | **0,1,2** |
| **Modem (RS232)** | **2** | **0** |
| **Screen** | **3** | **0,1** |
| **Printer** | **4 or 5** | **0,7,etc** |
| **Disk Drive** | **8,9,10,11** | **2–14,15** |

**Table 12–3. Device Numbers**

Serial devices can be numbered 4 through 30, but they usually conform to the above conventions. All devices can be accessed in a single OPEN statement by using an integer variable:

```
10 INPUT "SELECT DEVICE NUMBER";D%
20 INPUT "SELECT SECONDARY ADDRESS";S%
30 INPUT "NAME OF FILE OR CONTROL REGISTER";A$
40 OPEN 9,D%,S%,A$
50 GOSUB 80
60 CLOSE 9
70 END
80 REM PLACE SUBROUTINE HERE
90 RETURN
```

# THE INPUT/OUTPUT PINOUTS

## THE SERIAL PORT

The Serial Port is a daisy chain arrangement designed to let the Commodore 128 computer communicate with more than one device, such as Commodore disk drives and printers. Up to five devices can be connected at one time. The serial bus can transmit control signals from the computer, send data onto the bus, and receive data from the bus. Data or control signals are routed to the proper device by opening a specific bus address ranging from 4 to 31. Typically, addresses 4 and 5 are reserved for printers, and addresses 8 through 11 are for the disk drive.

Six lines are used in serial bus operation—three input and three output. The three input lines bring data, control and timing signals into the computer. The three output lines send data, control and timing signals from the computer to the external devices. See Hardware Specifications (Chapter 16) for details on the C128 serial bus protocol.

**SERIAL I/O**

| PIN | TYPE |
|---|---|
| 1 | SERIAL SRQIN |
| 2 | GND |
| 3 | SERIAL ATN IN/OUT |
| 4 | SERIAL CLK IN/OUT |
| 5 | SERIAL DATA IN/OUT |
| 6 | RESET |

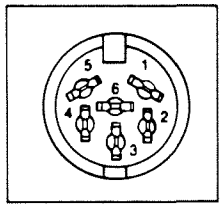



**Figure 12–5. Serial Port Pinouts**

## USER PORT (RS-232 CHANNEL)

The User Port (RS-232 Channel) connects the Commodore 128 to the outside world, such as another Commodore computer. It is directly connected to the 6526 CIA #1 chip. Figure 12–6 shows the User Port pinouts. Table 12–4 defines the pinouts.

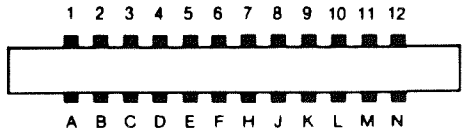



**Figure 12–6. User Port Pinouts**

| PIN | | |
|---|---|---|
| TOP SIDE | DESCRIPTION | NOTES |
| 1 | GROUND | |
| 2 | +5V | (100 mA MAX.) |
| 3 | RESET | By grounding this pin, the Commodore 128 will do a cold start, resetting completely. The pointers to a BASIC program will be reset, but memory will not be cleared. This is also a RESET output for the external devices. |
| 4 | CNT1 | Serial port counter from CIA-1 (see CIA specs). |
| 5 | SP1 | Serial port from CIA-1 (see 6526 CIA specs). |
| 6 | CNT2 | Serial port counter from CIA-2 (see CIA specs). |
| 7 | SP2 | Serial port from CIA-1 (see 6526 CIA specs). |
| 8 | PC2 | Handshaking line from CIA-2 (see CIA specs). |
| 9 | SERIAL ATN IN | This pin is connected to the ATN line of the serial bus. |
| 10 | 9 VAC + phase | Connected directly to the Commodore 128 transformer (100 mA Max.). |
| 11 | 9 VAC–phase | |
| 12 | GROUND | |
| BOTTOM SIDE | | |
| A | GROUND | The Commodore 128 gives you control over Port B on CIA chip 1. Eight lines for input or output are available, as well as two lines for handshaking with an outside device. The I/0 lines for Port B are controlled by two locations. One is the Port itself, and is located at 56577 ($DD01 HEX). Naturally you PEEK it to read an INPUT, or POKE it to set an OUTPUT. Each of the eight I/O lines can be set up as either an INPUT or an OUTPUT by setting the Data Direction Register properly. |
| B | FLAG2 | |
| C | PB0 | |
| D | PB1 | |
| E | PB2 | |
| F | PB3 | |
| H | PB4 | |
| J | PB5 | |
| K | PB6 | |
| L | PB7 | |
| M | PA2 | |
| N | GROUND | |

**Table 12–4. User Port Pin Descriptions**

The **Data Direction Register** has its location at 56579 ($DD03 hex). Each of the eight lines in the port has a bit in the 8-bit Data Direction Register (DDR) that controls whether that line will be an input or an output. If a bit in the DDR is a 1, the

corresponding line of the port will be an output. If a bit in the DDR is a 0, the corresponding line of the port will be an input. For example, if bit 3 of the DDR is set to 1, then line 3 of the port will be an output. If the DDR is set like this:

```
BIT #:  7 6 5 4 3 2 1 0
VALUE: 0 0 1 1 1 0 0 0
```

lines 5, 4 and 3 will be outputs since those bits are 1's. The rest of the lines will be inputs, since those lines are 0's.

To PEEK or POKE the User Port, it is necessary to use both the DDR and the port itself.

Remember that the PEEK and POKE statements need a number from 0 to 255. The numbers given in the example must be translated into decimal before they can be used. The value would be:

$$2^5 + 2^4 + 2^3 = 32 + 16 + 8 = 56$$

Notice that the bit number for the DDR is the same number that is equal to 2 raised to a power to turn the bit value on.

(16 = 2↑4=2×2×2×2, 8 = 2↑3=2×2×2)

The two other lines, flag1 and PA2, are different from the rest of the User Port. These two lines are mainly for handshaking, and are programmed differently from port B.

Handshaking is needed when two devices communicate. Since one device may run at a different speed than another device, it is necessary to give each device some way of knowing what the other device is doing. Even when the devices are operating at the same speed, handshaking is necessary to communicate when data is to be sent, and if it has been received. The flag1 line has special characteristics that make it well suited for handshaking.

Flag1 is a negative-edge–sensitive input that can be used as a general-purpose interrupt input. Any negative transition on the flag line will set the flag interrupt bit. If the flag interrupt is enabled, this will cause an **Interrupt Request.** If the flag bit is not enabled, it can be polled from the Interrupt Register under program control.

PA2 is bit 2 of port A of the CIA. It is controlled like any other bit in the port. The port is located at 56576 ($DD00). The Data Direction Register is located at 56578 ($DD02).

For more information on the 6526, see the hardware chapter.

## THE COMPOSITE VIDEO CONNECTOR

This DIN connector supplies direct audio and composite video signals. These can be connected to the Commodore monitor or used with separate components. This is the 40-column output connector. Figure 12–7 shows the pinouts for the composite video connector. Table 12–5 describes the pinouts.

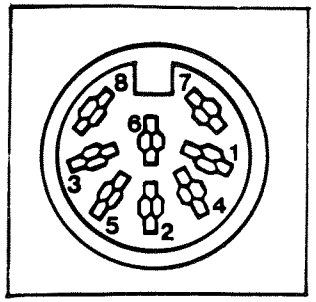



**Figure 12–7. Composite Video Connector Pinouts**

| PIN | TYPE | NOTE |
|---|---|---|
| 1 | LUM/SYNC | Luminance/SYNC output |
| 2 | GND | |
| 3 | AUDIO OUT | |
| 4 | VIDEO OUT | Composite signal output |
| 5 | AUDIO IN | |
| 6 | COLOR OUT | Chroma signal output |
| 7 | NC | No connection |
| 8 | NC | No connection |

**Table 12–5. Composite Video Connector Pin Descriptions**

# THE RGBI VIDEO CONNECTOR

The RGBI video connector is a 9-pin connector that supplies an RGBI (Red/Green/Blue/Intensity) signal. This is the 80-column output. Figure 12–8 shows the RGBI pinouts. Table 12–6 defines the RGBI pinouts.

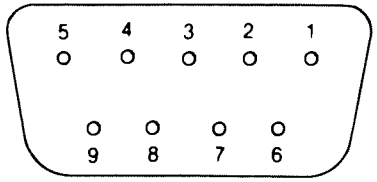



**Figure 12–8. RGBI Connector Pinouts**

| PIN | SIGNAL |
|---|---|
| 1 | Ground |
| 2 | Ground |
| 3 | Red |
| 4 | Green |
| 5 | Blue |
| 6 | Intensity |
| 7 | Monochrome |
| 8 | Horizontal Sync |
| 9 | Vertical Sync |

**Table 12–6. RGBI Connector Pin Descriptions**

## THE CASSETTE CONNECTOR

A 1530 Datassette recorder can be attached to the cassette port to store programs and information. Figure 12–9 shows the cassette port pinouts. Table 12–7 describes the pinouts.

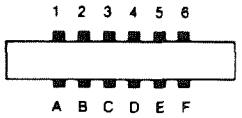



**Figure 12–9. Cassette Port Pinouts**

| PIN | TYPE |
|---|---|
| A-1 | GND |
| B-2 | +5V |
| C-3 | CASSETTE MOTOR |
| D-4 | CASSETTE READ |
| E-5 | CASSETTE WRITE |
| F-6 | CASSETTE SENSE |

**Table 12–7. Cassette Port Pin Descriptions**

## THE CONTROLLER PORTS

There are two controller ports, numbered 1 and 2. Each controller port can accept a joystick, mouse or game controller paddle. A light pen can be plugged only into Port 1, the port closest to the front of the computer. Use the ports as instructed with the software.

Figure 12–10 shows the Controller Port pinouts. Table 12–8 describes the pinouts.

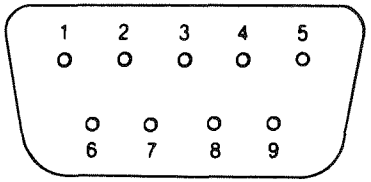



**Figure 12–10. Controller Port Pinouts**

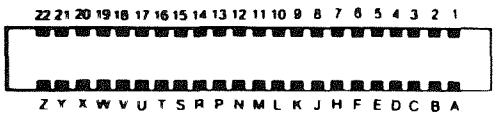
| CONTROL PORT 1 | | | CONTROL PORT 2 | | |
|---|---|---|---|---|---|
| PIN | TYPE | NOTE | PIN | TYPE | NOTE |
| 1 | JOYA0 | | 1 | JOYB0 | |
| 2 | JOYA1 | | 2 | JOYB1 | |
| 3 | JOYA2 | | 3 | JOYB2 | |
| 4 | JOYA3 | | 4 | JOYB3 | |
| 5 | POT AY | | 5 | POT BY | |
| 6 | BUTTON A/LP | | 6 | BUTTON B | |
| 7 | +5V | MAX. 50mA | 7 | +5V | MAX. 50mA |
| 8 | GND | | 8 | GND | |
| 9 | POT AX | | 9 | POT BX | |

**Table 12–8. Controller Port Pin Descriptions**

# THE EXPANSION PORT

The expansion port connector is a 44-pin female edge connector accessing the computer's address and data buses. It accepts Commodore's preprogrammed game and utility software cartridges. As a parallel port, it can accept Commodore's IEEE peripherals with an IEEE interface for controlling instrumentation and other devices. RAM expansion modules will also connect to this port and accept the BASIC commands of FETCH, STASH and SWAP.

Figure 12–11 shows the expansion port pinouts. Table 12–9 describes the pinouts.




**(view of port from the back of the C128)**

**Figure 12–11. Expansion Port Pinouts**

| PIN | TYPE | PIN | TYPE |
|---|---|---|---|
| 1 | GND | A | GND |
| 2 | +5V | B | ROMH |
| 3 | +5V | C | RESET |
| 4 | IRQ | D | NMI |
| 5 | R/W | E | S 02 |
| 6 | Dot Clock | F | A15 |
| 7 | I/O 1 | H | A14 |
| 8 | GAME | J | A13 |
| 9 | EXROM | K | A12 |
| 10 | I/O 2 | L | A11 |
| 11 | ROML | M | A10 |
| 12 | BA | N | A9 |
| 13 | DMA | P | A8 |
| 14 | D7 | R | A7 |
| 15 | D6 | S | A6 |
| 16 | D5 | T | A5 |
| 17 | D4 | U | A4 |
| 18 | D3 | V | A3 |
| 19 | D2 | W | A2 |
| 20 | D1 | X | A1 |
| 21 | D0 | Y | A0 |
| 22 | GND | Z | GND |

**Table 12–9. Expansion Port Pin Descriptions**

# 13

# THE COMMODORE 128 OPERATING SYSTEM

The Commodore 128 operating system controls, directly or indirectly, all functions of your computer. The Commodore 128 operating system is housed in a ROM chip called the **Kernal**, which contains about 16K of machine language instructions. These instructions make up the routines that control all the machine's functions—even the ones you take for granted. For instance, the Kernal controls all input and output functions, including receiving the characters from the keyboard when you type, sending text to a printer, and displaying graphics and text on the screen. Every task performed by the computer other than application program activities is controlled by the Kernal. The Kernal even manipulates and executes the application programs you load or type into your computer's memory.

## TAKING FULL ADVANTAGE OF THE COMMODORE 128 OPERATING SYSTEM

The 16K of operating system instructions contained in the Kernal are available for use with your own programs. Instead of "reinventing the wheel" and duplicating code, you can call (that is, use) these Kernal routines in your own programs. You do this through the Kernal Jump Table, which consists of a series of ROM entry points in which you can call machine language routines already available in the Commodore 128 Kernal. By calling these routines, which handle the most fundamental functions of the computer, you avoid duplicating code. This helps you utilize your computer to its fullest potential.

The Kernal Jump Table also facilitates compatibility. If the Commodore 128 operating system is modified or upgraded, which happens frequently in the microcomputer industry, the entry points in the jump table are revised to reflect address changes of Kernal subroutines. The key to keeping applications programs compatible from one version of the operating system to another is to enter the operating system through the Kernal Jump Table. Instead of jumping directly to a subroutine (JSR), use the Kernal Jump Table as the entry point, since it contains the correct address vector to the specified routine, regardless of the version of the Kernal being used. If you always enter the operating system routines from the Kernal Jump Table, the address of the desired routine will always be reached. On the other hand, if you bypass the Kernal Jump Table and jump directly to the address where you think the routine resides, you may cause an error, because the starting point of the desired routine may have changed from one version of the operating system to another.

# HOW TO USE (CALL) THE KERNAL ROUTINES IN YOUR OWN PROGRAMS

Most of the system subroutines require specific parameters, or values, which must be loaded into the accumulator, X or Y index registers. Some Kernal subroutines require additional preparatory routines to be called before invoking the specific target routine. Each Kernal subroutine terminates with an RTS instruction that tells the microprocessor to return from the subroutine. Many of the Kernal routines return values that are placed in the accumulator, X or Y registers. Some even return error codes in the status register, which can be acted on in your applications program.

Here is the procedure for calling a Kernal subroutine:

1. Place the necessary preparatory values in the required registers—either the A, X or Y registers.
2. Enable the appropriate system configuration. For example, if the routine requires the Kernal to be present, invoke either configuration 12, 13, 14 or 15, since these make the Kernal available. Bank 15 is the default configuration.
3. Call the subroutine with the JSR instruction, using the address of the jump vector as shown in the Kernal Jump Table in Figure 13-1.

   For example, the starting address of the routine ACPTR is stored, starting in location $FFA5. Actually, the operation code (opcode) for the JMP instruction is stored at location $FFA5 and the address of the entry point of the routine is stored at locations $FFA6 and $FFA7. What really happens is that the JSR instruction in your application program transfers control to the jump table address ($FFA5, for example); then the JMP instruction at $FFA5 jumps to the subroutine at the address specified in locations $FFA6 and $FFA7. In other words, when you issue a JSR instruction to the Kernal Jump Table, you actually perform two jumps: one (JSR) to the jump table, and then a second jump (JMP) to the actual starting address of the routine.

   The routine terminates with an RTS instruction that is already part of the Kernal subroutine. Your application program resumes with the instruction immediately following the JSR instruction.
4. Upon return from the subroutine, check the status register for any error conditions. If an error condition is present, take precautions in your application program to handle the error and act on the value of the status register.

   Figure 13–1 is the C128 system vector and jump table that includes the name, address and function description of the operating system Kernal routines.

## C128 SYSTEM VECTORS

| | | | |
|---|---|---|---|
| 1. | $FFFB | SYSTEM | ;operating system vector (RAM1) |
| 2. | $FFFA | NMI | ;processor NMI vector |
| 3. | $FFFC | RESET | ;processor RESET vector |
| 4. | $FFFE | IRQ | ;processor IRQ/BRK vector |

## CBM STANDARD KERNAL JUMP TABLE CALLS

| | | | |
|---|---|---|---|
| 1. | $FF81 | JMP CINT | ;init screen editor and devices |
| 2. | $FF84 | JMP IOINIT | ;init I/O devices |
| 3. | $FF87 | JMP RAMTAS | ;init RAM and buffers |
| 4. | $FF8A | JMP RESTOR | ;init indirect vectors (system) |
| 5. | $FF8D | JMP VECTOR | ;init indirect vectors (user) |
| 6. | $FF90 | JMP SETMSG | ;kernal messages on/off |
| 7. | $FF93 | JMP SECND | ;serial: send SA after LISTN |
| 8. | $FF96 | JMP TKSA | ;serial: send SA after TALK |
| 9. | $FF99 | JMP MEMTOP | ;set/read top of system RAM |
| 10. | $FF9C | JMP MEMBOT | ;set/read bottom of system RAM |
| 11. | $FF9F | JMP KEY | ;scan keyboard |
| 12. | $FFA2 | JMP SETTMO | ;(reserved) |
| 13. | $FFA5 | JMP ACPTR | ;serial: byte input |
| 14. | $FFA8 | JMP CIOUT | ;serial: byte output |
| 15. | $FFAB | JMP UNTLK | ;serial: send untalk |
| 16. | $FFAE | JMP UNLSN | ;serial: send unlisten |
| 17. | $FFB1 | JMP LISTN | ;serial: send listen |
| 18. | $FFB4 | JMP TALK | ;serial: send talk |
| 19. | $FFB7 | JMP READSS | ;read I/O status byte |
| 20. | $FFBA | JMP SETLFS | ;set channel LA, FA, SA |
| 21. | $FFBD | JMP SETNAM | ;set filename pointers |
| 22. | $FFC0 | JMP (IOPEN) | ;open logical file |
| 23. | $FFC3 | JMP (ICLOSE) | ;close logical file |
| 24. | $FFC6 | JMP (ICHKIN) | ;set input channel |
| 25. | $FFC9 | JMP (ICKOUT) | ;set output channel |
| 26. | $FFCC | JMP (ICLRCH) | ;restore default channels |
| 27. | $FFCF | JMP (IBASIN) | ;input from channel |
| 28. | $FFD2 | JMP (IBSOUT) | ;output to channel |
| 29. | $FFD5 | JMP LOAD | ;load from file |
| 30. | $FFD8 | JMP SAVE | ;save to file |
| 31. | $FFDB | JMP SETTIM | ;set internal clock |
| 32. | $FFDE | JMP RDTIM | ;read internal clock |
| 33. | $FFE1 | JMP (ISTOP) | ;scan STOP key |

```
34.  $FFE4   JMP (IGETIN)     ;read buffered data
35.  $FFE7   JMP (ICLALL)     ;close all files and channels
36.  $FFEA   JMP UDTIM        ;increment internal clock

37.  $FFED   JMP SCRORG       ;get current screen window size
38.  $FFF0   JMP PLOT         ;read/set cursor position
39.  $FFF3   JMP IOBASE       ;read base address of I/O block
```

**NEW C128 KERNAL JUMP TABLE CALLS**

```
1.  $FF47   JMP SPIN SPOUT    ;setup fast serial ports for I/O

2.  $FF4A   JMP CLOSE ALL     ;close all files on a device
3.  $FF4D   JMP C64MODE       ;reconfigure system as a C64
4.  $FF50   JMP DMA CALL      ;send command to DMA device

5.  $FF53   JMP BOOT CALL     ;boot load program from disk
6.  $FF56   JMP PHOENIX       ;init function cartridges
7.  $FF59   JMP LKUPLA        ;search tables for given LA

8.  $FF5C   JMP LKUPSA        ;search tables for given SA
9.  $FF5F   JMP SWAPPER       ;switch between 40 and 80 columns
10. $FF62   JMP DLCHR         ;init 80-col character RAM

11. $FF65   JMP PFKEY         ;program a function key
12. $FF68   JMP SETBNK        ;set bank for I/O operations
13. $FF6B   JMP GETCFG        ;lookup MMU data for given bank

14. $FF6E   JMP JSRFAR        ;gosub in another bank
15. $FF71   JMP JMPFAR        ;goto another bank
16. $FF74   JMP INDFET        ;LDA (fetvec),Y from any bank

17. $FF77   JMP INDSTA        ;STA (stavec),Y to any bank
18. $FF7A   JMP INDCMP        ;CMP (cmpvec),Y to any bank
19. $FF7D   JMP PRIMM         ;print immediate utility
```

**Figure 13–1. User-Callable Kernal Routines**

Figure 13–2 lists the conventions used in the description of each Kernal subroutine. The figure is followed by descriptions of the C128 system vectors and Kernal subroutines. Included in each description are the subroutine name, call address, preparatory routines needed (if any), the registers affected, the error codes associated with each routine, how to use them and an example of each kernal subroutine call.

# USER CALLABLE KERNAL ROUTINE CONVENTIONS

**Call Address:** This is the call address of the Kernal routine, given in hexadecimal.

**Function Name:** Name of the Kernal routine.

**Register:** Registers, memory and flags listed under this heading are used to pass parameters to and from the Kernal routines.

**Preparatory Routines:** Certain Kernal routines require that data be set up by preparatory routines before the target routine can be called. The necessary routines are listed here.

**Error Returns:** A return from a Kernal routine with the carry set indicates an error was encountered in processing. The accumulator will contain the number of the error.

**Error Codes:** Below is a list of error messages that can occur when using the Kernal routines. If an error occurs during a Kernal routine, the carry bit of the accumulator is set, and the number of the error message is returned in the accumulator.

> **NOTE:** Some Kernal I/O routines do not use these codes for error messages. Instead, errors are identified using the Kernal READST routine.

| NUMBER | MEANING |
|---|---|
| 0 | **Routine terminated by the STOP key** |
| 1 | **Too many open files** |
| 2 | **File already open** |
| 3 | **File not open** |
| 4 | **File not found** |
| 5 | **Device not present** |
| 6 | **File is not an input file** |
| 7 | **File is not an output file** |
| 8 | **File name is missing** |
| 9 | **Illegal device number** |
| 41 | **File read error** |

**Registers Affected:** All registers, memory and flags used by the Kernal routine are listed here.

**Examples:** An example of each Kernal routine is listed.

**Description:** A short explanation of the function of the Kernal routine is given here.

**Figure 13–2. User-Callable Kernal Routine Conventions**

# C128 SYSTEM VECTORS

The vectors listed below, with the exception of the SYSTEM vector, are located not only in system ROM but in each RAM bank. The beginning and end of the system interrupt handlers are found in the top page ($FFxx) of all memory configurations as well. The reason is simple: An interrupt can occur anytime, from any memory configuration. The registers and memory configuration must be preserved prior to bringing the operating system into context to process the interrupt. They must similarly be restored before control is finally returned to the interrupted code. Note that the system vectors are indirect jumps, and are not usually called by the user since they terminate with an RTI instruction, not an RTS instruction. In other words, they process interrupted events, not subroutine calls.

1. $FFF8 SYSTEM ;operating system vector (RAM1)

   The **SYSTEM** vector and accompanying **KEY** string provide applications software with a means of regaining system control after a hardware reset. With this vector, software may be protected from an otherwise unrecoverable situation. The KEY string provides the distinction between a "warm" reset and a "cold" power-up. If the system has just powered up, the KEY string is missing and thus the SYSTEM vector is invalid; the **CBM** KEY is installed and the SYSTEM vector is set to C128 mode. If the system was reset (i.e., KEY was found), the SYSTEM vector is moved to common RAM at $02 and an indirect JMP is performed. In most cases, the user need only call IOINIT beore resuming control. The layout in RAM-1 is:

   | | | |
   |---|---|---|
   | $FFF5 | 'C' | ($43) |
   | $FFF6 | 'B' | ($42) |
   | $FFF7 | 'M' | ($4D) |
   | $FFF8 | SYSTEM | vector low |
   | $FFF9 | SYSTEM | vector high |

   For example, suppose a programmer set out to do some heavy-duty "hacking" using the built-in Monitor. Fully realizing the likelihood of losing control in a crash, the programmer could redirect the SYSTEM vector to the Monitor itself and thus regain control with minimal RAM loss, simply by pressing the RESET button. To accomplish this, the programmer would enter:

   ```
   a.  >1FFF8    00  13      :aim SYSTEM to $1300
   b.  A 1300  JSR   $FF84   :call IOINIT
               JMP   $B000   :call Monitor
   ```

2. $FFFA NMI ;processor NMI vector

   The **Non-Maskable Interrupt** (**NMI**) vector is activated whenever the processor NMI pin detects a negative edge. There are two possible sources of NMI's under normal conditions: the **RESTORE** key and RS-232 I/O. In the event of an NMI, the operating system disables IRQ's, saves the registers and current memory

configuration on the stack, brings the system configuration (ROM's, I/O, RAM0) into context, and executes an indirect jump through the RAM vector located at $318. The system NMI handler clears the ICR register of CIA-2, from which it determines the source of the interrupt. If it is from timer A, control passes to the RS-232 transceiver. If not, the **RESTORE** key is assumed, and for safety, the CBM convention of requiring the **STOP** key to be simultaneously depressed is checked. If the STOP key is depressed, all system indirect vectors are restored, IOINIT and CINT are called, and the SYSTEM_VECTOR (do not confuse with the SYSTEM vector!) is taken. Control is returned upon restoration of the registers and memory configuration. NMI's may be disabled by causing an initial NMI from timer A, but never reading the ICR to clear it, thus keeping the NMI signal grounded.

Application software can intercept an NMI event by modifying either of the two RAM vectors mentioned above. The NMI indirect vector at $318 in common RAM will pass control whenever an NMI occurs. The SYSTEM_VECTOR, located at $A00 in RAM0, will pass control after STOP/RESTORE is detected and handled.

For example, suppose a situation similar to the one illustrated previously for the SYSTEM vector occurs. To return control to the Monitor whenever STOP/RESTORE is detected, enter:

```
a.  >A00  00 B0  :aim SYSTEM_VECTOR to Monitor
```

Similarly, to perform an action anytime an NMI occurs (e.g., RESTORE alone), use the NMI indirect. For example, increment the VIC border color whenever you press **RESTORE** (or cause any other NMI). Enter:

```
a.  >318 00 13          :aim indirect to $1300
b.  A 1300 INC $D020    :change border color
           JMP $FF33    :return from interrupt
```

3. $FFFC RESET ;processor RESET vector

The processor **RESET** vector is activated whenever the system RESET signal is low. It is low at power-up, and is pulled low by pressing the RESET button. This signal effects not only the processor but most of the I/O devices found in the system. In fact, RESET is the one processor control signal that is shared between the two processors (8502 and Z80) of the C128. The Z80 gains initial (default) control of the system while the 8502 is held in a waiting state. When the 8502 finally starts after a reset, the Kernal initialization routine **START** always receives control and immediately performs the following actions:

1. Brings the system map into context.
2. Disables IRQ's.
3. Resets the processor stack pointer.
4. Clears decimal mode.
5. Initializes the MMU.
6. INSTALLS the Kernal RAM code.

Up to this point there is no provision for user code. The next two routines in the initialization path actually look for installed user code:

7. SECURE: Check and initialize the SYSTEM vector.
8. POLL: Check for a ROM cartridge.

**POLL** first scans for any installed C64 cartridges. They are recognized by either the **GAME** or **EXROM** signal being pulled low. If so, GO64 is executed (see the Kernal jump entry for details). Polling for C64 cartridges is actually redundant at this point since the Z80 processor, which powers up initially, did this already. POLL then scans for installed C128 cartridges and function ROM's. They are recognized by the existence of the C= key in any of the four function ROM slots (two internal, two external) and are polled in the order external low (16 or 32KB), external high (16KB), internal low (16 or 32KB), internal high (16KB). The entire format is:

$x000 → cold start entry
$x003 → warm start entry (unused)
$x006 → ID. ($01-$FF)
$x007 → "CBM" key string
where x = $8--- or $C---.

The ID of any C128 cartridge found is entered into the Physical Address Table (PAT) located at SAC1-SAC4. ID's must be non-zero. Cartridges may recognize each other by examining the PAT for particular ID's. An ID of 1 indicates an auto-start cartridge, and its cold start entry will be called immediately. All others will be called later (see PHOENIX jump), as will any auto-starters that RTS is to POLL. A cartridge can determine where it is installed by examining CURBNK, located at $ACO. Because it is possible for a cartridge to be called with interrupts enabled, the following diversion from the above format is recommended (the warm start entry is never called by the system):

```
$x000  SEI
$x001  JMP STARTUP
$x004  NOP
$x005  NOP
```

The balance of the C128 initialization is:

9. IOINIT: Initialize I/O devices.
10. Check for STOP AND C= keys.
11. RAMTAS: Initialize system RAM.
12. RESTOR: Initialize system indirects.
13. CINT: Initialize video displays.
14. Enable IRQ's (except foreign systems).
15. Dispatch.

**IOINIT** is perhaps the major function of the Reset handler. It initializes both CIA's (timers, keyboard, serial port, user port, cassette) and the 8502 port (keyboard,

cassette, VIC bank). It distinguishes a PAL system from an NTSC one and sets PALCNT ($A03) if PAL. The VIC, SID and 8563 devices are initialized, including the downloading of character definitions to 8563 display RAM (if necessary). The system 60Hz IRQ source (the VIC raster) is started. IOINIT is callable by the user via the jump table.

During initialization, the user may press certain keys as a means of selecting an operating mode. One key checked is the Commodore key C= , indicating C64 mode is desired. While this key was scanned much earlier by the Z80 to speed the switchover to C64 mode, there is a redundant check for it here. The only other key scanned at this time is the **STOP** key, which signals a request by the user to power up into the Monitor utility. Note that control does not pass from the initialization process at this point; the Kernal needs to know if **RAMTAS** should be skipped. Only if the **STOP** key is depressed *and* this was a "warm" reset (vs. "cold" power-up) can RAMTAS be skipped.

RAMTAS clears all page-zero RAM, allocates the cassette and RS-232 buffers, sets pointers to the top and bottom of system RAM (RAM-0), and installs the SYSTEM_VECTOR (discussed earlier under NMI) that points to BASIC cold start. Lastly it sets a flag, DEJAVU ($A02), to indicate to other routines that system RAM has been initialized. This is the difference between a "cold" and a "warm" system. If DEJAVU contains $A5, the system is "warm" and SYSTEM_VECTOR is valid. Many programmers debugging code need to recover from a system hang or crash via **RESET** but do not want RAM cleared. This is why the **STOP** key is scanned, RAMTAS is skipped, and the Monitor (rather than BASIC) is selected. RAMTAS is callable by the user via the jump table.

**RESTOR** initializes the Kernal indirect vectors. This must be done before many system routines will function. Applications that complement the operating system via "wedges" must install them after they are initialized. RESTOR is user callable from the jump table (see also the VECTOR call).

**CINT** is the Editor's initialization routine. Both 40- and 80-column display modes are prepared, editor indirect vectors installed, programmable key definitions assigned, and the 40/80 key scnaned for default display determination. CINT is also a jump table entry.

Finally, the IRQ's are enabled and control is passed to either BASIC initialization, GO64 code, or the ML Monitor. BASIC will call the Kernal PHOENIX routine upon the conclusion of its initialization, which will call any installed C128 cartridges (any ID) and attempt to auto-boot an application from disk.

An initialization status byte, INIT_STATUS ($A04), marks the progress of the initialization process. It is cleared automatically at the beginning of the Reset code, and as specific stages are completed, a particular bit is set. The layout is:

B7 → 8563 characters installed
B6 → CINT performed
B0 → BASIC initialized

Any IOINIT calls, including Reset, will *not* result in 8563 character RAM initialization if B7 is set. Similarly, CINT will *not* initialize the keyboard matrix lookup tables and the programmable key definitions if B6 is set. This is how NMI's, for example, can call IOINIT and CINT without destroying users' setups. Finally, BASIC initialization must be complete before B0 is set. This determines whether the IRQ handler, for example, should call the BASIC IRQ routines. Note that the following sequence of events should be performed for a BASIC programmer to recover from a crash via **RESET** :

1. Hold down **STOP** key to enter Monitor.
2. Press and release **RESET** button.
3. Release **STOP** key.
4. Enter: >A04 C1 :re-enable BASIC IRQ.
5. Enter: X :exit Monitor to BASIC.

This sequence is necessary because INIT_STATUS was cleared by the reset, and the BASIC initialization was skipped, leaving B0 reset. If B0 had not been set, BASIC IRQ routines such as SOUND, PLAY, and SPRITE handlers would not have functioned, usually resulting in an apparent "hang."

4. $FFFE IRQ ;processor IRQ/BRK vector

This hardware vector is taken whenever the **IRQ** pin of the processor is pulled low, or the processor executes a **BRK** instruction. For proper operation an IRQ must occur sixty times every second [NTSC (60Hz) presents no problems, but adjustments have to be made for PAL (50Hz) systems]. The usual source of IRQ's in the C128 is the VIC raster, which is unleashed during system initialization by IOINIT. In the event of an IRQ or BRK, the operating system saves the registers and current memory configuration on the stack and brings the system bank (ROM's, I/O, RAM0) into context. The processor status at the time of interruption is then read from the system stack to determine if the interrupt was an IRQ or a BRK. The C128 uses the following code to accomplish this:

```
TSX          ;get stack pointer
LDA $105,X   ;retrieve processor status
AND #$10     ;examine BRK flag
```

If the BRK flag is set, control passes to the ML Monitor through the BRK indirect vector at $316, which usually points to the Monitor BREAK entry. Here the program counter (PC), processor status, registers, memory configuration and stack pointer are retrieved and displayed.

If the BRK flag is 0, an IRQ is assumed and control passes through the IRQ indirect vector at $314, normally to the system IRQ handler. The following processes are then performed in the order shown:

1. IRQ's disabled.
2. Editor: split screen handler.
3. Editor: clear VIC raster IRQ.
4. IRQ's enabled.
5. Editor: keyboard scan.
6. Editor: VIC cursor blink.
7. Kernal: "jiffie" clock.
8. Kernal: cassette switches.
9. Kernal: clear CIA-1 ICR.
10. BASIC: sprites, sounds, etc.
11. Return from interrupt.

As indicated in the preceding description, the C128 operating system uses the IRQ heavily. In particular, the Editor split screen handler has rather strict requirements that programmers must recognize and accommodate. An all-text screen or a fully bit mapped screen presents no particular problem, but a split text and bit map screen requires twice the IRQ frequency (two every sixtieth of a second). Thus, the Editor (and consequently, IRQ-dependent applications) must distinguish between the "main" 60Hz IRQ and the "middle" IRQ. Only during the main IRQ will all the actions listed above be performed; the middle IRQ is only used by the Editor to split the screen. The Editor IRQ routine sets the carry flag to designate a main IRQ. Moreover, there is no margin in the timing requirements of a split screen. Programmers should note the way the Editor uses the VIC during IRQ's and avoid direct VIC I/O if the Editor screen operations are enabled. There is a flag byte called **GRAPHM**, located at $D8, which can be set to $FF to disable the Editor's use of the VIC.

The Editor is also responsible for scanning the C128 keyboard. Programmers should note that **SCNKEY** has two indirect jumps, **KEYLOG** and **KEYPUT**, it takes during its execution. The keyboard is controlled via CIA-1 PRA, PRB, VIC register #47 (extended key matrix), and the 8502 port (bit 6 = CAPS LOCK). The SCNKEY routine is callable from the jump table.

The balance of the IRQ routines up to calling BASIC are self-explanatory. The Kernal software clock is maintained by UDTIM, which is in the jump table. The IRQ processing makes one last call to BASIC_IRQ ($4006), but only if INIT_STATUS bit 0 is set indicating BASIC is ready to handle IRQs (this was discussed earlier in the RESET section). BASIC_IRQ is a heavy user of the VIC and SID, and the same precautions should be taken regarding direct VIC and SID I/O as with the Editor. BASIC_IRQ also utilizes a hold-off byte called IRQ_WRAP _FLAG. It is normally used by the system to block IRQ-ed IRQ calls, but can be set by the user with the effect of disabling the BASIC_IRQ handler. Alternatively you could clear bit 0 of the INIT_STATUS byte as mentioned above and achieve the same result.

The use of the IRQ indirect ($314) by an application usually requires more care than most other wedges for several reasons. The likelihood of an IRQ occurring while the wedge is being installed is greater, there exists the possibility that the user or some other software has already wedged the vector, and usually it is

desirable for the system IRQ functions to continue normally (e.g., keyscan) as opposed to replacing them totally with our own (as we did with the NMI examples). The following examples accomplish these objectives as well as masking out all but the main IRQ. First we must install our IRQ handler. This example converts the **40/80** key into a SLOW/FAST key:

```
A 1302 BIT  $D011      ;test VIC reg 17
       BPL  $1324      ;branch if wrong IRQ
       LDA  $D505      ;preparation
       ORA  #$80       ;b7 of MMU MCR
       STA  $D505      ;set for input
       LDA  $D011      ;assume FAST setup
       AND  #$6F       ;blank VIC (RC8=0)
       LDX  #$01       ;2MHz
       BIT  $D505      ;test 40/80 key
       BPL  $131E      ;branch if down (FAST)
       ORA  #$10       ;unblank VIC
       DEX             ;1MHz
 131E  STX  $D030      ;set speed
       STA  $D011      ;set blank bit
 1324  JMP  ($1300)    ;continue system IRQ
```

Now we need a small routine to actually wedge our code into the system IRQ. The following code saves the current IRQ vector for our handler above to exit through and substitutes a pointer to our code:

```
A 1400 SEI             ;prevent interruptions
       LDA  $314       ;get current IRQ lsb
       STA  $1300      ;and save it
       LDA  $315       ;get current IRQ msb
       STA  $1301      ;and save it
       LDA  #$02       ;get our IRQ lsb
       STA  $314       ;and substitute it
       LDA  #$13       ;get our IRQ msb
       STA  $315       ;and substitute it
       CLI             ;re-enable IRQ processing
       RTS
```

Enable the wedge by typing **J 1400**, that's all there is to it. Depressing the Locking **40/80** key now puts you in FAST mode; releasing it SLOWS you down, and the keyscan, etc., continues to function. Note, however, that on this

split screen our code throws off the timing, making for an unattractive display. There are really only three things to watch out for when toying with the C128 system IRQ: First, be sure to keep the raster compare value on-screen to keep the IRQ's happening (the best way is to keep RC8 zero as in example above); second, never attempt to access the 8563 during an IRQ if there is any chance that it is in use; finally, be sure the source of the IRQ is being cleared.

# CBM STANDARD KERNAL CALLS

The following system calls make up the set of standard CBM system calls for the Commodore 128 Personal Computer. Several of the calls, however, function somewhat differently or may require slightly different setups than C64 mode. This was necessary to accommodate specific features of the system, notably the 40/80 column windowing Editor and banked memory facilities. As with all Kernal calls, the system configuration (high ROM, RAM-0 and I/O) must be in context at the time of the call.

1. $FF81 CINT ;initialize screen editor and devices

PREPARATION:

| | |
|---|---|
| Registers: | none |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A used |
| | .X used |
| | .Y used |
| Memory: | init Editor RAM |
| | init Editor I/O |
| Flags: | none |

**EXAMPLE:**

```
SEI
JSR $FF81       ;initialize screen editor
CLI
```

**CINT** is the Editor's initialization routine. Both 40- and 80-column display modes are prepared, editor indirect vectors installed, programmable key definitions assigned, and the **40/80** key scanned for default display determination. CINT sets the VIC bank and VIC nybble bank, enables the character ROM, resets SID volume, places both 40- and 80-column screens and clears them. The only thing it

does not do that pertains to the Editor is I/O initialization, which is needed for IRQ's (keyscan, VIC cursor blink, split screen modes), key lines, screen background colors, etc. (see IOINIT). Because CINT updates Editor indirect vectors that are used during IRQ processing, you should disable IRQ's prior to calling it. CINT utilizes the status byte INIT_STATUS ($A04) as follows:

```
$A04 bit 6 = 0 → Full initialization.
                 (set INIT_STATUS bit 6)
           1 → Partial initialization.
                 (not key matrix pointers)
                 (not program key definitions)
```

CINT is also an Editor jump table entry ($C000).

2. $FF84 IOINIT ;init I/O devices

PREPARATION:

| | |
|---|---|
| Registers: | none |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A used |
| | .X used |
| | .Y used |
| Memory: | initialize I/O |
| Flags: | none |

**EXAMPLE:**

```
SEI
JSR $FF84       ;initialize system I/O
CLI
```

**IOINIT** is perhaps the major function of the Reset handler. It initializes both CIA's (timers, keyboard, serial port, user port, cassette) and the 8502 port (keyboard, cassette, VIC bank). It distinguishes a PAL system from an NTSC one and sets PALCNT ($A03) if PAL. The VIC, SID and 8563 devices are initialized, including the downloading of character definitions to 8563 display RAM (if necessary). The system 60Hz IRQ source, the VIC raster, is started (pending IRQs are cleared). IOINIT utilizes the status byte INIT_STATUS ($A04) as follows:

```
$A04 bit 7 = 0 → Full initialization.
                 (set INIT_STATUS bit 7)
           = 1 → Partial initialization.
                 (not 8563 character definitions)
```

You should be sure IRQ's are disabled before calling IOINIT to avoid interrupts while the various I/O devices are being initialized.

3. $FF87 RAMTAS ;init RAM and buffers

PREPARATION:

| | |
|---|---|
| Registers: | none |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A used |
| | .X used |
| | .Y used |
| Memory: | initializes RAM |
| Flags: | none |

**EXAMPLE:**

```
JSR $FF87       ;initialize system RAM
```

**RAMTAS** clears all page-zero RAM, allocates the cassette and RS-232 buffers, sets pointers to the top and bottom of system RAM (RAM 0) and points the SYSTEM_VECTOR ($A00) to BASIC cold start ($4000). Finally, it sets a flag, DEJAVU ($A02), to indicate to other routines that system RAM has been initialized and that the SYSTEM_VECTOR is valid. It should be noted that the C128 RAMTAS routine does *not* in any way test RAM.

4. $FF8A RESTOR ;init Kernal indirects

PREPARATION:

| | |
|---|---|
| Registers: | none |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A used |
| | .X used |
| | .Y used |
| Memory: | Kernal indirects restored |
| Flags: | none |

**EXAMPLE:**

```
SEI
JSR $FF8A       ;restore Kernal indirects
CLI
```

**RESTOR** restores the default values of all the Kernal indirect vectors from the Kernal ROM list. It does *not* affect any other vectors, such as those used by the Editor (see CINT) and BASIC. Because it is possible for an interrupt (IRQ or NMI) to occur during the updating of the interrupt indirect vectors, you should disable interrupts prior to calling RESTOR. See also the VECTOR call.

5. $FF8D VECTOR ;init or copy indirects

PREPARATION:

| | |
|---|---|
| Registers: | .X = adr (low) of user list |
| | .Y = adr (high) of user list |
| Memory: | system map |
| Flags: | .C = 0 → load Kernal vectors |
| | .C = 1 → copy Kernal vectors |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A used |
| | .Y used |
| Memory: | as per call |
| Flags: | none |

**EXAMPLE:**

```
LDX #save_lo
LDY #save_hi
SEC
JSR $FF87       ;copy indirects to "save"
```

**VECTOR** reads or writes the Kernal RAM indirect vectors. Calling VECTOR with the carry status set stores the current contents of the indirect vectors to the RAM address passed in the .X and .Y registers (to the current RAM bank). Calling VECTOR with the carry status clear updates the Kernal indirect vectors from the user list passed in the .X and .Y registers (from the current RAM bank). Interrupts (IRQ and NMI) should be disabled when updating the indirects. See also the RESTOR call.

6. $FF90 SETMSG ;Kernal messages on/off

PREPARATION:

| | |
|---|---|
| Registers: | .A = message control |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | none |
| Memory: | MSGFLG ($9D) updated |
| Flags: | none |

**EXAMPLE:**

```
LDA #0
JSR $FF90        ;turn OFF all Kernal messages
```

**SETMSG** updates the Kernal message flag byte MSGFLG ($9D) that determines whether system error and/or control messages will be displayed. BASIC normally disables error messages always and disables control messages in Run mode. Note that the Kernal error messages are not the verbose ones printed by BASIC, but simply the **I/O ERROR #** message that you see when in the Monitor, for example. Examples of Kernal control messages are **LOADING, FOUND,** and **PRESS PLAY ON TAPE.** The MSGFLG control bits are:

```
MSGFLG bit 7 = 1 → enable CONTROL messages
       bit 6 = 1 → enable ERROR messages
```

7. $FF93 SECND ;serial: send SA after LISTN

PREPARATION:

| | |
|---|---|
| Registers: | .A = SA (secondary address) |
| Memory: | system map |
| Flags: | none |
| Calls: | LISTN |

RESULTS:

| | |
|---|---|
| Registers: | .A used |
| Memory: | STATUS ($90) |
| Flags: | none |

**EXAMPLE:**

```
LDA #8
JSR $FF81        ;LISTN device 8
LDA #15
JSR $FF93        ;pass it SA #15
```

**SECND** is a low-level serial routine used to send a secondary address (SA) to a LISTNing device (see LISTN Kernal call). An SA is usually used to provide setup information to a device before the actual data I/O operation begins. Attention is released after a call to SECND. SECND is not used to send an SA to a TALKing device (see TKSA). (Most applications should use the higher level I/O routines: see OPEN and CKOUT).

8. $FF96 TKSA ;serial: send SA after TALK

PREPARATION:

| | |
|---|---|
| Registers: | .A = SA (secondary address) |
| Memory: | system map |
| Flags: | none |
| Calls: | TALK |

RESULTS:

| | |
|---|---|
| Registers: | .A used |
| Memory: | STATUS ($90) |
| Flags: | none |

**EXAMPLE:**

```
LDA #8
JSR $FFB4       ;TALK device 8
LDA #15
JSR $FF93       ;pass it SA #15
```

**TKSA** is a low-level serial routine used to send a secondary address (SA) to a device commanded to TALK (see TALK Kernal call). An SA is usually used to provide setup information to a device before the actual data I/O operation begins. (Most applications should use the higher-level I/O routines; see OPEN and CHKIN).

9. $FF99 MEMTOP ;set/read top of system RAM

PREPARATION:

| | |
|---|---|
| Registers: | .X = lsb of MEMSIZ |
| | .Y = msb of MEMSIZ |
| Memory: | system map |
| Flags: | .C = 0 → set top of memory |
| | .C = 1 → read top of memory |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .X = lsb of MEMSIZ |
| | .Y = msb of MEMSIZ |
| Memory: | MEMSIZ ($A07) |
| Flags: | none |

**EXAMPLE:**

```
SEC
JSR $FF99       ;get top of user RAM0
DEY
CLC
JSR $FF99       ;lower it 1 block
```

**MEMTOP** is used to read or set the top of system RAM, pointed to by MEMSIZ ($A07). This call is included in the C128 for completeness, but neither the Kernal nor BASIC utilizes MEMTOP since it has little meaning in the banked memory environment of the C128 (even the RS-232 buffers are permanently allocated). Nonetheless, set the carry status to load MEMSIZ into .X and .Y, and clear it to update the pointer from .X and .Y. Note that MEMSIZ references only system RAM (RAM0). The Kernal initially sets MEMSIZ to $FF00 (MMU and Kernal RAM code start here).

10. $FF9C MEMBOT ;set/read bottom of system RAM

PREPARATION:

| | |
|---|---|
| Registers: | .X = lsb of MEMSTR |
| | .Y = msb of MEMSTR |
| Memory: | system map |
| Flags: | .C = 0 → set bot of memory |
| | .C = 1 → read bot of memory |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .X = lsb of MEMSTR |
| | .Y = msb of MEMSTR |
| Memory: | MEMSTR ($A05) |
| Flags: | none |

**EXAMPLE:**

```
SEC
JSR $FF9C       ;get bottom of user RAM0
INY
CLC
JSR $FF9C       ;raise it 1 block
```

**MEMBOT** is used to read or set the bottom of system RAM, pointed to by MEMSTR ($A05). This call is included in the C128 for completeness, but neither the Kernal nor BASIC utilizes MEMBOT since it has little meaning in the banked memory environment of the C128. Nonetheless, set the carry status to load MEMSTR into .X and .Y, and clear it to update the pointer from .X and .Y. Note that MEMSTR references only system RAM (RAM0). The Kernal initially sets MEMSTR to $1000 (BASIC text starts here).

11. $FF9F KEY ;scan keyboard

PREPARATION:

| | |
|---|---|
| Registers: | none |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | none |
| Memory: | keyboard buffer |
| | keyboard flags |
| Flags: | none |

**EXAMPLE:**

```
JSR $FF9F       ;scan the keyboard
```

**KEY** is an Editor routine that scans the entire C128 keyboard (except the **40/80** key). It distinguishes between ASCII keys, control keys, and programmable keys, setting keyboard status bytes and managing the keyboard buffer. After decoding the key, KEY will manage such features as toggling cases, pauses or delays, and key repeats. It is normally called by the operating system during the 60Hz IRQ processing. Upon conclusion, KEY leaves the keyboard hardware driving the keyline on which the **STOP** key is located.

There are two indirect RAM jumps encountered during a keyscan: **KEYVEC** ($33A) and **KEYCHK** ($33C). KEYVEC (alias **KEYLOG**) is taken whenever a key depression is discovered, before the key in .A has been decoded. KEYCHK is taken after the key has been decoded, just before putting it into the key buffer. KEYCHK carries the ASCII character in .A, the keycode in .Y, and the shiftkey status in .X.

The keyboard decode matrices are addressed via indirect RAM vectors as well, located at DECODE ($33E). The following table describes them:

| | | |
|---|---|---|
| $33E | Mode 1 → | normal keys |
| $340 | Mode 2 → | **SHIFT** keys |
| $342 | Mode 3 → | **C=** keys |
| $344 | Mode 4 → | **CONTROL** keys |
| $346 | Mode 5 → | **CAPS LOCK** keys |
| $348 | Mode 6 → | **ALT** keys |

The following list briefly describes some of the more vital variables utilized or maintained by KEY:

| | | | |
|---|---|---|---|
| ROWS | $DC01 | → | I/O port outputting keys |
| COLM | $DC00 | → | I/O port driving C64 keys |
| VIC #47 | $D02F | → | I/O port driving new keys |
| 8502 P6 | $0001 | → | I/O port sensing CAPS key |
| NDX | $D0 | → | keyboard buffer index |
| KEYD | $34A | → | keyboard buffer |
| XMAX | $A20 | → | keyboard buffer size |
| SHFLAG | $D3 | → | shift key status |
| RPTFLG | $A22 | → | repeat key enables |
| LOCKS | $F7 | → | pause, mode disables |

KEY is also found in the Editor jump table as SCNKEY at $C012.

12. $FFA2 SETTMO ;(reserved)

PREPARATION:

| | |
|---|---|
| Registers: | none |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | none |
| Memory: | TIMOUT ($A0E) |
| Flags: | none |

**EXAMPLE:**

```
LDA #value
JSR $FFA2        ;update TIMOUT byte
```

**SETTMO** is not used in the C128 but is included for compatibility and completeness. It is used in the C64 by the IEEE communication cartridge to disable I/O timeouts.

13. $FFA5 ACPTR ;serial:byte input

PREPARATION:

| | |
|---|---|
| Registers: | none |
| Memory: | system map |
| Flags: | none |
| Calls: | TALK |
| | TKSA (if necessary) |

RESULTS:

| | |
|---|---|
| Registers: | .A = data byte |
| Memory: | STATUS ($90) |
| Flags: | none |

**EXAMPLE:**

```
JSR $FFA5       ;input a byte from serial bus
STA data
```

**ACPTR** is a low-level serial I/O utility to accept a single byte from the current serial bus TALKer using full handshaking. To prepare for this routine, a device must first have been established as a TALKer (see TALK) and passed a secondary address if necessary (see TKSA). The byte is returned in .A. (Most applications should use the higher-level I/O routines; see BASIN and GETIN).

14. $FFA8 CIOUT ;serial: byte output

PREPARATION:

| | |
|---|---|
| Registers: | .A = data byte |
| Memory: | system map |
| Flags: | none |
| Calls: | LISTN |
| | SECND (if necessary) |

RESULTS:

| | |
|---|---|
| Registers: | .A used |
| Memory: | STATUS ($90) |
| Flags: | none |

**EXAMPLE:**

```
LDA data
JSR $FFA8       ;send a byte via serial bus
```

**CIOUT** is a low-level serial I/O utility to transmit a single byte to the current serial bus LISTNer using full handshaking. To prepare for this routine, a device must first have been established as a LISTNer (see LISTN) and passed a secondary address if necessary (see SECND). The byte is passed in .A. Serial output data is buffered by one character, with the last character being transmitted with EOI after a call to UNLSN. (Most applications should use the higher level I/O routines; see BSOUT.)

15. $FFAB UNTLK ;serial: send untalk

PREPARATION:

| | |
|---|---|
| Registers: | none |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A used |
| Memory: | STATUS ($90) |
| Flags: | none |

**EXAMPLE:**

```
JSR $FFAB       ;UNTALK serial device
```

**UNTLK** is a low-level Kernal serial bus routine that sends an UNTALK command to all serial bus devices. It commands all TALKing devices to stop sending data. (Most applications should use the higher-level I/O routines; see CLRCH.)

16. $FFAE UNLSN ;serial: send unlisten

PREPARATION:

| | |
|---|---|
| Registers: | none |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A used |
| Memory: | STATUS ($90) |
| Flags: | none |

**EXAMPLE:**

```
JSR $FFAE       ;UNLISTEN serial device
```

**UNLSN** is a low-level Kernal serial bus routine that sends an UNLISTEN command to all serial bus devices. It commands all LISTENing devices to stop reading data. (Most applications should use the higher-level I/O routines; see ICLRCH.)

17. $FFB1 LISTN ;serial: send listen command

PREPARATION:

| | |
|---|---|
| Registers: | .A = device (0–31) |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A used |
| Memory: | STATUS ($90) |
| Flags: | none |

**EXAMPLE:**

```
JSR $FFB1        ;command device to LISTEN
```

**LISTN** is a low-level Kernal serial bus routine that sends a LISTEN command to the serial bus device in .A. It commands the device to start reading data. (Most applications should use the higher-level I/O routines; see ICKOUT.)

18. $FFB4 TALK ;serial: send talk

PREPARATION:

| | |
|---|---|
| Registers: | .A = device (0–31) |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A used |
| Memory: | STATUS ($90) |
| Flags: | none |

**EXAMPLE:**

```
JSR $FFB4        ;command device to TALK
```

**TALK** is a low-level Kernal serial bus routine that sends a TALK command to the serial bus device in .A. It commands the device to start sending data. (Most applications should use the higher-level I/O routines; see ICHKIN.)

19. $FFB7 READSS ;read I/O status byte

PREPARATION:

| | |
|---|---|
| Registers: | none |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A = STATUS ($90 or $A14) |
| Memory: | STATUS cleared if RS-232 |
| Flags: | none |

**EXAMPLE:**

```
JSR $FFB7       ;STATUS for last I/O
```

**READSS** (alias **READST**) returns the status byte associated with the last I/O operation (serial, cassette or RS-232) performed. Serial and cassette tape operations update STATUS ($90) and RS-232 I/O updates RSSTAT ($A14). Note that to simulate an ACIA, RSSTAT is cleared after it is read via READSS. The last I/O operation is determined by the contents of FA ($BA); thus applications that drive I/O devices using the lower-level Kernal calls should not use READSS.

20. $FFBA SETLFS ;set channel LA, FA, SA

PREPARATION:

| | |
|---|---|
| Registers: | .A = LA (logical #) |
| | .X = FA (device #) |
| | .Y = SA (secondary adr) |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | none |
| Memory: | LA, FA, SA updated |
| Flags: | none |

**EXAMPLE:**

See OPEN

**SETLFS** sets the logical file number (LA, $B8), device number (FA, $BA) and secondary address (SA, $B9) for the higher-level Kernal I/O routines. The LA must be unique among OPENed files and is used to identify specific files for I/O

operations. The device number range is 0 to 31 and is used to target I/O. The SA is a command to be sent to the indicated device, usually to place it in a particular mode. If the SA is not needed, the .Y register should pass $FF. SETLFS is often used along with SETNAM and SETBNK calls prior to OPENs. See the Kernal OPEN, LOAD and SAVE calls for examples.

21. $FFBD SETNAM ;set filename pointers

PREPARATION:

| | |
|---|---|
| Registers: | .A = string length |
| | .X = string adr low |
| | .Y = string adr high |
| Memory: | system map |
| Flags: | none |
| Calls: | SETBNK |

RESULTS:

| | |
|---|---|
| Registers: | none |
| Memory: | FNLEN, FNADR updated |
| Flags: | none |

**EXAMPLE:**

See OPEN

**SETNAM** sets up the filename or command string for higher-level Kernal I/O calls such as OPEN, LOAD and SAVE. The string (filename or command) length is passed in .A and updates FNLEN ($B7). The address of the string is passed in .X (low) and .Y (high). See the companion call, SETBNK, which specifies in which RAM bank the string is found. If there is no string, SETNAM should still be called and a null ($00) length specified (the address does not matter). SETNAM is often used along with SETBNK and SETLFS calls prior to OPENs. See the Kernal OPEN, LOAD and SAVE calls for examples.

22. $FFC0 OPEN ;open logical file

PREPARATION:

| | |
|---|---|
| Registers: | none |
| Memory: | system map |
| Flags: | none |
| Calls: | SETLFS, SETNAM, SETBNK |

RESULTS:

| | |
|---|---|
| Registers: | .A = error code (if any) |
| | .X used |
| | .Y used |
| Memory: | setup for I/O |
| | STATUS, RSSTAT updated |
| Flags: | .C = 1 → error |

**EXAMPLE:** OPEN 1,8,15,"I0"

```
LDA  #length       ;fnlen
LDX  #<filename    ;fnadr (command)
LDY  #>filename
JSR  $FFBD         ;SETNAM

LDX  #0            ;fnbank (Ram0)
JSR  $FF68         ;SETBNK

LDA  #1            ;la
LDX  #8            ;fa
LDY  #15           ;sa
JSR  $FFBA         ;SETLFS

JSR  $FFC0         ;OPEN

BCS  error

filename .BYTE 'I0'
length   = 2
```

**OPEN** prepares a logical file for I/O operations. It creates a unique entry in the Kernal logical file tables LAT ($362), FAT ($36C) and SAT ($376) using its index LDTND ($98) and data supplied by the user via SETLFS. There can be up to ten logical files OPENed simultaneously. OPEN performs device-specific opening tasks for serial, cassette and RS-232 devices, including clearing the previous status and transmitting any given filename or command string supplied by the user via SETNAM and SETBNK. The I/O status is updated appropriately and can be read via READSS.

The path to OPEN is through an indirect RAM vector at $31A. Applications may therefore provide their own OPEN procedures or supplement the system's by redirecting this vector to their own routine.

23. $FFC3 CLOSE ;close logical file

PREPARATION:

| | |
|---|---|
| Registers: | .A = LA (logical #) |
| Memory: | system map |
| Flags: | .C (see text below) |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A = error code (if any) |
| | .X used |
| | .Y used |
| Memory: | logical tables updated |
| | STATUS, RSSTAT updated |
| Flags: | .C = 1 → error |

**EXAMPLE:**

```
LDA #1          ;la
JSR $FFC3       ;CLOSE
BCS error       ;(tape files only)
```

**CLOSE** removes the logical file (LA) passed in .A from the logical file tables and performs device-specific closing tasks. Keyboard, screen and any unOPENed files pass through. Cassette files opened for output are closed by writing the last buffer and (optionally) an EOT mark. RS-232 I/O devices are reset, losing any buffered data. Serial files are closed by transmitting a CLOSE command (if an SA was given when it was opened), sending any buffered character, and UNLSTNing the bus.

There is a special provision incorporated into the CLOSE routine of systems featuring a BASIC DOS command. If the following conditions are all *true*, a full CLOSE is *not* performed; the table entry is removed but a CLOSE command is *not* transmitted to the device. This allows the disk command channel to be properly OPENed and CLOSEd without the disk operating system closing *all* files on its end:

```
.C  =  1   → indicates special CLOSE
FA >=8     → device is a disk
SA  =  15  → command channel
```

The path to CLOSE is through an indirect RAM vector at $31C. Applications may therefore provide their own CLOSE procedures or supplement the system's by redirecting this vector to their own routine.

24. $FFC6 CHKIN ;set input channel

PREPARATION:

| | |
|---|---|
| Registers: | .X = LA (logical #) |
| Memory: | system map |
| Flags: | none |
| Calls: | OPEN |

RESULTS:

| | |
|---|---|
| Registers: | .A = error code (if any) |
| | .X used |
| | .Y used |
| Memory: | LA, FA, SA, DFLTN |
| | STATUS, RSSTAT updated |
| Flags: | .C = 1 → error |

**EXAMPLE:**

```
LDX #1          ;la
JSR  $FFC6      ;CHKIN
BCS  error
```

**CHKIN** establishes an input channel to the device associated with the logical address (LA) passed in .X, in preparation for a call to BASIN or GETIN. The Kernal variable DFLTN ($99) is updated to indicate the current input device and the variables LA, FA and SA are updated with the file's parameters from its entry in the logical file tables (put there by OPEN). CHKIN performs certain device specific tasks: screen and keyboard channels pass through, cassette files are confirmed for input, and serial channels are sent a TALK command and the SA transmitted (if necessary). Call CLRCH to restore normal I/O channels.

CHKIN is required for all input except the keyboard. If keyboard input is desired and no other input channel is established, you do not need to call CHKIN or OPEN. The keyboard is the default input device for BASIN and GETIN.

The path to CHKIN is through an indirect RAM vector at $31E. Applications may therefore provide their own CHKIN procedures or supplement the system's by redirecting this vector to their own routine.

25. $FFC9 CKOUT ;set output channel

PREPARATION:

| | |
|---|---|
| Registers: | .X = LA (logical #) |
| Memory: | system map |
| Flags: | none |
| Calls: | OPEN |

RESULTS:

| | |
|---|---|
| Registers: | .A = error code (if any) |
| | .X used |
| | .Y used |
| Memory: | LA, FA, SA, DFLTO |
| | STATUS, RSSTAT updated |
| Flags: | .C = 1 → error |

**EXAMPLE:**

```
LDX #1          ;la
JSR $FFC9       ;CKOUT
BCS error
```

**CKOUT** establishes an output channel to the device associated with the logical address (LA) passed in .X, in preparation for a call to BSOUT. The Kernal variable DFLTO ($9A) is updated to indicate the current output device and the variables LA, FA and SA are updated with the file's parameters from its entry in the logical file tables (put there by OPEN). CKOUT performs certain device specific tasks: keyboard channels are illegal, screen channels pass through, cassette files are confirmed for output, and serial channels are sent a LISTN command and the SA transmitted (if necessary). Call CLRCH to restore normal I/O channels.

CKOUT is required for all output except the screen. If screen output is desired and no other output channel is established, you do not need to call CKOUT or OPEN. The screen is the default output device for BSOUT.

The path to CKOUT is through an indirect RAM vector at $320. Applications may therefore provide their own CKOUT procedures or supplement the system's by redirecting this vector to their own routine.

26. $FFCC CLRCH ;restore default channels

PREPARATION:

| | |
|---|---|
| Registers: | none |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A used |
| | .X used |
| Memory: | DFLTI, DFLTO updated |
| Flags: | none |

**EXAMPLE:**

```
JSR $FFCC       ;restore default I/O
```

**CLRCH** (alias CLRCHN) is used to clear all open channels and restore the system default I/O channels after other channels have been established via CHKIN and/or CHKOUT. The keyboard is the default input device and the screen is the default output device. If the input channel was to a serial device, CLRCH first UNTLKs it. If the output channel was to a serial device, it is UNLSNed first.

The path to CLRCH is through an indirect RAM vector at $322. Applications may therefore provide their own CLRCH procedures or supplement the system's by redirecting this vector to their own routine.

27. $FFCF BASIN ;input from channel

PREPARATION:

| | |
|---|---|
| Registers: | none |
| Memory: | system map |
| Flags: | none |
| Calls: | CHKIN (if necessary) |

RESULTS:

| | |
|---|---|
| Registers: | .A = character (or error code) |
| Memory: | STATUS, RSSTAT updated |
| Flags: | .C = 1 if error |

**EXAMPLE:**

```
      LDY #0          ;index
MORE  JSR $FFCF       ;input a character
      STA data,Y      ;buffer it
      INY
      CMP #$0D        ;carriage return?
      BNE MORE
```

**BASIN** (alias CHRIN) reads a character from the current input device (DFLTN $99) and returns it in .A. Input from devices other than the keyboard (the default input device) must be OPENed and CHKINed. The character is read from the input buffer associated with the current input channel:

a. Cassette data is returned a character at a time from the cassette buffer at $B00, with additional tape blocks being read when necessary.
b. RS-232 data is returned a character at a time from the RS-232 input buffer at $C00, waiting until a character is received if necessary. If RSSTAT ($A14) is bad from a prior operation, input is skipped and null input (carriage return) is substituted.
c. Serial data is returned a character at a time directly from the serial bus, waiting until a character is sent if necessary. If STATUS ($90) is bad from a prior operation, input is skipped and null input (carriage return) is substituted.
d. Screen data is read from screen RAM starting at the current cursor position and ending with a pseudo carriage return at the end of the logical screen line. The way the BASIN routine is written, the end of line (EOL) is not recognized. Users must therefore count characters themselves or otherwise detect when the logical EOL has been reached.

e. Keyboard data is input by turning on the cursor, reading characters from the keyboard buffer, and echoing them on the screen until a carriage return is encountered. Characters are then returned one at a time from the screen until all characters input have been passed, including the carriage return. Any calls after the EOL will start the process over again.

The path to BASIN is through an indirect RAM vector at $324. Applications may therefore provide their own BASIN procedures or supplement the system's by redirecting this vector to their own routine.

28. $FFD2 BSOUT ;output to channel

PREPARATION:

| | |
|---|---|
| Registers: | .A = character |
| Memory: | system map |
| Flags: | none |
| Calls: | CKOUT (if necessary) |

RESULTS:

| | |
|---|---|
| Registers: | .A = error code (if any) |
| Memory: | STATUS, RSSTAT updated |
| Flags: | .C = 1 if error |

**EXAMPLE:**

```
LDA #character
JSR $FFD2       ;output a character
```

**BSOUT** (alias CHROUT) writes the character in .A to the current output device (DFLTO $9A). Output to devices other than the screen (the default output device) must be OPENed and CKOUTed. The character is written to the output buffer associated with the current output channel:

a. Cassette data is put a character at a time into the cassette buffer at $B00, with tape blocks being written when necessary.
b. RS-232 data is put a character at a time into the RS-232 output buffer at $D00, waiting until there is room if necessary.
c. Serial data is passed to CIOUT, which buffers one character and sends the previous character.
d. Screen data is put into screen RAM at the current cursor position.
e. Keyboard output is illegal.

The path to BSOUT is through an indirect RAM vector at $326. Applications may therefore provide their own BSOUT procedures or supplement the system's by redirecting this vector to their own routine.

29. $FFD5 LOAD ;load from file

PREPARATION:

| | |
|---|---|
| Registers: | .A = 0 → LOAD |
| | .A > 0 → VERIFY |
| | .X = load adr lo (if SA = 0) |
| | .Y = load adr hi (if SA = 0) |
| Memory: | system map |
| Flags: | none |
| Calls: | SETLFS, SETNAM, SETBNK |

RESULTS:

| | |
|---|---|
| Registers: | .A = error code (if any) |
| | .X = ending adr lo |
| | .Y = ending adr hi |
| Memory: | per command |
| | STATUS updated |
| Flags: | .C = 1 → error |

**EXAMPLE:** **LOAD "program",8,1**

```
        LDA #length     ;fnlen
        LDX #<filename  ;fnadr
        LDY #>filename
        JSR $FFBD       ;SETNAM

        LDA #0          ;load/verify bank (RAM 0)
        LDX #0          ;fnbank (RAM 0)
        JSR $FF68       ;SETBNK

        LDA #0          ;la (not used)
        LDX #8          ;fa
        LDY #$FF        ;sa (SA>0 normal load)
        JSR $FFBA       ;SETLFS

        LDA #0          ;load, not verify
        LDX #<load adr  ;(used only if SA = 0)
        LDY #>load adr  ;(used only if SA = 0)
        JSR $FFD5       ;LOAD
        BCS error
        STX end lo
        STY end hi

filename .BYTE "program"
length   = 7
```

This routine **LOADs** data from an input device into C128 memory. It can also be used to VERIFY that data in memory matches that in a file. LOAD performs

device-specific tasks for serial and cassette LOADs. You cannot LOAD from RS-232 devices, the screen or the keyboard. While LOAD performs all the tasks of an OPEN, it does *not* create any logical files as an OPEN does. Also note that LOAD cannot "wrap" memory banks. As with any I/O, the I/O status is updated appropriately and can be read via READSS. LOAD has two options that the user must select:

a. LOAD vs. VERIFY: The contents of .A passed at the call to LOAD determines which mode is in effect. If .A is zero, a LOAD operation will be performed and memory will be overwritten. If .A is nonzero, a VERIFY operation will be performed and the result passed via the error mechanism.
b. LOAD ADDRESS: the secondary address (SA) setup by the call to SETLFS determines where the LOAD starting address comes from. If the SA is zero, the user wants the address in .X and .Y at the time of the call to be used. If the SA is nonzero, the LOAD starting address is read from the file header itself and the file is loaded into the same place from which it was SAVEd.

The C128 serial LOAD routine automatically attempts to BURST load a file, and resorts to the normal load mechanism (but still using the FAST serial routines) if the BURST handshake is not returned.

The path to LOAD is through an indirect RAM vector at $330. Applications may therefore provide their own LOAD procedures or supplement the system procedures by redirecting this vector to their own routine.

30. $FFD8 SAVE ;save to file

PREPARATION:

| | |
|---|---|
| Registers: | .A = pointer to start adr |
| | .X = end adr lo |
| | .Y = end adr hi |
| Memory: | system map |
| Flags: | none |
| Calls: | SETLFS, SETNAM, SETBNK |

RESULTS:

| | |
|---|---|
| Registers: | .A = error code (if any) |
| | .X = used |
| | .Y = used |
| Memory: | STATUS updated |
| Flags: | .C = 1 → error |

**EXAMPLE:** SAVE "program",8

```
LDA #length      ;fnlen
LDX #<filename   ;fnadr
LDY #>filename
JSR $FFBD        ;SETNAM
```

```
        LDA #0          ;save from bank (RAM 0)
        LDX #0          ;fnbank (RAM 0)
        JSR $FF68       ;SETBNK

        LDA #0          ;la (not used)
        LDX #8          ;fa
        LDY #0          ;sa (cassette only)
        JSR $FFBA       ;SETLFS

        LDA #start      ;pointer to start address
        LDX end         ;ending address lo
        LDY end+1       ;ending adr hi
        JSR $FFD8       ;SAVE
        BCS error

filename .BYTE ''program''
length  = 7
start   .WORD address1  ;page-0
end     .WORD address2
```

This routine **SAVEs** data from C128 memory to an output device. SAVE performs device-specific tasks for serial and cassette SAVEs. You cannot SAVE from RS-232 devices, the screen or the keyboard. While SAVE performs all the tasks of an OPEN, it does *not* create any logical files as an OPEN does. The starting address of the area to be SAVEd must be placed in a zero-page vector and the address of this vector passed to SAVE in .A at the time of the call. The address of the last byte to be SAVEd PLUS ONE is passed in .X and .Y at the same time. Cassette SAVEs utilize the secondary address (SA) to specify the type of tape header(s) to be generated:

```
SA (bit 0) = 0 → relocatable  (blf) file
           = 1 → absolute     (plf) file
SA (bit 1) = 0 → normal end
           = 1 → write EOT header at end
```

There is no BURST save; the normal FAST serial routines are used. As with any I/O, the I/O status will be updated appropriately and can be read via READSS.

The path to SAVE is through an indirect RAM vector at $332. Applications may therefore provide their own SAVE procedures or supplement the system's by redirecting this vector to their own routine.

31. $FFDB SETTIM ;set internal clock

PREPARATION:

| | |
|---|---|
| Registers: | .A = low byte |
| | .X = middle byte |
| | .Y = high byte |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | none |
| Memory: | TIME ($A0) updated |
| Flags: | none |

**EXAMPLE:**

```
LDA #0          ;reset clock
TAX
TAY
JSR $FFDB       ;SETTIM
```

**SETTIM** sets the system software (jiffie) clock, which counts sixtieths (1/60) of a second. The timer is incremented during system IRQ processing (see UDTIM), and reset at the 24-hour point. SETTIM disables IRQ's, updates the three-byte timer with the contents of .A, .X and .Y, and re-enables IRQ's.

32. $FFDE RDTIM ;read internal clock

PREPARATION:

| | |
|---|---|
| Registers: | none |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A = low byte |
| | .X = middle byte |
| | .Y = high byte |
| Memory: | none |
| Flags: | none |

**EXAMPLE:**

```
JSR $FFDE       ;RDTIM
```

**RDTIM** reads the system software (jiffie) clock, which counts sixtieths (1/60) of a second. The timer is incremented during system IRQ processing (see UDTIM), and reset at the 24-hour point. RDTIM disables IRQ's, loads .A, .X and .Y with the contents of the 3-byte timer, and re-enables IRQ's.

33. $FFE1 STOP ;scan STOP key

PREPARATION:

| | |
|---|---|
| Registers: | none |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A = last keyboard row |
| | .X = used (if STOP key) |
| Memory: | none |
| Flags: | status valid |

**EXAMPLE:**

```
JSR $FFE1       ;scan STOP key
BEQ stop        ;branch if down
```

**STOP** checks a Kernal variable STKEY ($91), which is updated by UDTIM during normal IRQ processing and contains the last scan of keyboard column C7. The STOP key is bit 7, which will be 0 if the key is down. If it is, default I/O channels are restored via CLRCH and the keyboard queue is flushed by resetting NDX ($D0). The keys on keyboard line C7 are:

| Bit: | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Key: | STOP | Q | C= | SPACE | 2 | CTRL | ← | 1 |

The path to STOP is through an indirect RAM vector at $328. Applications may therefore provide their own STOP procedures or supplement the system's by redirecting this vector to their own routine.

34. $FFE4 GETIN ;read buffered data

PREPARATION:

| | |
|---|---|
| Registers: | none |
| Memory: | system map |
| Flags: | none |
| Calls: | CHKIN (if necessary) |

RESULTS:

| | |
|---|---|
| Registers: | .A = character (or error code) |
| | .X used |
| | .Y used |
| Memory: | STATUS, RSSTAT updated |
| Flags: | .C = 1 if error |

**EXAMPLE:**

```
wait JSR $FFE4      ;get any key
     BEQ wait
     STA character
```

**GETIN** reads a character from the current input device (DFLTN ($99)) buffer and returns it in .A. Input from devices other than the keyboard (the default input device) must be OPENed and CHKINed. The character is read from the input buffer associated with the current input channel:

a. Keyboard input: A character is removed from the keyboard buffer and passed in .A. If the buffer is empty, a null ($00) is returned.
b. RS-232 input: A character is removed from the RS-232 input buffer at $C00 and passed in .A. If the buffer is empty, a null ($00) is returned (use READSS to check validity).
c. Serial input: GETIN automatically jumps to BASIN. See BASIN serial I/O.
d. Cassette input: GETIN automatically jumps to BASIN. See BASIN cassette I/O.
e. Screen input: GETIN automatically jumps to BASIN. See BASIN serial I/O.

The path to GETIN is through an indirect RAM vector at $32A. Applications may therefore provide their own GETIN procedures or supplement the system's by redirecting this vector to their own routine.

35. $FFE7 CLALL ;close all files and channels

PREPARATION:

| | |
|---|---|
| Registers: | none |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A used |
| | .X used |
| Memory: | LDTND, DFLTN, DFLTO updated |
| Flags: | none |

**EXAMPLE:**

```
JSR $FFE7       ;close files
```

**CLALL** deletes all logical file table entries by resetting the table index, LDTND ($98). It clears current serial channels (if any) and restores the default I/O channels via CLRCH.

The path to CLALL is through an indirect RAM vector at $32C. Applications may therefore provide their own CLALL procedures or supplement the system's by redirecting this vector to their own routine.

36. $FFEA UDTIM ;increment internal clock

PREPARATION:

| | |
|---|---|
| Registers: | none |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A used |
| | .X used |
| Memory: | TIME, TIMER, STKEY updated |
| Flags: | none |

**EXAMPLE:**

```
SEI
JSR $FFEA       ;UDTIM
CLI
```

**UDTIM** increments the system software (jiffie) clock, which counts sixtieths (1/60) of a second when called by the system 60Hz IRQ. TIME, a 3-byte counter located at $A0, is reset at the 24-hour point. UDTIM also decrements TIMER, also a 3-byte counter, located at $A1D (BASIC uses this for the SLEEP command, for example). You should be sure IRQ's are disabled before calling UDTIM to prevent system calls to UDTIM while you are modifying TIME and TIMER.

UDTIM also scans key line C7, on which the STOP key lies, and stores the result in STKEY ($91). The Kernal routine STOP utilizes this variable.

37. $FFED SCRORG ;get current screen window size

PREPARATION:

| | |
|---|---|
| Registers: | none |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A = screen width |
| | .X = window width |
| | .Y = window height |
| Memory: | none |
| Flags: | none |

**EXAMPLE:**

```
JSR $FFED       ;SCRORG
```

**SCRORG** is an Editor routine that has been slightly changed from previous CBM systems. Instead of returning the maximum SCREEN dimensions in .X and .Y, the C128 SCRORG returns the current WINDOW dimensions. It does return the maximum SCREEN width in .A. These changes make it possible for applications to "fit" themselves on the current screen window. SCRORG is also an Editor jump table entry ($C00F).

38. $FFF0 PLOT ;read/set cursor position

PREPARATION:

| | |
|---|---|
| Registers: | .X = cursor line |
| | .Y = cursor column |
| Memory: | system map |
| Flags: | .C = 0 → set cursor position |
| | .C = 1 → get cursor position |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .X = cursor line |
| | .Y = cursor column |
| Memory: | TBLX, PNTR updated |
| Flags: | .C = 1 → error |

**EXAMPLE:**

```
SEC
JSR $FFF0       ;read current position
INX             ;move down one line
INY             ;move right one space
CLC
JSR $FFF0       ;set cursor position
BCS error       ;new position outside window
```

**PLOT** is an Editor routine that has been slightly changed from previous CBM systems. Instead of using *absolute* coordinates when referencing the cursor position, PLOT now uses *relative* coordinates, based upon the current screen *window*. The following *local* Editor variables are useful:

| | | | |
|---|---|---|---|
| SCBOT | $E4 | → | window bottom |
| SCTOP | $E5 | → | window top |
| SCLF | $E6 | → | window left side |
| SCRT | $E7 | → | window right side |
| TBLX | $EB | → | cursor line |
| PNTR | $EC | → | cursor column |
| LINES | $ED | → | maximum screen height |
| COLUMNS | $EE | → | maximum screen width |

When called with the carry status set, PLOT returns the current cursor position relative to the current window origin (*not* screen origin). When called with the carry status clear, PLOT attempts to move the cursor to the indicated line and column relative to the current window origin (*not* screen origin). PLOT will return a clear carry status if the cursor was moved, and a set carry status if the requested position was outside the current window (*no change* has been made).

39. $FFF3 IOBASE ;read base address of I/O block

PREPARATION:

| | |
|---|---|
| Registers: | none |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .X = lsb of I/O block |
| | .Y = msb of I/O block |
| Memory: | none |
| Flags: | none |

**EXAMPLE:**

```
JSR $FFF3       ;find the I/O block
```

**IOBASE** is not used in the C128 but is included for compatibility and completeness. It returns the address of the I/O block in .X and .Y.

# NEW C128 KERNAL CALLS

The following system calls are a set of extensions to the standard CBM jump table. They are specifically for the C128 and as such should not be considered as permanent additions to the standard jump table. With the exception of C64 MODE, they are all true subroutines and will terminate via an RTS. As with all Kernal calls, the system configuration (high ROM, RAM-0 and I/O) must be in context at the time of the call.

1. $FF47 SPIN SPOUT ;setup fast serial ports for I/O

PREPARATION:

| | |
|---|---|
| Registers: | none |
| Memory: | system map |
| Flags: | .C = 0 → select SPINP |
| | .C = 1 → select SPOUT |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A used |
| Memory: | CIA-1, MMU |
| Flags: | none |

**EXAMPLE:**

```
CLC
JSR $FF47        ;setup for fast serial input
```

The C128/1571 fast serial protocol utilizes CIA 1 (6526 at $DC00) and a special driver circuit controlled in part by the MMU (at $D500). **SPINP** and **SPOUT** are routines used by the system to set up the CIA and fast serial driver circuit for input or output. SPINP sets up CRA (CIA 1 register 14) and clears the FSDIR bit (MMU register 5) for input. SPOUT sets up CRA, ICR (CIA 1 register 13), timer A (CIA 1 registers 4 and 5), and sets the FSDIR bit for output. Note the state of the TODIN bit of CRA is always preserved, but the state of the GAME, EXROM and SENSE40 outputs of the MMU are not (reading these ports return the state of the port and *not* the register values—consequently they cannot be preserved). These routines are required only by applications driving the fast serial bus themselves from the lowest level.

2. $FF4A CLOSE ALL ;close all files on a device

PREPARATION:

| | |
|---|---|
| Registers: | .A → device # (FA: 0–31) |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A used |
| | .X used |
| | .Y used |
| Memory: | none |
| Flags: | none |

**EXAMPLE:**

```
LDA #$08
JSR $FF4A        ;close all files on device 8
```

The **FAT** is searched for the given FA. A proper CLOSE is performed for all matches. If one of the CLOSEd channels is the current I/O channel, then the default channel is restored.

This call is utilized, for example, by the BASIC command **DCLOSE**. It is also called by the Kernal BOOT routine.

3. $FF4D C64MODE ;reconfigure system as a C64

PREPARATION:

| | |
|---|---|
| Registers: | none |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | none |
| Memory: | none |
| Flags: | none |

**EXAMPLE:**

```
JMP $FF4D       ;switch to C64 mode
```

*There is no return from this routine.* The 8502 DDR and port are initialized, and the VIC is set to 1MHz (slow) mode. Control is passed to code in common (shared) RAM, which sets the MMU mode register (#5) to C64 mode. From this point on, the MMU and C128 ROMs are not accessible. The routine exits via an indirect jump through the C64 RESET vector.

Since C64 operation does not allow for MMU access, all MMU registers must be configured for proper operation before the C64 mode bit is set. Similarly, because the start-up of the C64 operating system is not from a true hardware reset, there is the possibility that unusual I/O states in effect prior to C64MODE calls can cause unpredictable and presumably undesirable situations once in C64 mode.

There is no way to switch from C64 mode back to C128 mode; only a hardware reset or power off/on will restore the C128 mode of operation. A reset will always initiate C128 mode, although altering the SYSTEM vector beforehand is one way to automatically "throw" a system back to C64 mode.

4. $FF50 DMA CALL ;send command to DMA device

PREPARATION:

| | |
|---|---|
| Registers: | .X = bank (0–15) |
| | .Y = DMA controller command |
| Memory: | DMA registers set up system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A used |
| | .X used |
| Memory: | changed as per command |
| Flags: | none |

**EXAMPLE:**

```
LDA #$00        ;setup C128 base address
STA $DF02       ;low
LDA #$20
STA $DF03       ;high

LDA #$00        ;setup expansion RAM address
STA $DF04       ;low
STA $DF05       ;high
STA $DF06       ;bank (0–n, where n = 3 if 256K)

LDA #$40        ;setup number of bytes
STA $DF07       ;low
LDA #$1F
STA $DF08       ;high

LDX #$00        ;C128 bank
LDY #$84        ;DMA command to "STASH"
JSR $FF50       ;execute DMA command
```

**DMA CALL** is designed to communicate with an external expansion cartridge capable of DMA and mapped into system memory at IO2 ($DFxx). The DMA CALL converts the logical C128 bank parameter to MMU configuration via GETCFG, OR's in the I/O enable bit, and transfers control to RAM code at $3F0. Here the C128 bank specified is brought into context, and the user's command is issued to the DMA controller. The actual DMA transfer is performed at this point, with the 8502 kept off the bus in a wait state. As soon as the DMA controller releases the processor, memory is reconfigured to the state it was in at the time of the call and control is returned to the caller. The user must analyze the completion status by reading the DMA status register at $DF00.

Care should be taken in the utilization of the C128 RAM expansion product by any application using the built-in Kernal interface. This includes especially the use of the C128 BASIC commands FETCH, STASH and SWAP. In the routine that prepares a DMA request for the user, the Kernal forces the I/O block to be always in context. Consequently, data from the DMA device is likely to corrupt sensitive I/O devices. Users should either bypass the Kernal DMA routine by providing their own interface, or limit the DMA data transfers to the areas above and below the I/O block. Only strict observance of the latter will guarantee proper utilization of the BASIC commands. The following code, used instead of the DMA CALL in the above example, illustrates a work-around:

```
LDX #$00   ;C128 bank
LDY #$84   ;DMA command to 'STASH'
JSR  $FF6B ;GETCFG
TAX
JSR  $3F0  ;execute DMA command
```

5. $FF53 BOOT CALL ;boot load program from disk

PREPARATION:

| | |
|---|---|
| Registers: | .A = drive number (ASCII) |
| | .X = device number (0–31) |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A used |
| | .X used |
| | .Y used |
| Memory: | changed as per command |
| Flags: | .C → 1 if I/O error |

**EXAMPLE:**

```
LDA #$30        ;drive 0
LDX #$08        ;device 8
JSR  $FF53      ;BOOT
BCS IO ERROR
BCC NO BOOT SECTOR
```

**BOOT** attempts to load and execute the boot sector from an auto-boot disk in the given drive and device. The BOOT protocol is as follows:

a. Close all open files on boot device.
b. Read track 1 sector 0 into TBUFFR ($B00).
c. Check for auto-boot header, RTS if not.
d. If (blk# > 0), BLOCK READ sequential sectors into RAM at given (adrl, adrh, bank) location.
e. If LEN(filename) > 0, LOAD file into RAM-0 (normal load).
f. JSR to user code at location C above.

On any error, the BOOT operation is aborted and the UI command is issued to the disk. A return may or may not be made to the caller depending upon the completion status and the BOOTed code. The BOOT sector has the following layout:

| $00 | $01 | $02 | $03 | $04 | $05 | $06 | | A | | B | C |
|---|---|---|---|---|---|---|---|---|---|---|---|
| C | B | M | adrl | adrh | bank | blk# | title | 0 | file | 0 | code |

where: A = $07 + LEN(title)
B = A + LEN(filename)
C = B + 1

The following examples illustrate the flexibility of this layout. This loads and runs a BASIC program:

```
$00  →  CBM                  :key
$03  →  $00,$00,$00,$00      :no other BOOT sector
$07  →  NAME,$00             :message "NAME"
$0C  →  $00                  :no filename
$0D  →  $A2,$13,$A0,$0B      :code
        $4C,$A5,$AF
$14  →  RUN"PROGRAM"         :data (BASIC stmt)
$20  →  $00
```

This results in the message **Booting NAME...** being displayed and, utilizing a C128 BASIC jump table entry that finds and executes a BASIC statement, loads and runs the BASIC program named "PROGRAM." The same header can be used to load and execute a binary (machine code) program by simply changing RUN to BOOT. (While the file auto-load feature of the boot header could be used to load binary files simply by furnishing a filename, to execute it you must know the starting address and JMP to it. BASIC's BOOT command does that, and allows a more generic mechanism.) In the next example, a menu is displayed and you are asked to select the operating mode. Nothing else is loaded in this "configure"-type header:

```
$00  →  CBM                  :key
$03  →  $00,$00,$00,$00      :no other BOOT sector
$07  →  $00                  :no message
$0C  →  $00                  :no filename
$0D  →  $20,$7D,$FF,$0D,$53,$45,$4C,$45
        $43,$54,$20,$4D,$4F,$44,$45,$3A
        $0D,$0D,$20,$31,$2E,$20,$43,$36
        $34,$20,$20,$42,$41,$53,$49,$43
        $0D,$20,$32,$2E,$20,$43,$31,$32
        $38,$20,$42,$41,$53,$49,$43,$0D
        $20,$33,$2E,$20,$43,$31,$32,$38
        $20,$4D,$4F,$4E,$49,$54,$4F,$52
        $0D,$0D,$00,$20,$E4,$FF,$C9,$31
        $D0,$03,$4C,$4D,$FF,$C9,$32,$D0
        $03,$4C,$03,$40,$C9,$33,$D0,$E3
        $4C,$00,$B0
```

The loading of sequential sectors is designed primarily for specialized applications (such as CP/M or games) that do not need a disk directory entry.

6. $FF56 PHOENIX ;init function cartridges

PREPARATION:

| | |
|---|---|
| Registers: | none |
| Memory: | cartridge map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A used |
| | .X used |
| | .Y used |
| Memory: | changed as per command |
| Flags: | none |

**EXAMPLE:**

```
JSR $FF56       ;PHOENIX
```

The C128 Kernal initialization routine POLL creates a **Physical Address Table** (**PAT**) containing the ID's of all installed function ROM cartridges. **PHOENIX** calls each logged cartridge's cold-start entry in the order: external low/high, and internal low/high. After calling the cartridges (if any), PHOENIX calls the Kernal BOOT routine to look for an auto-boot disk in drive 0 of device 8 (see BOOT CALL). Control may or may not be returned to the user. PHOENIX is called by BASIC at the conclusion of its cold initialization.

7. $FF59 LKUPLA ;search tables for given LA

8. $FF5C LKUPSA ;search tables for given SA

PREPARATION:

| | |
|---|---|
| Registers: | .A = LA (logical file number) if LKUPLA |
| | .Y = SA (secondary address) if LKUPSA |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A = LA (only if found) |
| | .X = FA (only if found) |
| | .Y = SA (only if found) |
| Memory: | none |
| Flags: | .C = 0 if found |
| | .C = 1 if not found |

**EXAMPLE:**

```
        LDY #$60          ;find an available SA
AGAIN   INY
        CPY #$6F
        BCS TOO MANY      ;too many files open
        JSR $FF5C         ;LKUPSA
        BCC AGAIN         ;get another if in use
```

**LKUPLA** and **LKUPSA** are Kernal routines used primarily by BASIC DOS commands to work around a user's open disk channels. The Kernal requires unique logical device numbers (LA's), and the disk requires unique secondary addresses (SA's); therefore BASIC must find alternative unused values whenever it needs to establish a disk channel.

9. $FF5F SWAPPER ;switch between 40 and 80 columns

PREPARATION:

| | |
|---|---|
| Registers: | none |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A used |
| | .X used |
| | .Y used |
| Memory: | local variables swapped |
| Flags: | none |

**EXAMPLE:**

```
LDA $D7          ;check display mode
BMI is_80        ;branch if 80-column
JSR $FF5F        ;switch from 40 to 80
```

**SWAPPER** is an Editor utility used to switch between the 40-column VIC (composite) video display and the 80-column 8563 (RGBI) video display. The routine simply swaps local (associated with a particular screen) variables, TAB tables and line wrap maps with those describing the other screen. The MSB of MODE, location $D7, is toggled by SWAPPER to indicate the current display mode: $80 = 80-column, $00 = 40-column.

10. $FF62 DLCHR ;init 80-col character RAM

PREPARATION:

| | |
|---|---|
| Registers: | none |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A used |
| | .X used |
| | .Y used |
| Memory: | 8563 character RAM initialized |
| Flags: | none |

**EXAMPLE:**

```
JSR $FF62        ;initialize 8563 char. defns.
```

**DLCHR** (alias INIT80) is an Editor utility to copy the VIC character definitions from ROM ($D000–$DFFF, bank 14) to 8563 display RAM ($2000–$3FFF, local to 8563–not in processor address space). The 8 by 8 VIC character cells are padded with nulls ($00) to fill out the 8 by 16 8563 character cells. Refer to Chapter 10, Programming the 80-Column (8563) Chip for details concerning the 8563 font layout.

11. $FF65 PFKEY ;program a function key

PREPARATION:

| | |
|---|---|
| Registers: | .A = pointer to string adr (lo/hi/bank) |
| | .Y = string length |
| | .X = key number (1–10) |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A used |
| | .X used |
| | .Y used |
| Memory: | PKYBUF, PKYDEF tables updated |
| Flags: | .C = 0 if successful |
| | .C = 1 if no room available |

**EXAMPLE:**

```
LDA #$FA        ;pointer to string address
LDY #$06        ;length
LDX #$0A        ;key # (HELP key)
JSR  $FF65      ;install new key def'n
BCS  NO ROOM

>000FA  00  13  00          :ptr to $1300 bank 0
>01300  53  54  52  49  4E  47  :"string"
```

**PFKEY** (alias KEYSET) is an Editor utility to replace a C128 function key string with a user's string. Keys 1–8 are F1–F8, 9 is the **SHIFT** RUN string, and 10 is the HELP string. The example above replaces the "help" **RETURN** string assigned at system initialization to the **HELP** key with the string "string." Both the key length table, PKYBUF ($1000–$1009), and the definition area, PKYDEF ($100A–$10FF) are compressed and updated. The maximum length of all ten strings is 246 characters. No change is made if there is insufficient room for a new definition.

12. $FF68 SETBNK ;set bank for I/O operations

PREPARATION:

| | |
|---|---|
| Registers: | .A = BA, memory bank (0–15) |
| | .X = FNBANK, filename bank |
| Memory: | system map |
| Flags: | none |
| Calls: | SETNAM |

RESULTS:

| | |
|---|---|
| Registers: | none |
| Memory: | BA, FNBANK updated |
| Flags: | none |

**EXAMPLE:**

See OPEN

**SETBNK** is a prerequisite for any memory I/O operations, and must be used along with SETLFS and SETNAM prior to OPENing files, etc. BA ($C6) sets the current 64KB memory bank for LOAD/SAVE/VERIFY operations. FNBANK ($C7) indicates the bank in which the filename string is found. The Kernal routine GETCFG is used to translate the given logical bank numbers (0–15). SETBNK is often used along with SETNAM and SETLFS calls prior to OPEN's. See the Kernal OPEN, LOAD and SAVE calls for examples.

13. $FF6B GETCFG ;lookup MMU data for given bank

PREPARATION:

| | |
|---|---|
| Registers: | .X = logical bank # (0–15) |
| Memory: | system map |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A = MMU configuration data |
| Memory: | none |
| Flags: | none |

**EXAMPLE:**

```
LDX #$00      ;logical bank 0 (RAM O)
JSR  $FF6B    ;GETCFG
STA  $FF01    ;setup MMU pre-config #1
```

**GETCFG** allows a universal, logical approach to physical bank numbers by providing a simple lookup conversion for obtaining the actual MMU configuration data. In all cases where a bank number 0–15 is required, you can expect GETCFG to be called to convert that number accordingly. There is *no* error checking; if the given logical bank number is out of range the result is invalid. Refer to the Memory Management Unit in the Commodore 128 section later in this chapter for details concerning memory configuration. The C128 Kernal memory banks are assigned as follows:

```
 0.  %00111111  :RAM 0 only
 1.  %01111111  :RAM 1 only
 2.  %10111111  :RAM 2 only
 3.  %11111111  :RAM 3 only
 4.  %00010110  :INT ROM,  RAM 0, I/O
 5.  %01010110  :INT ROM,  RAM 1, I/O
 6.  %10010110  :INT ROM,  RAM 2, I/O
 7.  %11010110  :INT ROM,  RAM 3, I/O
 8.  %00101010  :EXT ROM, RAM 0, I/O
 9.  %01101010  :EXT ROM, RAM 1, I/O
10.  %10101010  :EXT ROM, RAM 2, I/O
11.  %11101010  :EXT ROM, RAM 3, I/O
12.  %00000110  :KERNAL, INT LO,  RAM 0, I/O
13.  %00001010  :KERNAL, EXT LO, RAM 0, I/O
14.  %00000001  :KERNAL, BASIC,    RAM 0, CHAR ROM
15.  %00000000  :KERNAL, BASIC,    RAM 0, I/O
```

14. $FF6E JSRFAR ;gosub in another bank

15. $FF71 JMPFAR ;goto another bank

PREPARATION:

| | |
|---|---|
| Registers: | none |
| Memory: | system map, also: |
| | $02 → bank (0–15) |
| | $03 → PC high |
| | $04 → PC low |
| | $05 → .S (status) |
| | $06 → .A |
| | $07 → .X |
| | $08 → .Y |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | none |
| Memory: | as per call, also: |
| | $05 → .S (status) |
| | $06 → .A |
| | $07 → .X |
| | $08 → .Y |
| Flags: | none |

The two routines, **JSRFAR** and **JMPFAR**, enable code executing in the system bank of memory to call (or JMP to) a routine in any other bank. In the case of JSRFAR, a return will be made to the caller's bank. It should be noted that JSRFAR calls JMPFAR, which calls GETCFG. When calling a non-system bank, the user should take necessary precautions to ensure that interrupts (IRQ's and NMI's) will be handled properly (or disabled beforehand). Both JSRFAR and JMPFAR are RAM-based routines located in common (shared) RAM at $2CD and $2E3 respectively.

The following code illustrates how to call a subroutine in the second RAM bank from the system bank. Note that we need not worry about IRQ's and NMI's in this case because the system will handle them properly in any configuration that has the Kernal ROM or any valid RAM bank in context at the top page of memory.

```
STY $08     ;assumes registers and status
STX $07     ;already setup for call
STA $06
PHP
PLA
STA $05
```

```
LDA #1      ;want to call $2000 in bank 1
LDY #$20
LDX #$00
STA $02
STY $03
STX $04

JSR $FF6E   ;JSRFAR

LDA $05     ;restore status and registers
PHA
LDA $06
LDX $07
LDY $08
PLP
```

16. $FF74 INDFET ;LDA (fetvec),Y from any bank

PREPARATION:

| | |
|---|---|
| Registers: | .A = pointer to address |
| | .X = bank (0–15) |
| | .Y = index |
| Memory: | setup indirect vector |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .A = data |
| | .X used |
| Memory: | none |
| Flags: | status valid |

**EXAMPLE:**

```
LDA #$00    ;setup to read $2000
STA $FA
LDA #$20
STA $FB
LDA #$FA
LDX #$01    ;in bank 1
LDY #$00
JSR $FF74   ;LDA ($FA,RAM 1),Y
BEQ etc
```

**INDFET** enables applications to read data from any other bank. It sets up FETVEC ($2AA), calls GETCFG to convert the bank number, and JMPs to code in

common (shared) RAM at $2A2 which switches banks, loads the data, restores the user's bank, and returns. When calling a non-system bank, the user should take necessary precautions to ensure that interrupts (IRQ's and NMI's) will be handled properly (or disabled beforehand).

17. $FF77 INDSTA ;STA (stavec),Y to any bank

PREPARATION:

| | |
|---|---|
| Registers: | .A = data |
| | .X = bank (0–15) |
| | .Y = index |
| Memory: | setup indirect vector |
| | setup STAVEC ($2B9) pointer |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .X used |
| Memory: | changed per call |
| Flags: | status invalid |

**EXAMPLE:**

```
LDA #$00        ;setup write to $2000
STA $FA
LDA #$20
STA $FB
LDA #$FA
STA $2B9
LDA data
LDX #$01        ;in bank 1
LDY #$00
JSR $FF77       ;STA ($FA,RAM 1),Y
```

**INDSTA** enables applications to write data to any other bank. After you set up STAVEC ($2B9), it calls GETCFG to convert the bank number and JMPs to code in common (shared) RAM at $2AF which switches banks, stores the data, restores your bank, and returns. When calling a nonsystem bank, the user should take necessary precautions to ensure that interrupts (IRQ's and NMI's) will be handled properly (or disabled beforehand).

18. $FF7A INDCMP ;CMP (cmpvec),Y to any bank

PREPARATION:

| | |
|---|---|
| Registers: | .A = data |
| | .X = bank (0–15) |
| | .Y = index |
| Memory: | setup indirect vector |
| | setup CMPVEC ($2C8) pointer |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | .X used |
| Memory: | none |
| Flags: | status valid |

**EXAMPLE:**

```
LDA #$00        ;setup to verify $2000
STA $FA
LDA #$20
STA $FB
LDA #$FA
STA $2C3
LDA data
LDX #$01        ;in bank 1
LDY #$00
JSR $FF7A       ;CMP ($FA,RAM 1),Y
BEQ same
```

**CMPSTA** enables applications to compare data to any other bank. After you set up CMPVEC ($2C8), it calls GETCFG to convert the bank number and JMP's to code in common (shared) RAM at $2BE which switches banks, compares the data, restores your bank, and returns. When calling a nonsystem bank, the user should take necessary precautions to ensure that interrupts (IRQ's and NMI's) will be handled properly (or disabled beforehand).

19. $FF7D PRIMM ;print immediate utility

PREPARATION:

| | |
|---|---|
| Registers: | none |
| Memory: | none |
| Flags: | none |
| Calls: | none |

RESULTS:

| | |
|---|---|
| Registers: | none |
| Memory: | none |
| Flags: | none |

**EXAMPLE:**

```
JSR $FF7D       ;display following text

.BYTE "message"
.BYTE $00       ;terminator

JMP continue    ;execution resumes here
```

**PRIMM** is a Kernal utility used to print (to the default output device) an ASCII string which immediately follows the call. The string must be no longer than 255 characters and is terminated by a null ($00) character. It cannot contain any embedded null characters. Because PRIMM uses the system stack to find the string and a return address, you must *not* JMP to PRIMM. There must be a valid address on the stack.

# C128 DEVICE NUMBERS

The following are the device numbers for the Commodore 128:

0 → Keyboard
1 → Cassette
2 → RS-232
3 → Screen (current)
4 → Serial bus device:
    4–7 usually printers
    8–30 usually disks

Device number 31 should *not* be used. While it is specified to be a valid serial bus address, when it is ORed with certain serial commands it results in a bad command, hanging the bus and the serial drivers.

# MEMORY MANAGEMENT IN THE COMMODORE 128

## COMMODORE 128 MODE

In Commodore 128 mode, all memory management organization depends on the currently selected memory configuration. In C128 BASIC and the Machine Language Monitor, the memory is organized into sixteen default memory configurations. Different portions of memory are present depending on the memory configuration. Figure 13-3 lists the default memory configurations of the C128 on system power-up.

| BANK | CONFIGURATION |
|---|---|
| 0 | RAM(0) only |
| 1 | RAM(1) only |
| 2 | RAM(2) only (same as 0) |
| 3 | RAM(3) only (same as 1) |
| 4 | Internal ROM,RAM(0),I/O |
| 5 | Internal ROM,RAM(1),I/O |
| 6 | Internal ROM,RAM(2),I/O (same as 4) |
| 7 | Internal ROM,RAM(3),I/O (same as 5) |
| 8 | External ROM,RAM(0),I/O |
| 9 | External ROM,RAM(1),I/O |
| 10 | External ROM,RAM(2),I/O (same as 8) |
| 11 | External ROM,RAM(3),I/O (same as 9) |
| 12 | Kernal and Internal ROM (LOW), RAM(0),I/O |
| 13 | Kernal and External ROM (LOW), RAM(0),I/O |
| 14 | Kernal and BASIC ROM, RAM (0), Character ROM |
| 15 | Kernal and BASIC ROM, RAM(0), I/O |

**Figure 13–3. Memory Configuration (Bank) Table**

# THE MEMORY MANAGEMENT UNIT (MMU)

In Commodore 128 mode, all memory management is controlled through a series of I/O registers called the **Memory Management Unit (MMU)**. The Memory Management Unit consists of memory locations that reside between $D500 and $D50B and $FF00 and $FF04. The configuration register appears twice, once at $D500 and again at $FF00. This is done in case I/O is switched out of the $D000–$DFFF memory range, which makes the MMU registers from $D500–$D50B inaccessible.

When this occurs, some features of the MMU are no longer available to the programmer. So before switching out I/O in the $D000–$DFFF range, make sure you have made all the manipulations you need on the MMU (particularly, preselecting the preconfiguration register values for the load configuration registers). See Figure 13-4 for a graphic depiction of the way the MMU registers map into memory.

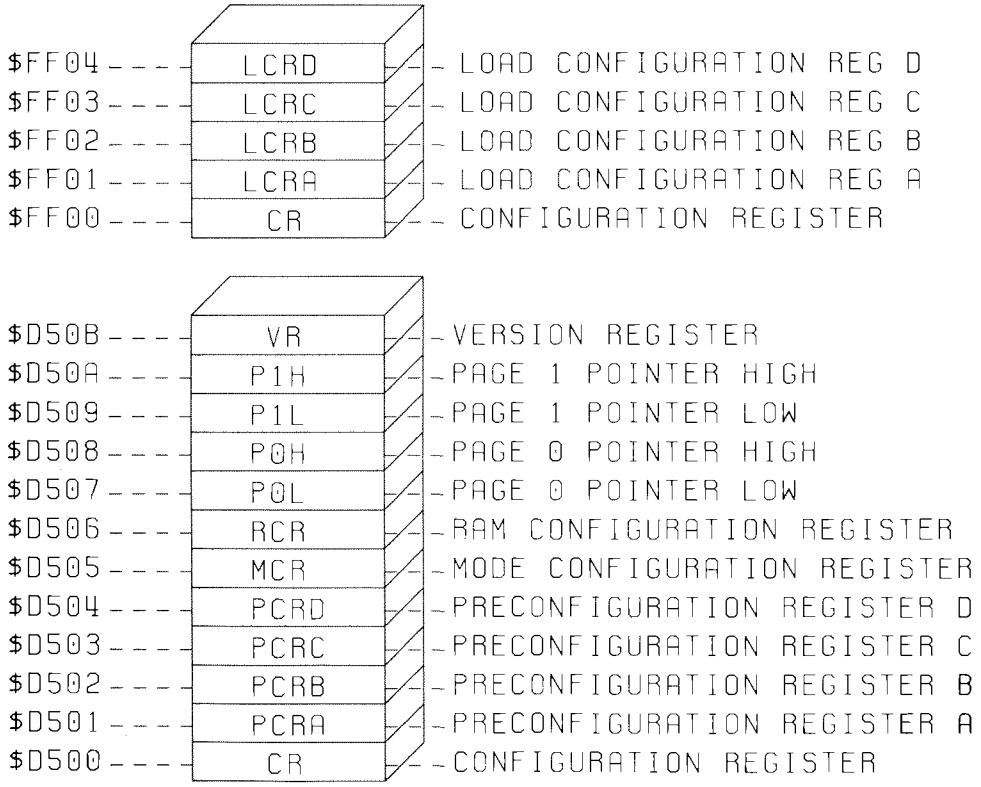



**Figure 13–4. MMU Register Map**

## HOW TO SWITCH BANKS

In BASIC, type the **BANK** command to switch from one bank to another as follows:

BANK n

where n is a digit between 0 and 15. When your application program needs to access a bank in which the microprocessor is not in context (for SYS, PEEK, POKE, and WAIT commands) use the BANK command to reach layers of the computer's memory that are not accessible in the current bank. For instance, suppose you want to access the VIC chip when you are doing graphics in the first part of your program. The next segment

of your program needs to access the character ROM, which is only visible to the 8502 in bank 14. Change to bank 14; then read the character ROM data, which makes up the images of the characters in the character sets. While the microprocessor is "looking" in bank 14, the VIC chip is not available to the microprocessor, so you must issue another BANK command in order to return to (a configuration containing I/O and) processing VIC video information.

In machine language, switching banks is a little more difficult. You must change the value of the registers [in particular the **Configuration Register (CR), Load Configuration Register (LCR)** and **Preconfiguration Register (PCR)**], either directly or indirectly. The Kernal routine GETCFG allows you to change configurations and maintains the same ones used by BASIC and the Monitor as they appear in Figure 13–3. There are four PCR's and four LCR's (as shown in Figure 13-4). Each PCR corresponds directly to a LCR. PCR A pertains to LCR A, PCR B corresponds to LCR B, and so on.

To change the value of the Configuration Register directly, perform a write operation (STA, STX, STY) to the Configuration Register.

To change the value of the Configuration Register indirectly, write a value specifying a memory configuration to the Preconfiguration Register. When a subsequent store instruction is performed to the corresponding Load Configuration Register, the value previously stored in the corresponding PCR is loaded into the CR. When a store instruction is executed on an LCR, the value in the corresponding PCR is loaded into the CR and the memory management organization conforms to the values associated with that memory management scheme. The value written to the LCR is of no consequence; any value triggers the preconfiguration mechanism.

> **NOTE:** Basic expects the LCR's and PCR's to be left alone. If you use them to manage memory in your application, do not plan on using BASIC.

## THE CONFIGURATION REGISTER

The Configuration Register (CR) is the most important register in the MMU. It specifies and organizes the ROM, RAM and input/output configurations for the entire Commodore 128 memory in C128 mode. (The MMU is not present at all in the C64 memory map.) The CR is located at address $D500 when I/O is available and at address $FF00 at all times. When I/O functions are disabled, the MMU memory disappears between $D500 and $D50B. The MMU memory is always present between $FF00 and $FF04. Each of the eight bits in the CR controls a separate memory function.

Bit 0 (zero) in the Configuration Register specifies whether I/O Registers are available, or whether ROM (High) is present in the memory range $D000 through $DFFF. The I/O Registers consist of the registers of the VIC chip, SID chip, MMU (from $D500 through $D50B); CIA number 1, which controls the joystick port; and CIA number 2, which controls the serial bus and user port. If bit 0 is high (equal to 1), ROM or RAM is present in the range $D000 through $DFFF, depending upon the values of the ROM HIGH bits (4 and 5) in this register. If bit 0 is low (equal to 0), I/O is present in this range. The value of bit 0 on power-up is 0.

When I/O is switched out of (not present in) this range, the registers in the MMU disappear from the memory map in the range $D500 through $D50B. The memory management is then controlled through the MMU registers at locations $FF00 through $FF04. Any write operation to an LCR ($FF01–$FF04) loads the corresponding PCR value (currently invisible to the 8502) into the CR. Reading an LCR returns the value stored in the currently inaccessible PCR. The MMU registers ranging from $FF00 through $FF04 are always present in the Commodore 128 memory, regardless of the memory configuration.

Bit 1 in the CR specifies how the microprocessor accesses the address range $4000 through $7FFF, called **ROM LOW** memory. If bit 1 is high, the microprocessor accesses RAM in this range. If bit 1 is low (equal to 0), the microprocessor maps in the **BASIC LOW ROM** in that range. Upon power-up or reset, this bit is set low, so BASIC is available to the user as soon as the computer is turned on.

Bits 2 and 3 determine the type of memory that resides in the midrange of memory, the address range $8000 through $BFFF. If both bits 2 and 3 are set high, RAM is placed in this range. If bit 2 is high and 3 is low, **INTERNAL FUNCTION ROM** is placed here. If bit 2 is low and 3 is high, **EXTERNAL FUNCTION ROM** appears. If bits 2 and 3 are low, the **BASIC HIGH ROM** is placed here. Upon power-up or reset, the MMU sets both bits 2 and 3 low, so BASIC is available to the user immediately.

Bits 4 and 5 work similarly to bits 2 and 3 and specify memory in the range $C000 through $FFFF, referred to as **HIGH** memory. If bits 4 and 5 are set high, RAM is placed in this range. If bit 4 is high and 5 is low, INTERNAL FUNCTION ROM is placed. If bit 4 is low and 5 is high, EXTERNAL FUNCTION ROM appears. If bits 4 and 5 are low, the Kernal and character ROMs are placed here. Upon power-up or reset, bits 4 and 5 are set low, so the Kernal and character ROM are available to the user at once.

Note that bit 0 in the Configuration Register, the bit that switches in and out I/O in address range $D000 through $DFFF, overrides the memory organization for bits 4 and 5. If bit 0 is set high (1), the I/O Registers are not in place in the address range $D000 through $DFFF. Either the character ROM, internal or external function ROM or RAM is located in this range, depending on the value of bits 4 and 5 of the configuration register. If bit 0 in the Configuration Register is set low (0), the Input/Output registers are present between $D000 through $DFFF, regardless of the value of bits 4 and 5 of the configuration register. This means no matter what memory configuration is chosen between $C000 and $FFFF with bits 4 and 5, if bit 0 is set low (0), the character ROM (or whatever was originally in this address range) is overlaid by the I/O registers and becomes unavailable to the microprocessor. This is why the character ROM and the I/O registers are never available at the same time.

Finally, the last two bits of the MMU, 6 and 7, determine the **RAM BANK** selection. For the base system of 128K, only bit 6 is significant; bit 7 is not implemented. When bit 6 is high (1), RAM bank 1 is selected. When bit 6 is low (0), RAM bank 0 is selected.

The 128K of RAM is organized into two 64K RAM banks. The microprocessor only addresses 64K at a time, but since the two 64K RAM banks can be switched in and out so quickly, the computer acts as though it addresses 128K at the same time. When one RAM bank is being addressed by the microprocessor, the other bank stores information to be processed once it is banked in. Portions of both banks can be shared in memory at the same time. This is called Common RAM, and is discussed in the section

The RAM Configuration Register. RAM bank 0 is used typically for BASIC text area, while RAM bank 1 is used for BASIC arrays and variable storage.

As indicated above, the MMU has a feature that allows a portion of RAM to be common for the two RAM banks. The RAM Configuration Register controls the amount of common RAM. This is discussed in detail later in this section under the description of the RAM Configuration Register.

Figure 13–5 is a diagram of the Configuration Register, showing how the bits control each memory organization.

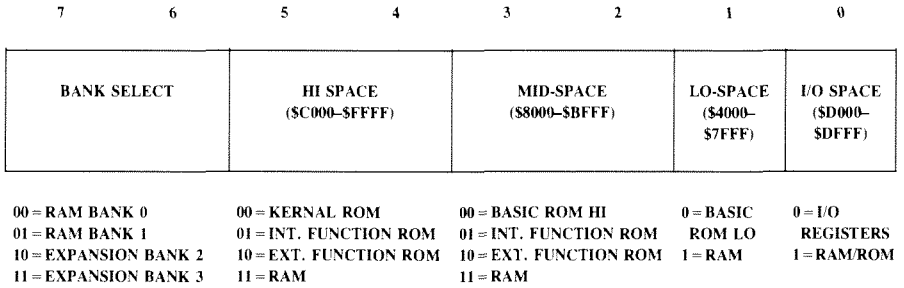



**Figure 13–5. Configuration Register**

## PRECONFIGURING COMMODORE 128 MEMORY

As noted, the Configuration Register in the MMU is the means by which the Commodore 128 organizes the memory layout. Commodore 128 memory management has a mechanism that allows the programmer to set up four predefined memory configurations besides the one already operating. The four Preconfiguration Registers (PCR) and the four Load Configuration Registers (LCR) are designed to provide this feature. When a different memory management structure is desired, a previously defined configuration in one of the Preconfiguration Registers can be chosen easily and instantly.

This mechanism allows the programmer to preset several memory configurations and, with a single store instruction to an LCR, reorganize the entire memory layout instantly. This enables the programmer to use several memory organizations interchangeably; thus, if one part of your program requires one memory configuration and another part requires a different configuration, you can switch back and forth between the two, each with a single store instruction, once you predefine the alternate memory setup in one of the Preconfiguration Registers.

To alter memory management through this preconfiguration mechanism, you must change the value of the Preconfiguration Register. The four PCR's and four LCR's of the MMU are shown in the MMU Register Map in Figure 13-6. Each PCR corresponds directly to an LCR (i.e., PCR A pertains to LCR A, PCR B

corresponds to LCR B, and so on). The format for the Load Configuration and Preconfiguration Registers is the same as for the Configuration Register. The four LCRs and four PCRs are all initialized to zero.

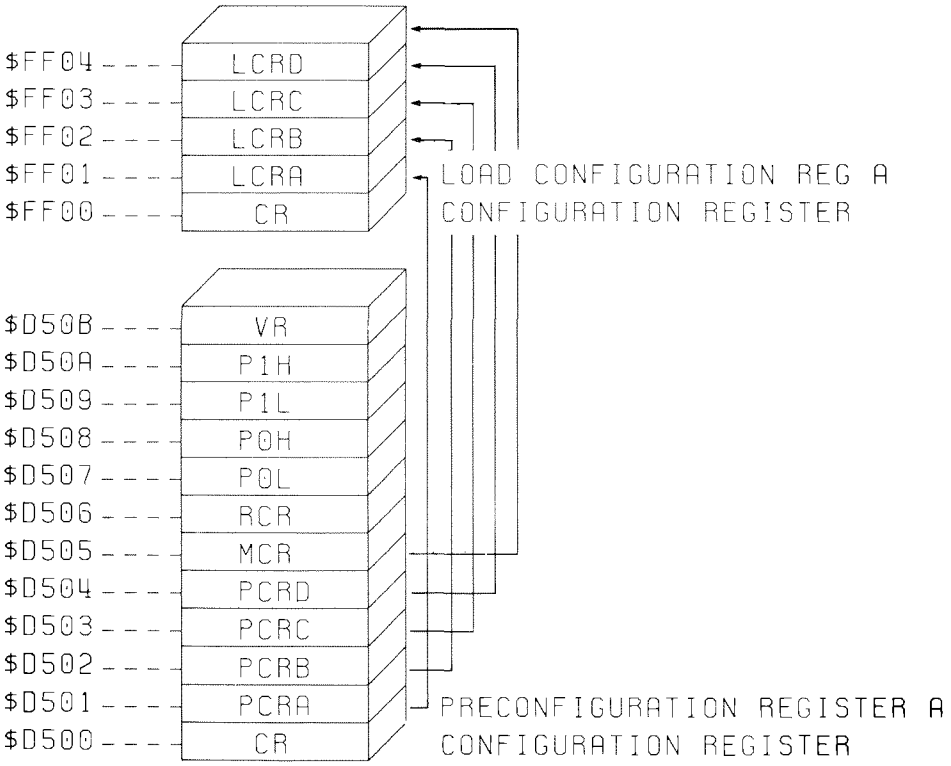



**Figure 13–6. MMU Register Map**

To *directly* change the value of the Configuration Register (therefore bypassing the preconfiguration mechanism), perform a write operation (STA, STX, STY) directly to the Configuration Register at either $D500 or $FF00.

To *indirectly* change the value of the Configuration Register (therefore utilizing the preconfiguration mechanism), perform a write operation (STA, STX, STY) to the Preconfiguration Register. This loads a value into the PCR. When a subsequent store instruction is performed to the corresponding Load Configuration Register, the value that was previously stored in the corresponding PCR is loaded into the CR. The store instruction acts as a triggering mechanism that passes the contents of a PCR into the CR. When a store instruction is executed upon an LCR, the value in the corresponding PCR is loaded into the CR and memory management organization conforms to the value associated with that PCR.

For example, to alter the memory management organization specified on power-up, use the following machine language segment:

```
PART 1   LDA #$0E      Load accumulator with 14
                         This selects
                       I/O ($D000–$DFFF)
                       RAM ($4000–$7FFF,$8000–$BFFF)
                       Kernal and Character ROM ($C000–$FFFF)
                       RAM Bank 0
         STA $D501     Store in PCR A—no immediate results
         :
         :             Place interim part of the program here
PART 2   STA $FF01     Write to LCR A, selects above configuration
```

In this program segment, PART 1 initializes PCR A ($D501) with the value 14 ($0E), which performs no immediate result. When PART 2 is encountered, the STA instruction performs a write operation to location $FF01, which triggers the preconfiguration mechanism and loads the value from PCR A into the Configuration Register. The store instruction value is not significant; it must operate only on the address, and any store instruction works. Once this instruction is executed, the memory management organization is immediately changed according to the value in the appropriate PCR, in this case PCR A ($D501).

The value 14 ($0E) loaded into PCR A selects the following memory organization:

| TYPE | ADDRESS RANGE | BIT(S) AFFECTED IN CR |
|---|---|---|
| **I/O** | **$D000–$DFFF** | **0** |
| **RAM** | **$4000–$7FFF** | **1** |
| **RAM** | **$8000–$BFFF** | **2,3** |
| **Kernal, Char** | **$C000–$FFFF** | **4,5** |
| **RAM BANK 0** | **——** | **6,7** |

**14 = $0E = 0000 1110 (binary)**

It is good practice to initialize the PCR's in the beginning of your program, as in PART 1 of the above program segment. Then place the interim portion of your program in memory. PART 2 marks the place in your program where you are actually going to alter the memory management setup. For the utmost speed and processor efficiency, use absolute addressing for the MMU registers, as in the above example.

As previously noted, when the Input/Output Registers are switched out, the registers of the MMU that appear in memory range $D500–$D50B are unavailable to the microprocessor. When this occurs, a write operation to an LCR still loads the corresponding PCR value (currently invisible to the 8502) into the CR, even though the PCR's are not accessible to the microprocessor when I/O is switched out. If you switch out I/O, make sure to set up the PCR's first, which are present when bit 0 of the Configuration Register is equal to 0. However, the Load Configuration Registers are available at all times since they appear in the address range $FF00–$FF04. If I/O is switched out, the PCR's must be

set prior to switching out I/O in order for them to be of any service. Note that BASIC uses the PCRs; if you alter them while BASIC is resident, you may obtain unpredictable results.

The MMU registers control the memory organization for RAM, BASIC, Kernal and character ROM, internal and external function ROM, and I/O. The MMU has additional registers that determine the mode (C128, C64 or CP/M) in which the Commodore 128 operates, the common RAM configurations, and the location of pages 0 and 1. The registers in the MMU that control these operations are the **Mode Configuration Register (MCR),** the **RAM Configuration Register (RCR),** and the **Page Pointers** respectively. The following sections explain how these additional registers of the Memory Management Unit operate.

## THE MODE CONFIGURATION REGISTER

The Mode Configuration Register (MCR) specifies which microprocessor is currently in operation (8502, Z80A) and which operating system mode is currently invoked (C128 or C64). The MCR is located at address $D505. As in the other registers in the MMU, each bit in the Mode Configuration Register controls a separate and independent operation.

Bit 0 determines which microprocessor is in control of the Commodore 128. Bit 0 is powered-up low so the Z80 microprocessor initiates control of the computer. The Z80 performs a small start-up procedure, then bit 0 is set to a 1 and the 8502 takes over if no CP/M system disk is present in the disk drive.

When the Commodore 128 is first powered-up or reset and the disk drive detects the CP/M operating system diskette in the drive, the Z80A microprocessor BOOTs the CP/M operating system from disk. The value of bit 0 in the Mode Configuration Register in this case is 0. When the Z80A takes control of the Commodore 128, all memory references from $0000 through $0FFF are translated to $D000 through $DFFF, where the CP/M BIOS exists in ROM. For memory accesses in the range $0000 through $0FFF in the Z80 BIOS, the memory status lines MS0 and MS1 are brought low to reflect ROM; otherwise they are high. Note that C64 mode and Z80A mode is an undefined configuration.

Bits 1 and 2 are not used. They are reserved for future expansion. IF0, bit 3 sets an input for FAST serial. It is not used as an input port at all.

Bit 3 is the fast serial (FSDIR) disk drive control bit. It acts like a bit in a bidirectional 6529 port, which means it acts differently depending upon whether the bit is used for input or output operation. As an output signal, bit 3 controls the direction of the data in the disk drive data buffer. The MMU pin FSDIR reflects the status of bit 3, which is reset to zero upon power-up. If bit 3 is equal to 1, an output operation occurs; selecting a data direction for the data in the serial bus buffer. If zero, bit 3 sets an input for FAST serial. It is not used as an input port at all.

Bits 4 and 5 are the /GAME and /EXROM sense bits respectively. These cartridge control lines initiate Commodore 64 mode and act as the /GAME and /EXROM hardware lines as in the Commodore 64. When these control lines detect a cartridge in the Commodore 128 expansion port, C64 mode is instantly enabled, the computer acts as a Commodore 64 and takes its instructions from the software built into the cartridge. Upon power-up, /GAME and /EXROM are pulled as inputs. If either one is low, C64

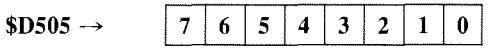
mode is selected. Thus, a C128 cartridge should not pull these lines low on power-up. In C128 mode, these lines are active as I/O lines (latched) to the expansion port. These can be used as input or output lines, but make sure they are not brought low upon power-up in C128 mode.

Bit 6 selects the operating system that takes over the Commodore 128. Upon power-up or reset, this bit is cleared (0) to enable all of the MMU registers and Commodore 128 mode features. Setting this bit high (1) initiates C64 mode.

Bit 7, a read-only bit, indicates whether the **40/80 DISPLAY** key is in the up (40-column) or down (80-column) position. The value of bit 7 is high (1) if the **40/80** key is in the up position. The value of bit 7 is low (0) if the **40/80 DISPLAY** key is in the down position.

This is useful in certain application programs that utilize both the 40- and 80-column displays. For instance, check the value of this bit to see if the user is viewing the VIC screen. If so, carry on with the program; otherwise display a message telling the user to switch from the 80-column screen to the VIC (40-column) screen in order to display a VIC bit map. This may be invalid if the user typed <ESC>X to switch screens. See the SCORG Kernal routine.

See Figure 13-7 for a summary of Mode Configuration Register activities.




**Mode Configuration Register**

| BIT | FUNCTION DESCRIPTION | VALUE HIGH | VALUE LOW |
|---|---|---|---|
| 0 | /Select microprocessor | 8502 | Z80A (inverted) |
| 1 | Unused | — | — |
| 2 | Unused | — | — |
| 3 | Fast serial DD control | Fast serial out | Fast serial in |
| 4 | /GAME Access game cartridge (C64 mode only) | | C64 mode set on power up |
| 5 | /EXROM Access external software cartridge (C64 mode only) | | C64 mode set on power up |
| 6 | Select operating system | C64 mode (MMU disappears from memory map) | C128 mode, assert MMU registers |
| 7 | Read **40/80** key position | 40/80 column key is UP | 40/80 column key is DOWN |

**Figure 13–7. Mode Configuration Register Summary**

## THE RAM CONFIGURATION REGISTER

The RAM Configuration Register (RCR) in the MMU specifies the amount of Common RAM shared between the two 64K RAM banks, how the RAM is shared, and which bank is delegated for the VIC chip. The value of the bits in the RCR determine how each RAM bank is allocated for specific purposes. The RAM Configuration Register is located within the I/O block at address $D506.

Bits 0 and 1 determine the amount of shared RAM between banks. If both bits 0 and 1 equal 0, 1K of RAM becomes common. If bit 0 is equal to 1 (high) and bit 1 is low (0), then 4K of common RAM is shared between banks. If bit 0 is low (0) and bit 1 is high (1), then 8K is common, and if both bits 2 and 3 are high (1), then 16K of RAM becomes common. (These bits have no effect in Commodore 64 mode). The reset values of these bits are both 0. See Figure 13-8 to understand how the two RAM banks share common RAM.

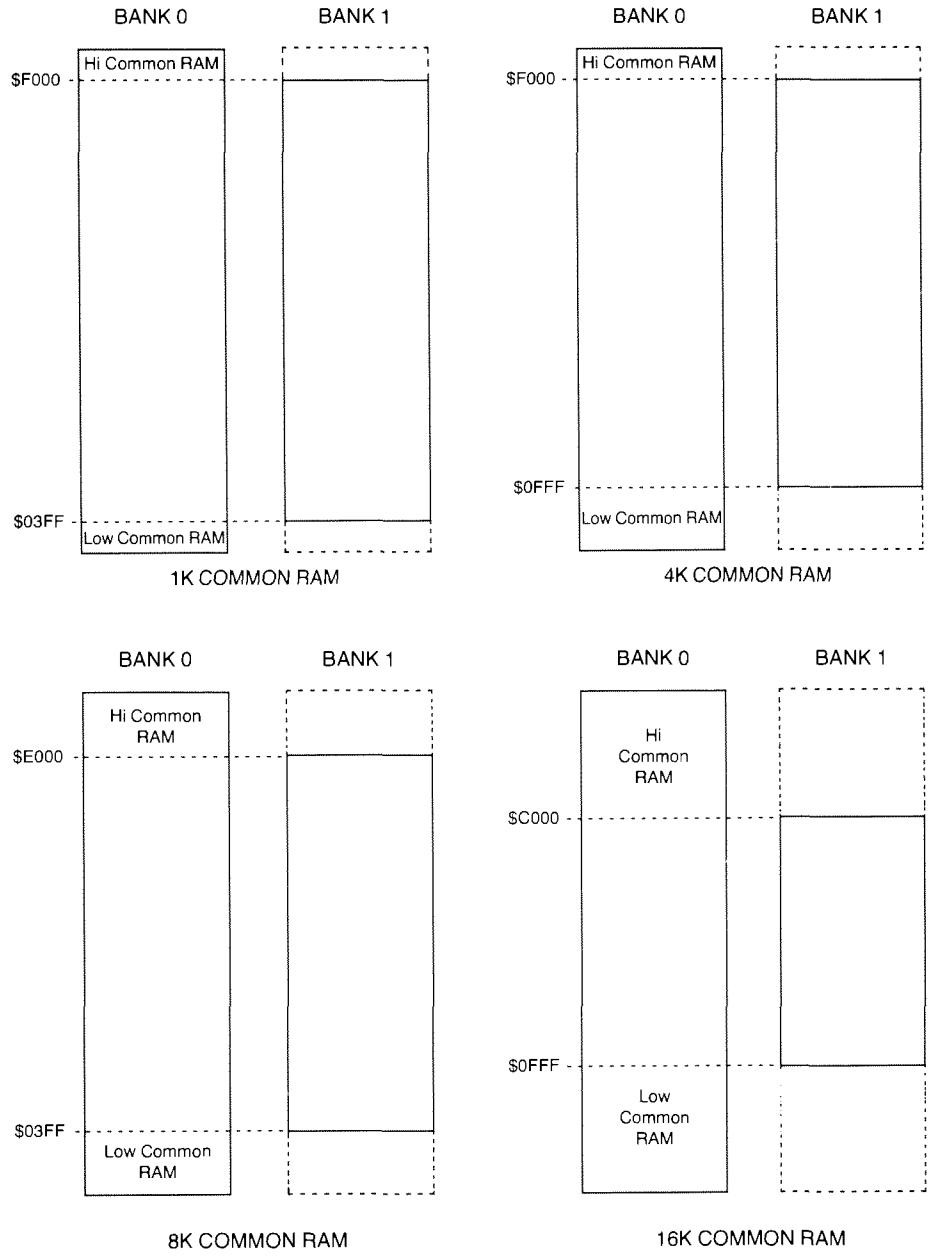



**Figure 13–8. Common RAM**

Bits 2 and 3 specify which portions of the two RAM banks are common, if at all. If both bits are low (0), no RAM sharing occurs. If bit 2 is set high (1) and bit 3 is low (0),

a section of the bottom of RAM bank 0 replaces the corresponding section of RAM bank 1 for all RAM address accesses. If bit 3 is set high (1) and bit 2 is set low (0), a section of the top RAM bank 0 replaces the corresponding section of RAM bank 1 for all RAM address accesses. If both bits 2 and 3 equal 1 (high), RAM is common to both the top and bottom of the RAM banks. Upon power-up or reset, bits 2 and 3 are set to 0, and no RAM is shared between banks. From a hardware standpoint, the 128K MMU selects the common RAM by forcing the CAS0 enable line low and CAS1 enable line high for all common memory accesses.

Bits 4 and 5 have no assigned function. They are reserved for future expansion.

Bits 6 and 7 in the RCR operate as a RAM bank pointer to tell the VIC chip which portion of RAM to use. At the present time, bit 7 is ignored. It too is reserved for future RAM expansion. When bit 6 is low (0) (driving CAS0 low), the VIC chip is told to look in RAM bank 0. When bit 6 is set high (1) (driving CAS1 low), the VIC chip is steered into RAM bank 1. Either setup allows the VIC chip RAM bank to be selected from the microprocessor RAM bank independently.

When the microprocessor speed is increased to 2MHz, the VIC chip is disabled and the 80-column (8563) chip takes over the video processing. The VIC chip is affected by holding the AEC hardware line high. The disabling of the VIC chip is not directly affected by the actions of the MMU.

Figure 13–9 summarizes the RAM Configuration Register activities.

**$D506 →**

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|

**RAM Configuration Register**

| BITS | | | FUNCTIONAL DESCRIPTION |
|---|---|---|---|
| **1** | **0** | **-** | **Determine amount of shared RAM between RAM banks** |
| **0** | **0** | **=** | **1K common RAM** |
| **0** | **1** | **=** | **4K common RAM** |
| **1** | **0** | **=** | **8K common RAM** |
| **1** | **1** | **=** | **16K common RAM** |
| **3** | **2** | **-** | **Determine which portions of RAM are shared** |
| **0** | **0** | **=** | **No common RAM** |
| **0** | **1** | **=** | **Bottom of RAM bank 0 is common** |
| **1** | **0** | **=** | **Top of RAM bank 0 is common** |
| **1** | **1** | **=** | **Both top and bottom of RAM bank 0 are common** |
| **5** | **4** | **-** | **Not implemented; reserved for future expansion** |
| **7** | **6** | **-** | **Selects RAM bank for the VIC chip (Bit 7 is ignored)** |
| **X** | **0** | **=** | **Select Bank 0 for VIC chip** |
| **X** | **1** | **=** | **Select Bank 1 for VIC chip** |

Note: X = not implemented

**Figure 13–9. RAM Configuration Register Summary**

## THE PAGE POINTERS

The Commodore 128 has a feature that allows you to relocate page 0 ($00–$FF) and page 1 ($100–$1FF) of memory. Certain applications may require you to keep page 0 intact while running BASIC, switch out the BASIC ROM, resume processing within the control of your machine language program, and then switch 0 page and the BASIC ROM back in. Instead of transferring 0 page with the machine language monitor, the page pointers make it easy to relocate pages 0 and 1. Four registers within the MMU, called the **page pointers**, allow you to do this. The page pointers are located within the I/O block in the range $D507 through $D50A. The page pointers follow the standard 8502 low-byte/high-byte format. Here are the actual addresses of the high and low bytes of the page pointers:

$D507 Page 0 Pointer (low)
$D508 Page 0 Pointer (high)
$D509 Page 1 Pointer (low)
$D50A Page 1 Pointer (high)

Bit 0 of the high byte page pointers corresponds to the RAM bank number for any page 0 address access. Bit 0 controls the generation of the CAS0 hardware control line if it is low and the CAS1 line if it is high, and bits 1 and 3 are ignored.

To relocate page 0, perform a write operation on the high-byte 0 page pointer. This is stored in the high-byte page pointer location and has no direct result until a write operation is performed on the low-byte 0 page pointer. When this occurs, bits 0 through 7 of the low-byte page pointer correspond to the Translated Address lines TA8 through TA15 for any 0 page address reference, which relocates the 0 page. Any subsequent 0 page address is relayed to the new page zero.

The page 1 pointer works the same way. Both pairs of pointers are initialized to 0 upon power-up, placing pages 0 and 1 in actual page 0 and page 1 locations.

It is important to note that memory addresses 0 and 1 are always available at those absolute address references, regardless of whether pages zero or one are relocated.

The page zero low-byte pointer directly replaces the high-order address of zero page (normally 00). When pages zero and one are directed to locations in RAM memory above page one, the MMU translates the addresses back to the normal locations of pages zero and one, effectively swapping those two pages of memory. This address translation applies only to RAM; ROM and I/O registers are not back-translated. VIC chip addresses are not translated back to their original memory locations, so you must ensure that you do not place page zero or one in the address of the VIC chip. The ROM appearing in these address ranges still overlays the RAM regardless of whether the RAM is zero page, page one or any other page of RAM memory. If you need to use the Kernal, the necessary variables required for the Kernal routine must be placed in memory where the Kernal is in context, Machine Language Monitor bank 15 ($0F) for example.

Bit 0 of both high-byte page pointers (0 and 1) corresponds to the RAM bank number for any address access in page zero or one. The page zero high byte (bit 0) normally overrides the RAM bank set by the configuration register. However, if common RAM (at the bottom of RAM bank 0) is specified, the high-byte pointer for pages zero and one is ignored and pages zero and one appear in common RAM. Otherwise, if common RAM is not allocated, pages zero and one appear where you specify according to the contents of the page pointers. In other words, common RAM takes priority over the page pointers, if common RAM is allocated by the RAM configuration register.

This feature of relocatable page zero and page one provides many benefits to the programmer. This allows machine language programs to create several pages of zero page variables or several different stacks. When you need to access the additional zero page variables or extra 256 bytes of the stack, simply change the pointer to look at the next page. This provides additional speed in your programs since you may use zero page addressing for subsequent zero pages. You can even place zero page in screen memory for extra fast writing to the screen. In addition, it gives you a way to implement deeper levels of subroutines since you have a larger stack area. Remember, though, to leave three bytes on the top of the stack for interrupt requests and servicing.

## THE SYSTEM VERSION REGISTER

The **System Version Register,** located in address $D50B, contains a value that tells the user which version of the MMU is inside the C128 and how large the memory is. Bits 0 through 3 contain the MMU version number. Bits 4 through 7 contain a value (in memory blocks) pertaining to the size of the C128 memory. This allows the programmer to check the version of the C128 and the memory size, and make it compatible with systems that will be expanded in the future. The current version of the C128 contains the value $20, signifying two 64K blocks.

# AUTO STARTING A ROM APPLICATION CARTRIDGE

Many of you may want to place your application program in a cartridge which plugs into the expansion port. In order to automatically start the program as soon as you turn on the computer, you must place a particular coded sequence in the first 6 bytes where the external (or internal) ROM cartridge maps into memory. Here's the auto start sequence in both C128 and C64 modes.

## COMMODORE 128 MODE

Any C64 cartridge is asserted automatically if the system recognizes the /GAME or /EXROM as being pulled low.

If any C128 cartridges are installed into the expansion port:

1. Log the cartridge I.D. into the Physical Address Table (PAT).
2. If the I.D. equals 1, call the cold start vector (which may RTS).

The first 9 bytes of the Commodore 128 auto start sequence are:

| BYTE | |
|---|---|
| **$X000** | **Cold Start Vector** |
| **$X003** | **Warm Start Vector (not used but must be specified)** |
| **$X006** | **Cartridge I.D. (The I.D. = 1 for an auto start card)** |
| **$X007** | **The ASCII Character "C" with the high bit set** |
| **$X008** | **The ASCII Character "B" with the high bit set** |
| **$X009** | **The ASCII Character "M" with the high bit set** |

where X is the hexadecimal digit "8" for $8000 or "C" for $C000.

There are four slots where cartridges (ROM's) may plug in (two internal, two external). They must follow the sequence described above, whether they are internal or external.

## COMMODORE 64 MODE

The first 10 bytes of the Commodore 64 auto start sequence are:

| BYTE | |
|---|---|
| **$X000** | **Cold Start Vector** |
| **$X003** | **Warm Start Vector** |
| **$X006** | **The ASCII Character "C" with the high bit set** |
| **$X007** | **The ASCII Character "B" with the high bit set** |
| **$X008** | **The ASCII Character "M" with the high bit set** |
| **$X009** | **The ASCII Character "8" with the high bit set** |
| **$X00A** | **The ASCII Character "0" with the high bit set** |

where X is the hexadecimal digit "8" for $8000 or "C" for $C000.

The recommended cartridge header for both operational modes is as follows:

```
SEI
JMP START
NOP
NOP
.
.
.
```

This header is recommended so that the interrupt disable status bit is set when control is passed to the software in the cartridge ROM.

# THE COMMODORE 128 SCREEN EDITOR

The Commodore screen editor is among the easiest to use of all screen editors. As soon as you turn on the computer, the screen editor is available to you. You don't have to call any additional text editors. Using the keys for manipulating text, the screen editor gives you more freedom than most other editors.

## C128 EDITOR ESCAPE CODES

To use the following ESCAPE functions, press the **ESCAPE** key and then press the key for the function you want.

| KEY | FUNCTION |
|---|---|
| A | Enable auto-insert mode |
| B | Set bottom right of screen window at cursor position |
| C | Disable auto-insert mode |
| D | Delete current line |
| E | Set cursor to nonflashing mode |
| F | Set cursor to flashing mode |
| G | Enable bell (control-G) |
| H | Disable bell |
| I | Insert line |
| J | Move to start of current line |
| K | Move to end of current line |
| L | Enable scrolling |
| M | Disable scrolling |
| N | Return screen to normal (nonreverse video) state (80-column screen only) |
| O | Cancel insert, quote, underline, flash and reverse modes |
| P | Erase to start of current line |
| Q | Erase to end of current line |
| R | Set screen to reverse video (80-column screen only) |
| S | Change to block cursor (80-column screen only) |
| T | Set top left of screen window at cursor position |
| U | Change to underline cursor (80-column screen only) |
| V | Scroll up |
| W | Scroll down |
| X | Swap 40/80-column display output device |
| Y | Set default tab stops (8 spaces) |
| Z | Clear all tab stops |
| @ | Clear to end of screen |

# C128 EDITOR CONTROL CODES

The following control characters in the CBM ASCII table have been added or changed from those found in the C64. Codes not shown in this table have the same function as found in the C64.

| CHR$ VALUE | KEYBOARD CONTROL | CHARACTER FUNCTION |
|---|---|---|
| 2 | B | **Underline ON (80-column screen only)** |
| 7 | G | **Produces bell tone** |
| 9 | I | **Tab character** |
| 10 | J | **Line feed character** |
| 11 | K | **Disable shift Commodore key (formerly code 9)** |
| 12 | L | **Enable shift Commodore key (formerly code 8)** |
| 15 | O | **Turn ON flash on (80-column screen only)** |
| 24 | X | **Tab set/clear** |
| 27 | [ | **Escape character** |
| 130 | | **Underline OFF (80-column screen only)** |
| 143 | | **Turn flash OFF (80-column screen only)** |

# C128 EDITOR JUMP TABLE

The editor calls listed below are a set of extensions to the standard CBM jump table. They are specifically for the C128 and should not be considered as permanent additions to the standard jump table. They are all true subroutines and terminate with the RTS instruction. As with all Kernal calls, the system configuration (high ROM, RAM-0 and I/O) must be in context at the time of the call.

```
 1.  $C000   CINT      ;initialize editor and screen
 2.  $C003   DISPLY    ;display .A=char, .X=color
 3.  $C006   LP2       ;get a key from irq buffer in .A
 4.  $C009   LOOP5     ;get a chr from screen line in .A
 5.  $C00C   PRINT     ;print character in .A
 6.  $C00F   SCRORG    ;get size of current window
 7.  $C012   SCNKEY    ;scan keyboard subroutine
 8.  $C015   REPEAT    ;repeat key logic and CKIT2
 9.  $C018   PLOT      ;read or set cursor position
10.  $C018   CURSOR    ;move 8563 cursor subroutine
11.  $C01E   ESCAPE    ;execute escape function
12.  $C021   KEYSET    ;redefine a programmable key
13.  $C024   IRQ       ;irq entry
14.  $C027   INIT80    ;initialize 80-column char set
15.  $C02A   SWAPPER   ;40/80 mode change
16.  $C02D   WINDOW    ;set UL or BR of window
17.  $C033   LDTB2     ;screen lines low byte table
18.  $C04C   LDTB1     ;screen lines high byte table
19.  $334    CONTRL    ;print CTRL indirect
20.  $336    SHIFTD    ;print SHFT indirect
21.  $338    ESCAPE    ;print ESC indirect
22.  $33A    KEYVEC    ;keyscan logic indirect
23.  $33C    KEYCHK    ;keyscan store indirect
24.  $33E    DECODE    ;keyboard decode table vectors
```

Entries 17, 18, and 24 are table pointers, and are not callable routines. Entries 19–23 are considered indirect vectors, not true entry points.

This chapter has presented the Commodore 128 operating system. Chapter 16 provides information on CP/M on the Commodore 128 and Commodore 64 memory maps.

# 14

# CP/M 3.0 ON THE COMMODORE 128

CP/M® is a microprocessor operating system produced by Digital Research, Inc. (DRI). The version of CP/M used on the Commodore 128 is CP/M Plus™ Version 3.0. In this chapter, CP/M is generally referred to as CP/M 3.0, or simply CP/M.

This chapter summarizes the non-C128-dependent aspects of CP/M on the Commodore 128. For detailed information on C128-dependent CP/M 3.0, see Appendix K of this guide. For detailed information on non-C128-dependent CP/M 3.0, see the Commodore-produced volume that includes DRI's *CP/M Plus User's Guide, Programmer's Guide,* and *System Guide.*

## REQUIREMENTS FOR A CP/M 3.0 SYSTEM

The general requirements for CP/M 3.0 are:

- A computer containing a Z80 microprocessor.
- A console consisting of a keyboard and a display screen.
- At least one floppy disk drive.
- CP/M system software on disk.

CP/M 3.0 on the Commodore 128 Personal Computer normally consists of the following elements:

- A built-in Z80 microprocessor.
- A console consisting of the full Commodore 128 keyboard and an 80-column monitor.
- The Commodore 1571 fast disk drive.
- The CP/M system disk, which includes the CP/M 3.0 system, an extensive HELP utility program and a number of other utility programs.

> **NOTE:** CP/M can also be used with a 40-column monitor. To view all 80 columns of display, you must scroll the screen horizontally by pressing the **CONTROL** key and the appropriate **CURSOR** key (left or right).

CP/M 3.0 can also be used with the 1541 disk drive. In this case, only single-sided GCR disks can be used, and the speed of operation will be approximately one-tenth the speed achieved using the 1571 disk drive. (See the discussion of disk formats later in this chapter for more details.)

# COMMODORE ENHANCEMENTS TO CP/M 3.0

Commodore has added a number of enhancements to CP/M 3.0. These enhancements tailor the capabilities of the Commodore 128 to those of CP/M 3.0. They include such things as a selectively displayed disk status line, a virtual disk drive, local/remote handling of keyboard codes, programmable function keys (strings) and a number of additional functions/characters that are assigned to various keys. These enhancements are described at appropriate points in this chapter.

# CP/M FILES

There are two types of CP/M files:

- **PROGRAM** or **COMMAND** files, consisting of a series of instructions that the computer follows to achieve a specified result.
- **DATA** files, consisting usually of a collection of related information (e.g., a list of customer names and addresses; inventory records; accounting records; the text of a document).

## FILE SPECIFICATION

A CP/M file is identified by a file specification that can consist of up to four individual elements, as follows:

- **Drive Specifier** (optional), consisting of a single letter, followed by a colon. Each disk drive is assigned a letter, in the range A through E. (E denotes a virtual drive, as explained later in this chapter.)
- **Filename** (mandatory), which can be from one to eight characters long. (Note that this is the only mandatory element of the file specification.)
- **Filetype** (optional), consisting of one to three characters. It must be separated from the filename by a period.
- **Password** (optional), which can be from one to eight characters. Must be separated from the filetype (or filename, if no filetype is included) by a semicolon.

**EXAMPLE:**

The following file specification contains all four possible elements, all separated by the appropriate symbols:

A:DOCUMENT.LAW;FIREBIRD

## USER NUMBER

CP/M 3.0 further identifies all files by assigning each one a user number, which can range from 0 to 15. CP/M 3.0 assigns the user number to a file when the file is created. User numbers allow you to separate your files into sixteen file groups.

Any user number other than 0 must precede the drive specifier. User 0, which is the default user number, is not displayed in the prompt.

If a file resides in user 0 and is marked with a system file attribute, that file can be accessed from any user number. Otherwise, a command can access only those files that have the current user number.

## CREATING A FILE

There are several ways to create a CP/M file, including:

- Using the CP/M text editor (ED).
- Using the PIP command to copy and rename a file.
- Using a program such as MAC (a CP/M machine language assembler program), which creates output files as it processes input files.

## USING WILDCARD CHARACTERS TO ACCESS MORE THAN ONE FILE

A *wildcard* is a character that can be used in a filename or filetype in place of some other characters. CP/M 3.0 uses the question mark (?) and asterisk (*) as wildcards. A ? stands for any character that may be encountered in that position. An * tells CP/M to fill the filename with question mark characters as indicated. A file specification containing a wildcard can refer to more than one file and is therefore called an ambiguous file specification.

## RESERVED CHARACTERS

The characters in Table 14–1 are reserved characters in CP/M 3.0. Use only as indicated.

| CHARACTER | MEANING |
|---|---|
| < $ , ! \| > [ ]<br>TAB, SPACE,<br>CARRIAGE RETURN | File specification delimiters |
| : | Drive delimiter in file specification |
| . | Filetype delimiter in file specification |
| ; | Password delimiter in file specification |
| ; | Comment delimiter at the beginning of a command line |
| * ? | Wildcard characters in an ambiguous file specification |
| < > & ! \| \ + – | Option list delimiters |
| [ ] | Option list delimiters for global and local options |
| ( ) | Delimiters for multiple modifiers inside square brackets for options that have modifiers |
| / $ | Option delimiters in a command line |

**Table 14–1. CP/M 3.0 Reserved Characters**

## RESERVED FILETYPES

The filetypes defined in Table 14–2 are reserved for system use.

| FILETYPE | MEANING |
|---|---|
| ASM | Assembler source file |
| BAS | BASIC source program |
| COM | Z80 or equivalent machine language program |
| HEX | Output file from MAC (used by HEXCOM) |
| HLP | HELP message file |
| $$$ | Temporary file |
| PRN | Print file from MAC or RMAC |
| REL | Output file from RMAC (used by LINK) |
| SUB | List of commands to be executed by SUBMIT |
| SYM | Symbol file from MAC, RMAC or LINK |
| SYS | System file |
| RSX | Resident System Extension (a file automatically loaded by a command file when needed) |

**Table 14–2. CP/M 3.0 Reserved Filetypes**

# CP/M COMMANDS

There are two types of commands in CP/M 3.0:

- Built-in commands, which identify programs in memory.
- Transient utility commands, which identify program files.

CP/M 3.0 has six built-in commands and over twenty transient utility commands. Utilities can be added by purchasing CP/M 3.0–compatible application programs. In addition, experienced programmers can write utilities that operate with CP/M 3.0.

## BUILT-IN COMMANDS

Built-in commands are entered in the computer's memory when CP/M 3.0 is loaded, and are always available for use, regardless of which disk is in which drive. Table 14–3 lists the Commodore 128 CP/M 3.0 built-in commands.

Some built-in commands have options that require support from a related transient utility. The related transient utility command has the same name as the built-in command and has a filetype of COM.

| COMMAND | FUNCTION |
|---|---|
| DIR | Displays filenames of all files in the directory except those marked with the SYS attribute. |
| DIRSYS | Displays filenames of files marked with the SYS (system) attribute in the directory. |
| ERASE | Erases a filename from the disk directory and releases the storage space occupied by the file. |
| RENAME | Renames a disk file. |
| TYPE | Displays contents of an ASCII (TEXT) file at your screen. |
| USER | Changes to a different user number. |

**Table 14–3. Built-In Commands**

## TRANSIENT UTILITY COMMANDS

The CP/M 3.0 transient utility commands are listed in Table 14–4. When a command keyword identifying a transient utility is entered, CP/M 3.0 loads the program file from the disk and passes that file any filenames, data or parameters specified in the command tail.

## USING CONTROL CHARACTERS FOR LINE EDITING

Table 14–5 lists the line-editing control characters for CP/M 3.0 on the Commodore 128.

## HOW TO MAKE COPIES OF CP/M 3.0 DISKS AND FILES

> **NOTE:** The Digital Research Inc. **COPYSYS** command, used in many CP/M systems in formatting a disk, is not implemented on the Commodore 128 computer. Instead, the Commodore 128 uses a special **FORMAT** command.

To make backups of the CP/M system disks, use the FORMAT and PIP utility programs. FORMAT formats the disk as either a C128 single-sided or C128 double-sided diskette.

| NAME | FUNCTION |
|---|---|
| DATE | Sets or displays the date and time. |
| DEVICE | Assigns logical CP/M devices to one or more physical devices, changes device driver protocol and baud rates, or sets console screen size. |
| DIR | Displays directory with files and their characteristics. |
| DUMP | Displays a file in ASCII and hexadecimal format. |
| ED | Creates and alters ASCII files. |
| ERASE | Used for wildcard erase. |
| GENCOM | Creates a special COM file with attached RSX file. |
| GET | Temporarily gets console input from a disk file rather than the keyboard. |
| FORMAT | Initializes disk in GCR format for CP/M use. |
| HELP | Displays information on how to use CP/M 3.0 commands. |
| INITDIR | Initializes a disk directory to allow time and date stamping. |
| KEYFIG | Allows redefinition of keys. |
| PATCH | Displays or installs patches to CP/M system. |
| PIP | Copies files and combines files. |
| PUT | Temporarily directs printer or console output to a disk file. |
| RENAME | Changes the name of a file, or a group of files, using wildcard characters. |
| SAVE | Saves a program in memory to disk. |
| SET | Sets file options including disk labels, file attributes, type of time and date stamping and password protection. |
| SETDEF | Sets system options including the drive search chain. |
| SHOW | Displays disk and drive statistics. |
| SUBMIT | Automatically executes multiple commands. |
| TYPE | Displays contents of text file (or group of files, if wildcard characters are used) on screen (and printer if desired). |

**Table 14–4. Transient Utility Commands**

| CHARACTER | MEANING |
|---|---|
| CTRL-A or SHIFT-LEFT CURSOR | Moves the cursor one character to the left. |
| CTRL-B | Moves the cursor to the beginning of the command line without having any effect on the contents of the line. If the cursor is at the beginning, CTRL-B moves it to the end of the line. |
| CTRL-E | Forces a physical carriage return but does not send the command line to CP/M 3.0. Moves the cursor to the beginning of the next line without erasing the previous input. |
| CTRL-F or RIGHT CURSOR | Moves the cursor one character to the right. |
| CTRL-G | Deletes the character at current cursor position. The cursor does not move. Characters to the right of the cursor shift left one place. |
| CTRL-H | Deletes the character to the left of the cursor and moves the cursor left one character position. Characters to the right of the cursor shift left one place. |
| CTRL-I | Moves the cursor to the next tab stop. Tab stops are automatically set at each eighth column. Has the same effect as pressing the **TAB** key. |
| CTRL-J | Sends the command line to CP/M 3.0 and returns the cursor to the beginning of a new line. Has the same effect as a **RETURN** or a CTRL-M keystroke. |
| CTRL-K | Deletes to the end of the line from the cursor. |
| CTRL-M | Sends the command line to CP/M 3.0 and returns the cursor to the beginning of a new line. Has the same effect as a **RETURN** or a CTRL-J keystroke. |
| CTRL-R | Retypes the command line. Places a # character at the current cursor location, moves the cursor to the next line, and retypes any partial command you typed so far. |
| CTRL-U | Discards all the characters in the command line, places a # character at the current cursor position, and moves the cursor to the next line. However, you can use a CTRL-W to recall any characters that were to the left of the cursor when you pressed CTRL-U. |
| CTRL-W or ↑ CRSR ↓ | Recalls and displays previously entered command line both at the operating system level and within executing programs, if the CTRL-W is the first character entered after the prompt. CTRL-J, CTRL-M, CTRL-U and RETURN define the command line you can recall. If the command line contains characters, CTRL-W moves the cursor to the end of the command line. If you press **RETURN** , CP/M 3.0 executes the recalled command. |
| CTRL-X | Discards all the characters left of the cursor and moves the cursor to the beginning of the current line. CTRL-X saves any characters to the right of the cursor. |

**Table 14–5. CP/M 3.0 Line-Editing Control Characters**

## MAKING COPIES WITH A SINGLE DISK DRIVE

You can copy the contents of a disk with a single Commodore disk drive (1541 or 1571). For example, use the following sequence of commands to create a bootable CP/M system disk. First type:

```
A>FORMAT
```

and follow the instructions given on the screen. When the copy disk is formatted, type:

```
A>PIP E:=A:CPM+.SYS
```

When the CPM + SYS file is copied, you type:

```
A>PIP E:=A:CCP.COM
```

To copy everything on a disk, use the following command sequence:

```
A>FORMAT
A>PIP E:=A:*.*
```

The system prompts the user to change disks as required.

Use drive A as the source drive and drive E as the destination drive. Drive E is referred to as a virtual drive; that is, it does not exist as an actual piece of hardware—it is strictly a logical drive.

## MAKING COPIES WITH TWO DISK DRIVES

You can back up disks in CP/M by using two drives: drive A and drive B. The drives can be named with other letters from A through D. To make a copy of your CP/M 3.0 system disk, first use the FORMAT utility to copy the operating system loader. Make sure that the CP/M system disk is in drive A, the default drive, and the blank disk is in drive B. Then enter the following command at the system prompt:

```
A>PIP B:=A:CPM+.SYS
```

When you have copied the CPM+.SYS file, use the PIP command to copy the CCP.COM file. This provides a copy of the operating system only. To copy all of the files from the system disk, enter the following PIP command:

```
A>PIP B:=A:*.*
```

This PIP command copies all the files on the disk in drive A to the disk in drive B. PIP displays the message **COPYING,** followed by each filename as the copy operation proceeds. When PIP finishes copying, the system prompt (A>) is displayed.

# GENERAL CP/M 3.0 SYSTEM LAYOUT

The Commodore 128 computer is a two-processor system, with the 8502 as the primary processor and the Z80 as the secondary processor. The 8502 has the same instruction set as the 6502. The Z80's primary function is to run CP/M 3.0. This section describes the general requirements and methods for implementing CP/M 3.0 on the C128.

When CP/M is running, the normal functions of the C128 are not supported (CP/M and BASIC cannot run at the same time). Also, CP/M does not directly support all the display modes of the VIC chip. (An application could be written to run under CP/M that could use additional graphics capabilities, but the application would have to keep track of all the details, such as memory maps).

## CP/M 3.0 OPERATING SYSTEM COMPONENTS

The CP/M 3.0 operating system consists of the following three modules: the Console Command Processor (CCP), the BASIC Disk Operating System (BDOS), and the Basic Input Output System (BIOS).

The CCP is a program that provides the basic user interface to the facilities of the operating system. The CCP supplies the six built-in commands: **DIR, DIRS, ERASE, RENAME, TYPE,** and **USER.** The CCP executes in the Transient Program Area (TPA), the region of memory where all application programs execute. The CCP contains the Program Loader Module, which loads transient (applications) programs from disk into the TPA for execution. On the Commodore 128, a 58K to 59K TPA area is provided for CP/M.

The BDOS is the logical nucleus and file system of CP/M 3.0. The BDOS provides the interface between the application program and the physical input/output routines of the BIOS.

The BIOS is a hardware-dependent module that interfaces the BDOS to a particular hardware environment. The BIOS performs all physical I/O in the system. The BIOS consists of a number of routines that are configured to support the specific hardware of the target computer system (in this case, the Commodore 128).

The BDOS and the BIOS modules cooperate to provide the CCP and other transient programs with hardware-independent access to CP/M 3.0 facilities. Because the BIOS can be configured for different hardware environments and the BDOS remains constant, you can transfer programs that run under CP/M 3.0 to systems with different hardware configurations.

## COMMUNICATION BETWEEN MODULES

The BIOS loads the CCP into the TPA at system cold and warm starts. The CCP moves the Program Loader Module to the top of the TPA and uses the Program Loader Module to load transient programs.

The BDOS contains a set of functions that the CCP and applications programs call to perform disk and character input and output operations.

The BIOS contains a Jump Table with a set of thirty-three entry points that the BDOS calls to perform hardware-dependent primitive functions, such as peripheral device I/O. For example, **CONIN** is an entry point of the BIOS called by the BDOS to read the next console input character.

Similarities exist between the BDOS functions and BIOS functions, particularly for simple device I/O. For example, when a transient program makes a console output function call to the BDOS, the BDOS makes a console output call to the BIOS. In the case of disk I/O, however, this relationship is more complex. The BDOS file may make many BIOS function calls to perform a single BDOS file I/O function. BDOS disk I/O is in terms of 128-byte logical records. BIOS disk I/O is in terms of physical sectors and tracks.

The System Control Block (SCB) is a 100-byte (decimal) CP/M 3.0 data structure that resides in the BDOS system component. The BDOS and the BIOS communicate through fields in the SCB. The SCB contains BDOS flags and data, CCP flags and data, and other system information, such as console characteristics and the current date and time. You can access some of the System Control Block fields from the BIOS.

Note that the SCB contains critical system parameters that reflect the current state of the operating system. If a program modifies these parameters, the operating system can crash. See Section 3 of the DRI *CP/M Plus System Guide* and the description of BDOS Function 49 in the DRI *CP/M Plus Programmer's Guide* for more information on the System Control Block.

Page zero is a region of memory that acts as an interface between transient programs and the operating system. Page zero contains critical system parameters, including the entry to the BDOS and entry to the BIOS Warm BOOT routine. At system start-up, the BIOS initializes these two entry points in page zero. All linkage between transient programs and the BDOS is restricted to the indirect linkage through page zero

# CP/M 3.0 BIOS OVERVIEW

This section describes the organization of the CP/M 3.0 BIOS and the BIOS jump vector. It provides an overview of the System Control Block, followed by a discussion of system initialization procedures, character I/O, clock support, disk I/O, and memory selects and moves.

## ORGANIZATION OF THE BIOS

The BIOS is the CP/M 3.0 module that contains all hardware-dependent input and output routines. To configure CP/M 3.0 for a particular hardware environment, the sample BIOS supplied with the *CP/M Plus System Guide* must be adapted to the specific hardware of the target system (the Commodore 128).

The BIOS essentially is a set of routines that performs system initialization,

character-oriented I/O to the console and printer devices, and physical sector I/O to disk devices. The BIOS also contains routines that manage block moves and memory selects for bank-switched memory. The BIOS supplies tables that define the layout of the disk devices and allocate buffer space that the BDOS uses to perform record blocking and deblocking. The BIOS can maintain the system time and date in the System Control Block.

Table 14–6 describes the entry points into the BIOS from the Cold Start Loader and the BDOS. Entry to the BIOS is through a set of jump vectors. The jump table is a set of thirty-three jump instructions that pass program control to the individual BIOS subroutines.

| NO. | INSTRUCTION | DESCRIPTION |
|---|---|---|
| 0 | JMP BOOT | Perform cold start initialization. |
| 1 | JMP WBOOT | Perform warm start initialization. |
| 2 | JMP CONST | Check for console input character ready. |
| 3 | JMP CONIN | Read console character in. |
| 4 | JMP CONOUT | Write console character out. |
| 5 | JMP LIST | Write list character out. |
| 6 | JMP AUXOUT | Write auxiliary output character. |
| 7 | JMP AUXIN | Read auxiliary input character. |
| 8 | JMP HOME | Move to Track 00 on selected disk. |
| 9 | JMP SELDSK | Select disk drive. |
| 10 | JMP SETTRK | Set track number. |
| 11 | JMP SETSEC | Set sector number. |
| 12 | JMP SETDMA | Set DMA address. |
| 13 | JMP READ | Read specified sector. |
| 14 | JMP WRITE | Write specified sector. |
| 15 | JMP LISTST | Return list status. |
| 16 | JMP SECTRN | Translate logical to physical sector. |
| 17 | JMP CONOST | Return output status of console. |
| 18 | JMP AUXIST | Return input status of aux. port. |
| 19 | JMP AUXOST | Return output status of aux. port. |
| 20 | JMP DEVTBL | Return address of char. I/O table. |
| 21 | JMP DEVINI | Initialize char. I/O devices. |
| 22 | JMP DRVTBL | Return address of disk drive table. |
| 23 | JMP MULTIO | Set number of logically consecutive sectors to be read or written. |
| 24 | JMP FLUSH | Force physical buffer flushing for user-supported deblocking. |
| 25 | JMP MOVE | Memory to memory move. |
| 26 | JMP TIME | Time set/get signal. |
| 27 | JMP SELMEM | Select bank of memory. |
| 28 | JMP SETBNK | Specify banks for inter-bank MOVE. |
| 29 | JMP XMOVE | Set banks for an inter-bank MOVE. |
| 30 | JMP USERF | Commodore CP/M system functions. |
| 31 | JMP RESERV1 | Reserved for future use. |
| 32 | JMP RESERV2 | Reserved for future use. |

**Table 14–6. CP/M 3.0 BIOS Jump Vectors**

All the entry points in the BIOS jump vector are included in the C128 CP/M 3.0 BIOS.

Each jump address in Table 14-6 corresponds to a particular subroutine that performs a specific system operation. Note that two entry points are reserved for future versions of CP/M, and one entry point is provided for the Commodore system functions.

Table 14-7 shows the five categories of system operations and the function calls that accomplish these operations.

| OPERATION | FUNCTION |
|---|---|
| **System Initialization** | **BOOT, WBOOT, DEVTBL, DEVINI, DRVTBL** |
| **Character I/O** | **CONST, CONIN, CONOUT, LIST, AUXOUT, AUXIN, LISTST, CONOST, AUXIST, AUXOST** |
| **Disk I/O** | **HOME, SELDSK, SETTRK, SETSEC, SETDMA, READ, WRITE, SECTRN, MULTIO, FLUSH** |
| **Memory Selects and Moves** | **MOVE, SELMEM, SETBNK, XMOVE** |
| **Clock Support** | **TIME** |

**Table 14–7. CP/M 3 BIOS Functions**

Appendix K illustrates how to call a Commodore CP/M system function in Z80 machine language.

# SYSTEM MEMORY ORGANIZATION

Figure 14–1 shows the general memory organization of CP/M 3.0.

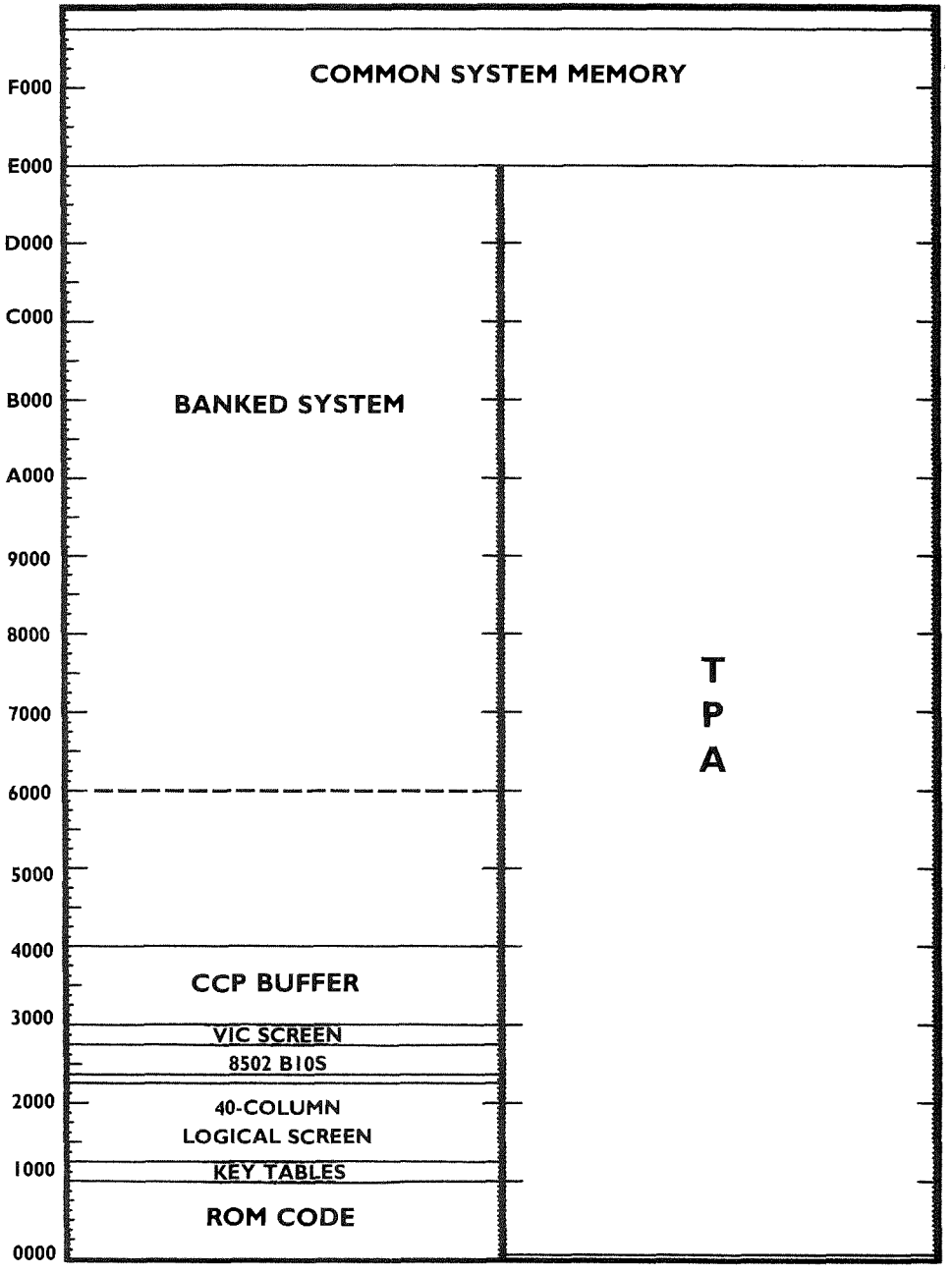



Figure 14–1. CP/M 3.0 General Memory Organization

The memory map is limited to 64K at any one point in time. However, the RAM bank can be selected and then different ROM areas can overlay the RAM (with bleed-through on write operations). The actual memory map is controlled by the MMU. The MMU can be accessed in the I/O area (only when I/O is enabled in Z80 space) or through the Load Configuration Register located at FF00 through FF04.

If the Load Configuration Registers are read, then the current value is read. A write to FF00 changes the configuration after completing the current instruction. A write to FF01 to FF04 updates the current configuration to the value stored in the Preconfiguration Registers (the data written is not used). The MMU page pointers have both a low (page) and a high (page) pointer. The high is written first and latched in the MMU; the high value is updated from the latch when the low byte is written. The MMU Control Registers are listed in Chapter 13.

# DISK ORGANIZATION

CP/M 3.0 supports a number of different disk formats, including three Commodore formats and a number of MFM formats. (MFM is the industry standard format.) The first Commodore format is single-sided Commodore GCR, which is compatible with the CP/M 2.2 that runs on the Commodore 64. With this format, the File Control Block (FCB) is set up as 32 tracks of 17 sectors each and a track offset of 2. The BIOS routine adds a 1 to tracks greater than 18 (this is the C64 directory track).

The second format, known as the C128 CP/M Plus disk format, is new and is also single-sided. This format, which is also a GCR format, takes advantage of the full disk capacity by setting up the FCB with 638 tracks of 1 sector each and a track offset of 0. This has the effect of having CP/M set the track to the block number relative to the beginning of the disk, with the sector always set to 0. The following algorithm is used to convert the requested TRACK to a real track and sector number.

| REQUESTED TRACK | ACTUAL TRACK |
|---|---|
| **000 > = TRACK > 355** | **((TRACK + 2)/21) + 1** |
| **355 > = TRACK > 487** | **((TRACK – 354)/19) + 18** |
| **487 > = TRACK > 595** | **((TRACK – 487)/18) + 25** |
| **595 > = TRACK > 680** | **((TRACK – 595)/17) + 31** |
| **680 > = TRACK > 1360** | **SET SIDE 2 TRACK = TRACK – 680** |

The effective sector is then translated to provide a skew that speeds up operations. The skew is used only with the new larger format. A different skew table is used for each region of the disk.

The third Commodore format is a GCR double-sided format. The disk is treated as 1276 sectors of data with a track offset of 0. Side 1 is used first; then side 2 is used.

> **NOTE:** This is not the usual way to handle a two-sided disk; however, allocating the disk in this manner, the user with a 1541 may still be able to read data written at the start of a two-sided disk.

The third Commodore format and all MFM formats require that the user have the new 1571 disk drive. This disk drive supports both single- and double-sided diskettes and both the Commodore GCR and industry standard MFM data coding formats.

The following table summarizes 1541/1571 disk drive capabilities with regard to the various disk formats.

| DISK FORMAT | 1541 DRIVE | 1571 DRIVE |
|---|---|---|
| **C64 GCR single-sided** | √ | √ |
| **C128 GCR single-sided** | √ | √ |
| **C128 GCR double-sided** | | √ |
| **MFM format** | | √ |

## C64 CP/M DISK FORMAT (SINGLE-SIDED)

This format, shown in Figure 14–2, is provided to allow the user to read/write files created using the C64 CP/M 2.2 cartridge. (However, do **not** use the CP/M 2.2 cartridge with the C128.) Notice the unused space in this format.

| | SECTOR | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 0 | B | B | B | B | B | B | B | B | B | B | B | B | B | B | B | B | B | x | x | x | x |
| 1 | B | B | B | B | B | B | B | B | B | B | B | B | B | B | B | B | B | x | x | x | x |
| 2 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | x | x | x | x |
| 3 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | x | x | x | x |
| 4 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | x | x | x | x |
| 5 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | x | x | x | x |
| 6 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | x | x | x | x |
| 7 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | x | x | x | x |
| 8 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | x | x | x | x |
| 9 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | x | x | x | x |
| 10 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | x | x | x | x |
| 11 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | x | x | x | x |
| 12 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | x | x | x | x |
| 13 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | x | x | x | x |
| 14 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | x | x | x | x |
| 15 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | x | x | x | x |
| 16 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | x | x | x | x |
| 17 | D | D | D | D | D | D | D | D | D | D | D | D | D | D | D | D | D | D | D | | |
| 18 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | x | x | | |
| 19 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | x | x | | |
| 20 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | x | x | | |
| 21 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | x | x | | |
| 22 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | x | x | | |
| 23 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | x | x | | |
| 24 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | x | | | |
| 25 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | x | | | |
| 26 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | x | | | |
| 27 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | x | | | |
| 28 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | x | | | |
| 29 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | x | | | |
| 30 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | | | | |
| 31 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | | | | |
| 32 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | | | | |
| 33 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | | | | |
| 34 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | | | | |

. Used by CP/M.
B Boot Sector (System).
D Directory Sector (Disk DOS).
x Not used by CP/M.

**Figure 14–2. C64 C/PM Disk Format**

# C128 CP/M DISK FORMATS (SINGLE- AND DOUBLE-SIDED)

Figure 14–3 shows the C128 CP/M Plus format for a single-sided disk.

SECTOR

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | B | . | . | . | . | x | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| 1 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| 2 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| 3 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| 4 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| 5 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| 6 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| 7 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| 8 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| 9 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| 10 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| 11 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| 12 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| 13 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| 14 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| 15 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| 16 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| 17 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | | |
| 18 | D | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | | |
| 19 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | | |
| 20 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | | |
| 21 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | | |
| 22 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | | |
| 23 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | | |
| 24 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | | | |
| 25 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | | | |
| 26 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | | | |
| 27 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | | | |
| 28 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | | | |
| 29 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | | | |
| 30 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | | | | |
| 31 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | | | | |
| 32 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | | | | |
| 33 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | | | | |
| 34 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | | | | |

. Used by CP/M, reg. = region (1–4).
B Boot Sector (System).
D Directory Sector (Disk DOS).
x Not used by CP/M.

SKEW TABLE

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 00 | 17 | 13 | 09 | 05 | 01 | 18 | 14 | 10 | 06 | 02 | 19 | 15 | 11 | 07 | 03 | 20 | 16 | 12 | 08 | 04 |
| 2 | 00 | 04 | 08 | 12 | 16 | 01 | 05 | 09 | 13 | 17 | 02 | 06 | 10 | 14 | 18 | 03 | 07 | 11 | 15 | | |
| 3 | 00 | 11 | 04 | 15 | 08 | 01 | 12 | 05 | 16 | 09 | 02 | 13 | 06 | 17 | 10 | 03 | 14 | 07 | | | |
| 4 | 00 | 07 | 14 | 04 | 11 | 01 | 08 | 15 | 05 | 12 | 02 | 09 | 16 | 06 | 13 | 03 | 10 | | | | |

**Figure 14–3. C128 CP/M Plus Disk Format**

**NOTE:** This format is duplicated on the second side of a two-sided disk (with the exception of B, which is unused and therefore becomes X).

## MFM DISK FORMATS

A number of MFM disk formats are built into CP/M on the C128. These formats, which can be selected at the time a program is to be run, include Osborne, Kaypro, Epson and IBM CP/M-86. The IBM CP/M-86 capability is provided so that data can be transferred between machines. However CP/M-86 programs cannot be run on the C128, since on the C128, CP/M Plus runs on a Z80, not an 8088.

When used with the 1571 Disk Drive, the Commodore 128 supports a variety of double-density MFM disk formats (for reading and/or writing), including:

| | |
|---|---|
| Epson QX10 | (512 byte sectors, double-sided, 10 sectors per track) |
| IBM-8 SS (CP/M 86) | (512 byte sectors, single-sided, 8 sectors per track) |
| IBM-8 DS (CP/M 86) | (512 byte sectors, double-sided, 8 sectors per track) |
| KayPro II | (512 byte sectors, single-sided, 10 sectors per track) |
| KayPro IV | (512 byte sectors, double-sided, 10 sectors per track) |
| Osborne DD | (1024 byte sectors, single-sided, 5 sectors per track) |

When you insert one of these disks into the disk drive and try to access it, the system senses the type of disk with respect to the number of bytes per sector and the number of sectors per track. If the disk format is not unique, a box is displayed near the bottom-left corner of the screen, showing which disk type you are accessing. The system requires you to select the specific disk type by scrolling through the choices given in this window.

**NOTE:** The choices are given one at a time; scroll through using the right and left arrow keys. Press **RETURN** when the disk type that you know is in the disk drive is displayed. Press **CONTROL RETURN** to lock this disk format so that you will not need to select the disk type each time you access the disk drive.

# KEYBOARD SCANNING

The keyboard scan routine that is called to get a keyboard character returns the key code of the pressed key, or a code indicating that no key is currently being pressed. The keyboard scan code is also responsible for handling programmable keys, programmable function keys, setting character and background colors, selecting MFM disk formats and selecting current screen emulation type.

Any key on the keyboard can be defined to generate a code or function except the following keys:

**LEFT SHIFT**
**RIGHT SHIFT**
**SHIFT LOCK**
**C=**
**CONTROL**
**RESTORE (8502 NMI)**
**40/80 DISPLAY**
**CAPS LOCK KEY**

The keyboard recognizes the following special functions:

Cursor left key–Used to define a key

Cursor right key–Used to define a string
(points to function keys)

ALT key–Used as toggle key filter

To indicate these functions, hold down the **CONTROL** key and the **RIGHT SHIFT** key and simultaneously press the desired function key.

## DEFINING A KEY

The **KEYFIG** utility program allows the user to define the code that a key can produce. Each key has four modes of use for this function:

- Normal
- Alpha shift
- Shift
- Control

The alpha shift mode is toggled on/off by pressing the Commodore (C=) key. When this mode is turned on, a small white box appears on the bottom of the screen. The first key that is pressed thereafter is the key to be defined. The current hex value assigned to this key is displayed, and the user can then type the new hex code for the key, or abort by typing a non-hex key. The following is a definition of the codes that can be assigned to a key. See KEYFIG HELP for more information.

| CODE | FUNCTION |
|---|---|
| 00h | Null (same as not pressing a key) |
| 01h to 7Fh | Normal ASCII codes |
| 80h to 9Fh | String assigned |
| A0h to AFh | 80-column character color |
| B0h to BFh | 80-column background color |
| C0h to CFh | 40-column character color |
| D0h to DFh | 40-column background color |
| E0h to EFh | 40-column border color |
| F0h | Toggle disk status on/off |
| F1h | System Pause |
| F2h | (Undefined) |
| F3h | 40-column screen window right |
| F4h | 40-column screen window left |
| F5h to FFh | (Undefined) |

## DEFINING A STRING

The **DEFINE STRING** function allows the user to assign more than one key code to a single key. Any key that is typed in this mode is placed in the string. The user can see the results of typing in a long box at the bottom of the screen. Note, however, that some keys may not display what they are.

To allow the user control of entering data, five special edit functions are provided. To access any of these functions, you first press the **CONTROL** and **RIGHT SHIFT** keys. (This allows the user to enter any key into the buffer.)

The functions assigned to the five string edit keys are as follows:

| KEY | FUNCTION |
|---|---|
| **RETURN** | End string definition. |
| + symbol* | Insert space in string. |
| – symbol* | Delete cursor character. |
| Right Cursor | Move cursor to right. |
| Left Cursor | Move cursor to left. |

*On main keyboard only.

## ALT MODE

This function is a toggle (on/off) and is provided to allow the user to send 8-bit codes to an application without the keyboard driver "eating" the code from 80h to FFh.

# UPDATING THE 40/80 COLUMN DISPLAY

As noted elsewhere in this book, there are two different display systems within the C128. The first, which is controlled by the VIC chip, produces a 25-line by 40-column display, has many graphics modes of operation, and can be used with a standard color (or black-and-white) television or color monitor. (See Chapters 8 and 9 for details.) The only VIC-controlled display mode used by CP/M is standard character mode, with each character and screen background having up to sixteen colors.

The second display system available in C128 CP/M is controlled by the 8563 display controller. The display format of this controller is 25 lines by 80 columns, with character color attributes. The VIC chip is a memory-mapped display, and the 8563 is I/O-controlled. The two display subsystems are treated as two separate displays. CP/M 3.0 can assign one or both to the console output device.

Both displays are controlled by a common terminal emulation package, a Lear Siegler ADM-31 (ADM-3A is a subset of this) driver. The terminal driver is divided into two parts: terminal emulation and terminal functions. Terminal emulation is handled by the Z80 BIOS, and the terminal function is handled primarily in the Z80 ROM.

The following section shows the various terminal emulation protocols supported by Commodore 128 CP/M.

# TERMINAL EMULATION PROTOCOLS

| FUNCTION | CHARACTER SEQUENCE | HEX CHAR CODE |
|---|---|---|
| **Position Cursor** | **ESC (row# + 32) (col# + 32)** | **1B 3D 20 + 20 +** |
| **Cursor Left** | **Control H** | **08** |
| **Cursor Right** | **Control L** | **0C** |
| **Cursor Down** | **Control J** | **0A** |
| **Cursor Up** | **Control K** | **0B** |
| **Home and Clear Screen** | **Control Z** | **1A** |
| **Carriage Return** | **Control M** | **0D** |
| **Escape** | **Control [** | **1B** |
| **Bell** | **Control G** | **07** |

**NOTE:** Display is 24 (1–24) by 80 (1–80); cursor origin is always 1/1.

**Figure 14–4. Lear Siegler ADM-3A Protocol**

**NOTE:** The following have been added to allow the system to emulate the KayPro II display more closely.

| FUNCTION | CHARACTER SEQUENCE |
|---|---|
| Home Cursor | Control ↑ |
| CEL (Clear to End of Line) | Control-X |
| CES (Clear to End of Screen) | Control-W |

| | CHARACTER SEQUENCE | HEX CHARACTER CODES |
|---|---|---|
| **Clear to end of line** | **ESC T** | **1B 54** |
| | **ESC t** | **1B 74** |
| **Clear to end of screen** | **ESC Y** | **1B 59** |
| | **ESC y** | **1B 79** |
| **Home cursor and clear screen** | **ESC :** | **1B 3A** |
| | **ESC *** | **1B 2A** |
| **Half intensity on** | **ESC )** | **1B 29** |
| **Half intensity off** | **ESC (** | **1B 28** |
| **Reverse video on** | **ESC G4** | **1B 47 34** |
| **Blinking on** | **ESC G2** | **1B 47 32** |
| **Underline on *** | **ESC G3** | **1B 47 33** |
| **Select alternate character set*** | **ESC G1** | **1B 47 31** |
| **Reverse video and blinking off** | **ESC G0** | **1B 47 30** |
| **Insert line** | **ESC E** | **1B 45** |
| **Insert character** | **ESC Q** | **1B 51** |
| **Delete line** | **ESC R** | **1B 52** |
| **Delete character** | **ESC W** | **1B 57** |
| **Set screen colors*** | **ESC ESC ESC color #** | |
| | **where color # = 20h to 2Fh—character color** | **Physical Colors** |
| | **30h to 3Fh—background color** | **Physical Colors** |
| | **40h to 4Fh—border color (40 columns only)** | **Physical Colors** |
| | **50h to 50Fh—character color** | **Logical Colors** |
| | **60h to 60Fh—background color** | **Logical Colors** |
| | **70h to 70Fh—border color (40 columns only)** | **Logical Colors** |

*Indicates this is not a normal ADM31 sequence.

**Note:** Display is 24 (1–24) by 80 (1–80); cursor origin is always 1/1.

**Figure 14–5. Lear Siegler ADM-31 Protocol**

# SYSTEM OPERATIONS

## SETTING SYSTEM TIME

The time of day is set with this function. The time of day is stored in packed BCD format in the System Control Block (SCB) in three locations (hours, minutes, seconds). This routine reads the SCB time and writes that time to the time of day clock within the 6526. This time is updated on the chip and is used by CP/M. The Z80 is able to read/write the 6526 directly.

## UPDATING SYSTEM TIME

The SCB time is updated from the time of day clock on the 6526 by doing a system call.

# 8502 BIOS ORGANIZATION

The 8502 is responsible for most of the low-level I/O functions. The request for these functions is made through a set of mailboxes. Once the mailboxes are set up, the Z80 shuts down and the 8502 starts up (BIOS85). The 8502 looks at the command in the mailbox and performs the required task, sets the command status and shuts down. The Z80 is re-enabled; it then looks at the command status and takes the appropriate actions.

The 8502 BIOS commands are defined in Appendix K.

This concludes the summary explanation of the Commodore 128 CP/M system. However, the Commodore 128 CP/M System includes many additional Commodore 128-dependent routines and functions that are performed by the Z80 microprocessor. Since most of these routines, system calls and functions are tabular information, they are covered in Appendix K. For information on any of the following topics, refer to Appendix K.

8502 BIOS commands
All Commodore 128 Z80 system-dependent user functions
Calling a CP/M BIOS, 8502 BIOS and CP/M User system functions in Z80 machine language
More information on MFM disk formats
Commodore 128 CP/M (Z80) Memory Map

For more information on the general (non-Commodore 128-dependent) CP/M 3.0 system, see the offer in section 15 of the *C128 System Guide* for the Digital Research *CP/M Plus User's Guide, Programmer's Guide* and *System Guide*.

# 15

# THE COMMODORE 128 AND COMMODORE 64 MEMORY MAPS

This chapter provides the memory maps for both C128 and C64 modes. A memory map tells you exactly how memory is laid out internally in both RAM and ROM. It tells you exactly what resides in each memory location. The memory map directs you in finding address vectors for routines and entry points and provides information about the general layout of the computer. The memory map is probably the most vital programming tool. Refer to the memory map whenever you need directions throughout the memory of your Commodore 128. Addresses listed with more than one address label are used for more than one purpose. To BASIC, the variable has one purpose; to the Machine Language Monitor, it may have another.

The conventions used for the memory maps are as follows:

| Column 1 | Column 2 | Column 3 | Column 4 |
|---|---|---|---|
| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |

See Appendix K for the Z80 memory map for CP/M on the Commodore 128.

# C128 MEMORY MAP

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| D6510 | 0000 | 0 | 6510 DATA DIRECTION REGISTER |
| R6510 | 0001 | 1 | 6510 DATA REGISTER |
| BANK | 0002 | 2 | TOKEN 'SEARCH' LOOKS FOR, OR BANK # |
| PC_HI | 0003 | 3 | ADDRESS FOR BASIC SYS COMMAND OR MONITOR AND LONG CALL/JUMP ROUTINES |
| PC_LO | 0004 | 4 | ADDRESS, STATUS, A-REG, X-REG, Y-REG |
| S_REG | 0005 | 5 | STATUS REG TEMP |
| A_REG | 0006 | 6 | .A REG TEMP |
| X_REG | 0007 | 7 | .X REG TEMP |
| Y_REG | 0008 | 8 | .Y REG TEMP |
| STKPTR | 0009 | 9 | STACK POINTER TEMP |
| | | | BASIC ZERO PAGE STORAGE |
| INTEGR | 0009 | 9 | |
| CHARAC | | | SEARCH CHARACTER |
| ENDCHR | 000A | 10 | FLAG: SCAN FOR QUOTE AT END OF STRING |
| TRMPOS | 000B | 11 | SCREEN COLUMN FROM LAST TAB |
| VERCK | 000C | 12 | FLAG: 0 = LOAD, 1 = VERIFY |

*C128 Memory Map (continued)*

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| COUNT | 000D | 13 | INPUT BUF.PTR / # OF SUBSCRIPTS |
| DIMFLG | 000E | 14 | FLAG: DEFAULT ARRAY DIMENSION |
| VALTYP | 000F | 15 | DATA TYPE: $FF = STRING, $00 = NUMERIC |
| INTFLG | 0010 | 16 | DATA TYPE: $00 = FLOAT.PT, $80 = INTEGER |
| GARBFL | 0011 | 17 | FLAG: DATA SCAN / LIST QUOTE / GARBAGE COLLECTION |
| DORES | | | |
| SUBFLG | 0012 | 18 | FLAG: SUBSCRIPT REF. / USER FUNC. CALL |
| INPFLG | 0013 | 19 | FLAG: $00 = INPUT, $40 = GET, $98 = READ |
| DOMASK | 0014 | 20 | |
| TANSGN | | | FLAG: TAN SIGN / COMPARISON RESULT |
| CHANNL | 0015 | 21 | |
| POKER | 0016 | 22 | |
| LINNUM | | | TEMP INTEGER VALUE |
| TEMPPT | 0018 | 24 | POINTER: TEMP STRING STACK |
| LASTPT | 0019 | 25 | LAST TEMP STRING ADDRESS |
| TEMPST | 001B | 27 | STACK FOR TEMP STRINGS |
| INDEX | 0024 | 36 | UTILITY POINTER AREA |
| INDEX1 | | | |
| INDEX2 | 0026 | 38 | |
| RESHO | 0028 | 40 | FLOAT.PT. PRODUCT OF MULTIPLY |
| RESMOH | 0029 | 41 | |
| ADDEND | 002A | 42 | |
| RESMO | | | |
| RESLO | 002B | 43 | |
| TXTTAB | 002D | 45 | POINTER: START OF BASIC TEXT |
| VARTAB | 002F | 47 | POINTER: START OF BASIC VARIABLES |
| ARYTAB | 0031 | 49 | POINTER: START OF BASIC ARRAYS |
| STREND | 0033 | 51 | POINTER: END OF BASIC ARRAYS + 1 |
| FRETOP | 0035 | 53 | POINTER: BOTTOM OF STRING STORAGE |
| FRESPC | 0037 | 55 | UTILITY STRING POINTER |
| MAX_MEM_1 | 0039 | 57 | TOP OF STRING/VARIABLE BANK (BANK 1) |
| CURLIN | 003B | 59 | CURRENT BASIC LINE NUMBER |
| TXTPTR | 003D | 61 | POINTER TO BASIC TEXT USED BY CHRGET,ETC. |
| FORM | 003F | 63 | USED BY PRINT USING |
| FNDPNT | | | POINTER TO ITEM FOUND BY SEARCH |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | BASIC ZERO PAGE STORAGE | |
| DATLIN | 0041 | 65 | CURRENT DATA LINE NUMBER |
| DATPTR | 0043 | 67 | CURRENT DATA ITEM ADDRESS |
| INPPTR | 0045 | 69 | VECTOR: INPUT ROUTINE |
| VARNAM | 0047 | 71 | CURRENT BASIC VARIABLE NAME |
| FDECPT | 0049 | 73 | |
| VARPNT | | | POINTER: CURRENT BASIC VARIABLE DATA |
| LSTPNT | 004B | 75 | |
| FORPNT | | | POINTER: INDEX VARIABLE FOR FOR/NEXT |
| ANDMSK | | | |
| EORMSK | 004C | 76 | |
| VARTXT | 004D | 77 | |
| OPPTR | | | |
| OPMASK | 004F | 79 | |
| GRBPNT | 0050 | 80 | |
| TEMPF3 | | | |
| DEFPNT | | | |
| DSCPNT | 0052 | 82 | |
| | 0054 | 84 | |
| HELPER | 0055 | 85 | FLAGS 'HELP' OR 'LIST' |
| JMPER | 0056 | 86 | |
| | 0057 | 87 | |
| OLDOV | 0058 | 88 | |
| TEMPF1 | 0059 | 89 | |
| PTARG1 | | | MULTIPLY DEFINED FOR INSTR |
| PTARG2 | 005B | 91 | |
| STR1 | 005D | 93 | |
| STR2 | 0060 | 96 | |
| POSITN | 0063 | 99 | |
| MATCH | 0064 | 100 | |
| ARYPNT | 005A | 90 | |
| HIGHDS | | | |
| HIGHTR | 005C | 92 | |
| TEMPF2 | 005E | 94 | |
| DECCNT | 005F | 95 | NUMBER OF DIGITS AFTER THE DECIMAL POINT |
| TENEXP | 0060 | 96 | |
| T0 | | | ML MONITOR Z.P. STORAGE IN FAC |
| GRBTOP | 0061 | 97 | |
| DPTFLG | | | DECIMAL POINT FLAG |
| LOWTR | | | |
| EXPSGN | 0062 | 98 | |
| FAC | 0063 | 99 | |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | BASIC ZERO PAGE STORAGE | |
| DSCTMP | | | |
| LEFT_FLAG | | | PAINT-LEFT FLAG |
| FACEXP | | | FAC#1 EXPONENT |
| T1 | | | MONITOR Z.P. STORAGE IN FAC |
| RIGHT_FLAG | 0064 | 100 | PAINT-RIGHT FLAG |
| FACHO | | | FAC#1 MANTISSA |
| FACMOH | 0065 | 101 | |
| INDICE | 0066 | 102 | |
| FACMO | | | |
| T2 | | | MONITOR Z.P. STORAGE IN FAC |
| FACLO | 0067 | 103 | |
| FACSGN | 0068 | 104 | FAC#1 SIGN |
| DEGREE | 0069 | 105 | |
| SGNFLG | | | POINTER:SERIES-EVAL.CONSTANT |
| ARGEXP | 006A | 106 | FAC#2 EXPONENT |
| ARGHO | 006B | 107 | FAC#2 MANTISSA |
| ARGMOH | 006C | 108 | |
| INIT_AS_0 | | | JUST A COUNT FOR INIT |
| ARGMO | 006D | 109 | |
| ARGLO | 006E | 110 | |
| ARGSGN | 006F | 111 | FAC#2 SIGN |
| STRNG1 | 0070 | 112 | |
| ARISGN | | | SIGN COMPARISON RESULT: FAC#1 VS #2 |
| FACOV | 0071 | 113 | FAC#1 LOW-ORDER (ROUNDING) |
| STRNG2 | 0072 | 114 | |
| POLYPT | | | |
| CURTOL | | | |
| FBUFPT | | | POINTER: CASSETTE BUFFER |
| AUTINC | 0074 | 116 | INC. VAL FOR AUTO (0=OFF) |
| MVDFLG | 0076 | 118 | FLAG IF 10K HIRES ALLOCATED |
| Z_P_TEMP_1 | 0077 | 119 | PRINT USING'S LEADING ZERO COUNTER<br>MOVSPR & SPRITE TEMPORARY<br>MID$ TEMPORARY |
| HULP | 0078 | 120 | COUNTER |
| KEYSIZ | | | |
| SYNTMP | 0079 | 121 | USED AS TEMP FOR INDIRECT LOADS |
| DSDESC | 007A | 122 | DESCRIPTOR FOR DS$ |
| TXTPTR | | | MONITOR Z.P. STORAGE |
| TOS | 007D | 125 | TOP OF RUN TIME STACK |
| RUNMOD | 007F | 127 | FLAGS RUN/DIRECT MODE |
| PARSTS | 0080 | 128 | DOS PARSER STATUS WORD |
| POINT | | | USING'S POINTER TO DEC.PT |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | **BASIC ZERO PAGE STORAGE** | |
| PARSTX | 0081 | 129 | |
| OLDSTK | 0082 | 130 | |
| | | **BASIC Z-P STORAGE FOR GRAPHIC COMMANDS** | |
| COLSEL | 0083 | 131 | CURRENT COLOR SELECTED |
| MULTICOLOR_1 | 0084 | 132 | |
| MULTICOLOR_2 | 0085 | 133 | |
| FOREGROUND | 0086 | 134 | |
| SCALE_X | 0087 | 135 | SCALE FACTOR IN X |
| SCALE_Y | 0089 | 137 | SCALE FACTOR IN Y |
| STOPNB | 008B | 139 | STOP PAINT IF NOT BACKGROUND/ NOT SAME COLOR |
| GRAPNT | 008C | 140 | |
| VTEMP1 | 008E | 142 | |
| VTEMP2 | 008F | 143 | |
| | | **KERNAL/EDITOR STORAGE** | |
| STATUS | 0090 | 144 | I/O OPERATION STATUS BYTE |
| STKEY | 0091 | 145 | STOP KEY FLAG |
| SVXT | 0092 | 146 | TAPE TEMPORARY |
| VERCK | 0093 | 147 | LOAD OR VERIFY FLAG |
| C3P0 | 0094 | 148 | SERIAL BUFFERED CHAR FLAG |
| BSOUR | 0095 | 149 | CHAR BUFFER FOR SERIAL |
| SYNO | 0096 | 150 | CASSETTE SYNC # |
| XSAV | 0097 | 151 | TEMP FOR BASIN |
| LDTND | 0098 | 152 | INDEX TO LOGICAL FILE |
| DFLTN | 0099 | 153 | DEFAULT INPUT DEVICE # |
| DFLTO | 009A | 154 | DEFAULT OUTPUT DEVICE # |
| PRTY | 009B | 155 | CASSETTE PARITY |
| DPSW | 009C | 156 | CASSETTE DIPOLE SWITCH |
| MSGFLG | 009D | 157 | OS MESSAGE FLAG |
| PTR1 | 009E | 158 | CASSETTE ERROR PASS1 |
| T1 | | | TEMPORARY 1 |
| PTR2 | 009F | 159 | CASSETTE ERROR PASS2 |
| T2 | | | TEMPORARY 2 |
| TIME | 00A0 | 160 | 24 HOUR CLOCK IN 1/60TH SECONDS |
| R2D2 | 00A3 | 163 | SERIAL BUS USAGE |
| PCNTR | | | CASSETTE |
| BSOUR1 | 00A4 | 164 | TEMP USED BY SERIAL ROUTINE |
| FIRT | | | |
| COUNT | 00A5 | 165 | TEMP USED BY SERIAL ROUTINE |
| CNTDN | | | CASSETTE SYNC COUNTDOWN |
| BUFPT | 00A6 | 166 | CASSETTE BUFFER POINTER |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | KERNAL/EDITOR STORAGE | |
| INBIT | 00A7 | 167 | RS-232 RCVR INPUT BIT STORAGE |
| SHCNL | | | CASSETTE SHORT COUNT |
| BITCI | 00A8 | 168 | RS-232 RCVR BIT COUNT IN |
| RER | | | CASSETTE READ ERROR |
| RINONE | 00A9 | 169 | RS-232 RCVR FLAG FOR START BIT CHECK |
| REZ | | | CASSETTE READING ZEROES |
| RIDATA | 00AA | 170 | RS-232 RCVR BYTE BUFFER |
| RDFLG | | | CASSETTE READ MODE |
| RIPRTY | 00AB | 171 | RS-232 RCVR PARITY STORAGE |
| SHCNH | | | CASSETTE SHORT CNT |
| SAL | 00AC | 172 | POINTER: TAPE BUFFER / SCREEN SCROLLING |
| SAH | 00AD | 173 | |
| EAL | 00AE | 174 | TAPE END ADDRESSES / END OF PROGRAM |
| EAH | 00AF | 175 | |
| CMP0 | 00B0 | 176 | TAPE TIMING CONSTANTS |
| TEMP | 00B1 | 177 | |
| TAPE1 | 00B2 | 178 | ADDRESS OF TAPE BUFFER |
| BITTS | 00B4 | 180 | RS-232 TRNS BIT COUNT |
| SNSW1 | | | |
| NXTBIT | 00B5 | 181 | RS-232 TRNS NEXT BIT TO BE SENT |
| DIFF | | | |
| RODATA | 00B6 | 182 | RS-232 TRNS BYTE BUFFER |
| PRP | | | |
| FNLEN | 00B7 | 183 | LENGTH CURRENT FILE N STR |
| LA | 00B8 | 184 | CURRENT FILE LOGICAL ADDR |
| SA | 00B9 | 185 | CURRENT FILE 2ND ADDR |
| FA | 00BA | 186 | CURRENT FILE PRIMARY ADDR |
| FNADR | 00BB | 187 | ADDR CURRENT FILE NAME STR |
| ROPRTY | 00BD | 189 | RS-232 TRNS PARITY BUFFER |
| OCHAR | | | |
| FSBLK | 00BE | 190 | CASSETTE READ BLOCK COUNT |
| DRIVE | 00BF | 191 | |
| MYCH | | | SERIAL WORD BUFFER |
| CAS1 | 00C0 | 192 | CASSETTE MANUAL/CNTRLED SWITCH (UPDATED DURING IRQ) |
| TRACK | 00C1 | 193 | |
| STAL | | | I/O START ADDRESS (LO) |
| SECTOR | 00C2 | 194 | |
| STAH | | | I/O START ADDRESS (HI) |
| MEMUSS | 00C3 | 195 | CASSETTE LOAD TEMPS (2 BYTES) |
| TMP2 | | | |
| DATA | 00C5 | 197 | TAPE READ/WRITE DATA |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | KERNAL/EDITOR STORAGE | |
| BA | 00C6 | 198 | BANK FOR CURRENT LOAD/SAVE/ VERIFY OPERATION |
| FNBANK | 00C7 | 199 | BANK WHERE CURRENT FN IS FOUND (AT 'FNADR') |
| RIBUF | 00C8 | 200 | RS-232 INPUT BUFFER POINTER |
| ROBUF | 00CA | 202 | RS-232 OUTPUT BUFFER POINTER |
| | | GLOBAL SCREEN EDITOR VARIABLES | |
| KEYTAB | 00CC | 204 | KEYSCAN TABLE POINTER |
| IMPARM | 00CE | 206 | PRIMM UTILITY STRING POINTER |
| NDX | 00D0 | 208 | INDEX TO KEYBOARD QUEUE |
| KYNDX | 00D1 | 209 | PENDING FUNCTION KEY FLAG |
| KEYIDX | 00D2 | 210 | INDEX INTO PENDING FUNCTION KEY STRING |
| SHFLAG | 00D3 | 211 | KEYSCAN SHIFT KEY STATUS |
| SFDX | 00D4 | 212 | KEYSCAN CURRENT KEY INDEX |
| LSTX | 00D5 | 213 | KEYSCAN LAST KEY INDEX |
| CRSW | 00D6 | 214 | <CR> INPUT FLAG |
| MODE | 00D7 | 215 | 40/80 COLUMN MODE FLAG |
| GRAPHM | 00D8 | 216 | TEXT/GRAPHIC MODE FLAG |
| CHAREN | 00D9 | 217 | RAM/ROM VIC CHARACTER FETCH FLAG (BIT-2) |
| | | THE FOLLOWING LOCATIONS ARE SHARED BY SEVERAL EDITOR ROUTINES. | |
| SEDSAL | 00DA | 218 | POINTERS FOR MOVLIN |
| BITMSK | 00DA | 218 | TEMPORARY FOR TAB & LINE WRAP ROUTINES |
| SAVER | 00DB | 219 | ANOTHER TEMPORARY PLACE TO SAVE A REG. |
| SEDEAL | 00DC | 220 | |
| SEDT1 | 00DE | 222 | SAVPOS |
| SEDT2 | 00DF | 223 | |
| KEYSIZ | 00DA | 218 | PROGRAMMABLE KEY VARIABLES |
| KEYLEN | 00DB | 219 | |
| KEYNUM | 00DC | 220 | |
| KEYNXT | 00DD | 221 | |
| KEYBNK | 00DE | 222 | |
| KEYTMP | 00DF | 223 | |
| | | LOCAL SCREEN EDITOR VARIABLES. THESE ARE SWAPPED OUT TO $0A40 WHEN SCREEN (40/80) MODE CHANGES. | |
| PNT | 00E0 | 224 | POINTER TO CURRENT LINE (TEXT) |
| USER | 00E2 | 226 | POINTER TO CURRENT LINE (ATTRIBUTE) |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | | LOCAL SCREEN EDITOR VARIABLES. THESE ARE SWAPPED OUT TO $0A40 WHEN SCREEN (40/80) MODE CHANGES. |
| SCBOT | 00E4 | 228 | WINDOW LOWER LIMIT |
| SCTOP | 00E5 | 229 | WINDOW UPPER LIMIT |
| SCLF | 00E6 | 230 | WINDOW LEFT MARGIN |
| SCRT | 00E7 | 231 | WINDOW RIGHT MARGIN |
| LSXP | 00E8 | 232 | CURRENT INPUT COLUMN START |
| LSTP | 00E9 | 233 | CURRENT INPUT LINE START |
| INDX | 00EA | 234 | CURRENT INPUT LINE END |
| TBLX | 00EB | 235 | CURRENT CURSOR LINE |
| PNTR | 00EC | 236 | CURRENT CURSOR COLUMN |
| LINES | 00ED | 237 | MAXIMUM NUMBER OF SCREEN LINES |
| COLUMNS | 00EE | 238 | MAXIMUM NUMBER OF SCREEN COLUMNS |
| DATAX | 00EF | 239 | CURRENT CHARACTER TO PRINT |
| LSTCHR | 00F0 | 240 | PREVIOUS CHAR PRINTED (FOR <ESC> TEST) |
| COLOR | 00F1 | 241 | CURR ATTRIBUTE TO PRINT (DEFAULT FGND COLOR) |
| TCOLOR | 00F2 | 242 | SAVED ATTRIB TO PRINT ('INSERT' & 'DELETE') |
| RVS | 00F3 | 243 | REVERSE MODE FLAG |
| QTSW | 00F4 | 244 | QUOTE MODE FLAG |
| INSRT | 00F5 | 245 | INSERT MODE FLAG |
| INSFLG | 00F6 | 246 | AUTO-INSERT MODE FLAG |
| LOCKS | 00F7 | 247 | DISABLES <SHIFT><C=>, <CTRL> S |
| SCROLL | 00F8 | 248 | DISABLES SCREEN SCROLL, LINE LINKER |
| BEEPER | 00F9 | 249 | DISABLES <CTRL> G |
| FREKZP | 00FA | 250 | FREE ZERO PAGE RESERVED FOR APPLICATIONS SOFTWARE ($FA–$FE) |
| LOFBUF | 00FF | 255 | |
| | | | BASIC/DOS INTERFACE VARS |
| BAD | 0100 | 256 | TAPE READ ERRORS |
| FBUFFR | | | AREA TO BUILD FILENAME IN (16 BYTES) |
| XCNT | 0110 | 272 | DOS LOOP COUNTER |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | | **BASIC/DOS INTERFACE VARS** |
| DOSF1L | 0111 | 273 | DOS FILENAME 1 LEN |
| DOSDS1 | 0112 | 274 | DOS DISK DRIVE 1 |
| DOSF2L | 0113 | 275 | DOS FILENAME 2 LEN |
| DOSDS2 | 0114 | 276 | DOS DISK DRIVE 2 |
| DOSF2A | 0115 | 277 | DOS FILENAME 2 ADDR |
| DOSOFL | 0117 | 279 | BLOAD/BSAVE STARTING ADDRESS |
| DOSOFH | 0119 | 281 | ....AND ENDING ADDRESS |
| DOSLA | 011B | 283 | DOS LOGICAL ADDR |
| DOSFA | 011C | 284 | DOS PHYS ADDR |
| DOSSA | 011D | 285 | DOS SEC. ADDR |
| DOSRCL | 011E | 286 | DOS RECORD LENGTH |
| DOSBNK | 011F | 287 | |
| DOSDID | 0120 | 288 | DOS DISK ID |
| DIDCHK | 0122 | 290 | DOS DSK ID FLG |
| | | | SPACE USED BY PRINT USING |
| BNR | 0123 | 291 | POINTER TO BEGIN. NO. |
| ENR | 0124 | 292 | POINTER TO END NO. |
| DOLR | 0125 | 293 | DOLLAR FLAG |
| FLAG | 0126 | 294 | COMMA FLAG |
| SWE | 0127 | 295 | COUNTER |
| USGN | 0128 | 296 | SIGN EXPONENT |
| UEXP | 0129 | 297 | POINTER TO EXPONENT |
| VN | 012A | 298 | # OF DIGITS BEFORE DECIMAL POINT |
| CHSN | 012B | 299 | JUSTIFY FLAG |
| VF | 012C | 300 | # OF POS BEFORE DECIMAL POINT (FIELD) |
| NF | 012D | 301 | # OF POS AFTER DECIMAL POINT (FIELD) |
| POSP | 012E | 302 | +/– FLAG (FIELD) |
| FESP | 012F | 303 | EXPONENT FLAG (FIELD) |
| ETOF | 0130 | 304 | SWITCH |
| CFORM | 0131 | 305 | CHAR COUNTER (FIELD) |
| SNO | 0132 | 306 | SIGN NO |
| BLFD | 0133 | 307 | BLANK/STAR FLAG |
| BEGFD | 0134 | 308 | POINTER TO BEGIN OF FIELD |
| LFOR | 0135 | 309 | LENGTH OF FORMAT |
| ENDFD | 0136 | 310 | POINTER TO END OF FIELD |
| SYSTK | 0137 | 311 | SYSTEM STACK ($0137–$01FF) |
| BUF | 0200 | 512 | INPUT BUFFER: BASIC & MONITOR ($0200–$02A1) |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | BASIC/DOS INTERFACE VARS | |
| FETCH | 02A2 | 674 | LDA(–),Y FROM ANY BANK |
| FETVEC | 02AA | 682 | |
| STASH | 02AF | 687 | STA(–),Y TO ANY BANK |
| STAVEC | 02B9 | 697 | |
| CMPARE | 02BE | 702 | CMP(–),Y TO ANY BANK |
| CMPVEC | 02C8 | 712 | |
| JSRFAR | 02CD | 716 | JSR XXXX TO ANY BANK & RETURN |
| JMPFAR | 02E3 | 739 | JMP XXXX TO ANY BANK |
| | | VECTORS | |
| ESC_FN_VEC | 02FC | 764 | VECTOR FOR ADDITIONAL FUNCTION ROUTINES |
| BNKVEC | 02FE | 766 | VECTOR FOR FUNCTION CART. USERS |
| IERROR | 0300 | 768 | VECTOR FOR PRINT BASIC ERROR (ERR IN .X) |
| IMAIN | 0302 | 770 | VECTOR TO MAIN (SYSTEM DIRECT LOOP) |
| ICRNCH | 0304 | 772 | VECTOR TO CRUNCH (TOKENIZATION ROUTINE) |
| IQPLOP | 0306 | 774 | VECTOR TO LIST BASIC TEXT (CHAR LIST) |
| IGONE | 0308 | 776 | VECTOR TO GONE (BASIC CHAR DISPATCH) |
| IEVAL | 030A | 778 | VECTOR TO BASIC TOKEN EVALUATION |
| IESCLK | 030C | 780 | VECTOR TO ESCAPE-TOKEN |
| IESCPR | 030E | 782 | CRUNCH, |
| IESCEX | 0310 | 784 | ...LIST, |
| IIRQ | 0314 | 788 | ...AND EXECUTE. |
| CINV | | | IRQ RAM VECTOR |
| IBRK | 0316 | 790 | BRK INSTR RAM VECTOR → |
| CBINV | | | |
| INMI | 0318 | 792 | NMI VECTOR |
| IOPEN | 031A | 794 | KERNAL OPEN ROUTINE VECTOR |
| ICLOSE | 031C | 796 | KERNAL CLOSE ROUTINE VECTOR |
| ICHKIN | 031E | 798 | KERNAL CHKIN ROUTINE VECTOR |
| ICKOUT | 0320 | 800 | KERNAL CHKOUT ROUTINE VECTOR |
| ICLRCH | 0322 | 802 | KERNAL CLRCHN ROUTINE VECTOR |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | VECTORS | |
| IBASIN | 0324 | 804 | KERNAL CHRIN ROUTINE VECTOR |
| IBSOUT | 0326 | 806 | KERNAL CHROUT ROUTINE VECTOR |
| ISTOP | 0328 | 808 | KERNAL STOP ROUTINE VECTOR |
| IGETIN | 032A | 810 | KERNAL GETIN ROUTINE VECTOR |
| ICLALL | 032C | 812 | KERNAL CLALL ROUTINE VECTOR |
| EXMON | 032E | 814 | MONITOR COMMAND VECTOR |
| ILOAD | 0330 | 816 | KERNAL LOAD ROUTINE VECTOR |
| ISAVE | 0332 | 818 | KERNAL SAVE ROUTINE VECTOR |
| | | EDITOR INDIRECT VECTORS | |
| CTLVEC | 0334 | 820 | EDITOR: PRINT 'CONTRL' INDIRECT |
| SHFVEC | 0336 | 822 | EDITOR: PRINT 'SHIFTD' INDIRECT |
| ESCVEC | 0338 | 824 | EDITOR: PRINT 'ESCAPE' INDIRECT |
| KEYVEC | 033A | 826 | EDITOR: KEYSCAN LOGIC INDIRECT |
| KEYCHK | 033C | 828 | EDITOR: STORE KEY INDIRECT |
| DECODE | 033E | 830 | VECTORS TO KEYBOARD MATRIX DECODE TABLES |
| KEYD | 034A | 842 | IRQ KEYBOARD BUFFER (10 BYTES) |
| TABMAP | 0354 | 852 | BITMAP OF TAB STOPS (10 BYTES, $0354-D) |
| BITABL | 035E | 862 | BITMAP OF LINE WRAPS TABMAP AND BITABL GET SWAPPED TO $0A60 WHEN SCREEN 40/80 MODE IS CHANGED. |
| LAT | 0362 | 866 | LOGICAL FILE NUMBERS |
| FAT | 036C | 876 | PRIMARY DEVICE NUMBERS |
| SAT | 0376 | 886 | SECONDARY ADDRESSES |
| CHRGET | 0380 | 896 | |
| CHRGOT | 0386 | 902 | |
| QNUM | 0390 | 912 | |
| | | INDIRECT LOAD SUBROUTINE AREA | |
| INDSUB_RAM0 | 039F | 927 | SHARED ROM FETCH SUB |
| INDSUB_RAM1 | 03AB | 939 | SHARED ROM FETCH SUB |
| INDIN1_RAM1 | 03B7 | 950 | INDEX1 INDIRECT FETCH |
| INDIN2 | 03C0 | 959 | INDEX2 INDIRECT FETCH |
| INDTXT | 03C9 | 968 | TXTPTR |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | | INDIRECT LOAD SUBROUTINE AREA |
| ZERO | 03D2 | 977 | NUMERIC CONSTANT FOR BASIC |
| CURRENT_ BANK | 03D5 | 979 | CONTEXT FOR SYS,POKE,PEEK FROM BANK CMMD |
| TMPDES | 03D6 | 980 | TEMP FOR INSTR |
| FIN_BANK | 03DA | 984 | BANK POINTER FOR STRING/ NUMBER CONVERT RTN |
| SAVSIZ | 03DB | 985 | TEMP WORK LOCATIONS FOR SSHAPE |
| BITS | 03DF | 989 | FAC#1 OVERFLOW DIGIT |
| SPRTMP_1 | 03E0 | 990 | TEMP FOR SPRSAV |
| SPRTMP_2 | 03E1 | 991 | |
| FG_BG | 03E2 | 992 | PACKED FOREGROUND/ BACKGROUND COLOR NYBBLES |
| FG_MC1 | 03E3 | 993 | PACKED FOREGROUND/ MULTICOLOR 1 COLOR NYBBLES |
| | | | PAGE FOUR & HIGHER DECLARATIONS |
| | | | (BEGINNING OF BANKABLE RAM) |
| VICSCN | 0400 | 1024 | VIDEO MATRIX #1: VIC 40-COLUMN TEXT SCREEN $0400–$07FF |
| | 0800 | 2048 | BASIC RUN-TIME STACK (512 BYTES) $0800–$09FF |
| | | | ABSOLUTE KERNAL VARIABLES |
| SYSTEM_VECTOR | 0A00 | 2560 | VECTOR TO RESTART SYSTEM (BASIC WARM) |
| DEJAVU | 0A02 | 2562 | KERNAL WARM/COLD INIT'N STATUS BYTE |
| PALNTS | 0A03 | 2563 | PAL/NTSC SYSTEM FLAG |
| INIT_STATUS | 0A04 | 2564 | FLAGS RESET VS. NMI STATUS FOR INIT'N RTNS |
| MEMSTR | 0A05 | 2565 | PTR TO BOTTOM OF AVAIL. MEMORY IN SYSTEM BANK |
| MEMSIZ | 0A07 | 2567 | PTR TO TOP OF AVAILABLE MEMORY IN SYSTEM BANK |
| IRQTMP | 0A09 | 2569 | TAPE HANDLER PRESERVES IRQ INDIRECT HERE |
| CASTON | 0A0B | 2571 | TOD SENSE DURING TAPE OPERATIONS |
| KIKA26 | 0A0C | 2572 | TAPE READ TEMPORARY |
| STUPID | 0A0D | 2573 | TAPE READ D1IRQ INDICATOR |
| TIMOUT | 0A0E | 2574 | FAST SERIAL TIMEOUT FLAG |
| ENABL | 0A0F | 2575 | RS-232 ENABLES |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | ABSOLUTE KERNAL VARIABLES | |
| M51CTR | 0A10 | 2576 | RS-232 CONTROL REGISTER |
| M51CDR | 0A11 | 2577 | RS-232 COMMAND REGISTER |
| M51AJB | 0A12 | 2578 | RS-232 USER BAUD RATE |
| RSSTAT | 0A14 | 2580 | RS-232 STATUS REGISTER |
| BITNUM | 0A15 | 2581 | RS-232 NUMBER OF BITS TO SEND |
| BAUDOF | 0A16 | 2582 | RS-232 BAUD RATE FULL BIT TIME (CREATED BY OPEN) |
| RIDBE | 0A18 | 2584 | RS-232 INPUT BUFFER INDEX TO END |
| RIDBS | 0A19 | 2585 | RS-232 INPUT BUFFER INDEX TO START |
| RODBS | 0A1A | 2586 | RS-232 OUTPUT BUFFER INDEX TO START |
| RODBE | 0A1B | 2587 | RS-232 OUTPUT BUFFER INDEX TO END |
| SERIAL | 0A1C | 2588 | FAST SERIAL INTERNAL/EXTERNAL FLAG |
| TIMER | 0A1D | 2589 | DECREMENTING JIFFIE REGISTER |
| | | GLOBAL ABSOLUTE SCREEN EDITOR DECLARATIONS | |
| XMAX | 0A20 | 2592 | KEYBOARD QUEUE MAXIMUM SIZE |
| PAUSE | 0A21 | 2593 | <CTRL>S FLAG |
| RPTFLG | 0A22 | 2594 | ENABLE KEY REPEATS |
| KOUNT | 0A23 | 2595 | DELAY BETWEEN KEY REPEATS |
| DELAY | 0A24 | 2596 | DELAY BEFORE A KEY STARTS REPEATING |
| LSTSHF | 0A25 | 2597 | DELAY BETWEEN <C=><SHFT> TOGGLES |
| BLNON | 0A26 | 2598 | VIC CURSOR MODE (BLINKING, SOLID) |
| BLNSW | 0A27 | 2599 | VIC CURSOR DISABLE |
| BLNCT | 0A28 | 2600 | VIC CURSOR BLINK COUNTER |
| GDBLN | 0A29 | 2601 | VIC CURSOR CHARACTER BEFORE BLINK |
| GDCOL | 0A2A | 2602 | VIC CURSOR COLOR BEFORE BLINK |
| CURMOD | 0A2B | 2603 | VDC CURSOR MODE (WHEN ENABLED) |
| VM1 | 0A2C | 2604 | VIC TEXT SCREEN/CHARACTER BASE POINTER |
| VM2 | 0A2D | 2605 | VIC BIT-MAP BASE POINTER |
| VM3 | 0A2E | 2606 | VDC TEXT SCREEN BASE |
| VM4 | 0A2F | 2607 | VDC ATTRIBUTE BASE |
| LINTMP | 0A30 | 2608 | TEMPORARY POINTER TO LAST LINE FOR LOOP4 |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | GLOBAL ABSOLUTE SCREEN EDITOR DECLARATIONS | |
| SAV80A | 0A31 | 2609 | TEMPORARY FOR 80-COL ROUTINES |
| SAV80B | 0A32 | 2610 | TEMPORARY FOR 80-COL ROUTINES |
| CURCOL | 0A33 | 2611 | VDC CURSOR COLOR BEFORE BLINK |
| SPLIT | 0A34 | 2612 | VIC SPLIT SCREEN RASTER VALUE |
| FNADRX | 0A35 | 2613 | SAVE .X DURING BANK OPERATIONS |
| PALCNT | 0A36 | 2614 | COUNTER FOR PAL SYSTEMS (JIFFIE ADJUSTMENT) |
| SPEED | 0A37 | 2615 | SAVE SYSTEM SPEED DURING TAPE AND SERIAL OPS |
| SPRITES | 0A38 | 2616 | SAVE SPRITE ENABLES DURING TAPE AND SERIAL OPS |
| BLANKING | 0A39 | 2617 | SAVE BLANKING STATUS DURING TAPE OPS |
| HOLD_OFF | 0A3A | 2618 | FLAG SET BY USER TO RESRV FULL CNTRL OF VIC |
| LDTB1_SA | 0A3B | 2619 | HI BYTE: SA OF VIC SCRN (USE W/VM1 TO MOVE SCRN) |
| CLR_EA_LO | 0A3C | 2620 | 8563 BLOCK FILL |
| CLR_EA_HI | 0A3D | 2621 | 8563 BLOCK FILL |
| | 0A40 | 2624 | $0A40–$0A7F RESERVED SWAP AREA FOR SCREEN VARIABLES WHEN (40/80) MODE CHANGES MONITOR'S DOMAIN |
| XCNT | 0A80 | 2688 | COMPARE BUFFER (32 BYTES) |
| HULP | 0AA0 | 2720 | |
| FORMAT | 0AAA | 2730 | |
| LENGTH | 0AAB | 2731 | ASM/DIS |
| MSAL | 0AAC | 2732 | FOR ASSEMBLER |
| SXREG | 0AAF | 2735 | 1 BYTE TEMP USED ALL OVER |
| SYREG | 0AB0 | 2736 | 1 BYTE TEMP USED ALL OVER |
| WRAP | 0AB1 | 2737 | 1 BYTE TEMP FOR ASSEMBLER |
| XSAVE | 0AB2 | 2738 | SAVE .X HERE DURING INDIRECT SUBROUTINE CALLS |
| DIRECTION | 0AB3 | 2739 | DIRECTION INDICATOR FOR 'TRANSFER' |
| COUNT | 0AB4 | 2740 | PARSE NUMBER CONVERSION |
| NUMBER | 0AB5 | 2741 | PARSE NUMBER CONVERSION |
| SHIFT | 0AB6 | 2742 | PARSE NUMBER CONVERSION |
| TEMPS | 0AB7 | 2743 | |
| | | FUNCTION KEY ROM CARD TABLES | |
| CURBNK | 0AC0 | 2752 | CURRENT FUNCTION KEY ROM BANK BEING POLLED |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | FUNCTION KEY ROM CARD TABLES | |
| PAT | 0AC1 | 2753 | PHYSICAL ADDRESS TABLE(IDS OF LOGGED-IN CARDS) |
| DK_FLAG | 0AC5 | 2757 | RESERVED FOR FOREIGN SCREEN EDITORS |
| | 0AC6 | 2758 | $0AC6–$0AFF RESERVED FOR SYSTEM |
| TBUFFR | 0B00 | 2810 | CASSETTE BUFFER (192 BYTES) $0B00–$0BC0, THIS PAGE ALSO USED AS A BUFFER FOR THE DISK AUTO-BOOT |
| RS232I | 0C00 | 3072 | RS-232 INPUT BUFFER |
| RS232O | 0D00 | 3328 | RS-232 OUTPUT BUFFER |
| | 0E00 | 3584 | SPRITE DEFINITION AREA (MUST BE BELOW $1000) $0E00–$0FFF, 512 BYTES |
| PKYBUF | 1000 | 4096 | PROGRAMMABLE FUNCTION KEY LENGTHS TABLE FOR 10 KEYS (F1–F8, <SHIFT RUN>, HELP) |
| PKYDEF | 100A | 4106 | PROGRAMMABLE FUNCTION KEY STRINGS |
| | | DOS/VSP AREA | |
| DOSSTR | 1100 | 4352 | DOS OUTPUT STR. BUF 48 BYTES TO BUILD DOS STRING |
| VWORK | 1131 | 4401 | GRAPHICS VARS |
| XYPOS | 1131 | 4401 | |
| XPOS | 1131 | 4401 | CURRENT X POSITION |
| YPOS | 1133 | 4403 | CURRENT Y POSITION |
| XDEST | 1135 | 4405 | X-COORDINATE DESTINATION |
| YDEST | 1137 | 4407 | Y-COORDINATE DESTINATION |
| XYABS | 1139 | 4409 | LINE DRAWING VARIABLES |
| XABS | 1139 | 4409 | |
| YABS | 113B | 4411 | |
| XYSGN | 113D | 4413 | |
| XSGN | 113D | 4413 | |
| YSGN | 113F | 4415 | |
| FCT | 1141 | 4417 | |
| ERRVAL | 1145 | 4421 | |
| LESSER | 1147 | 4423 | |
| GREATR | 1148 | 4424 | |
| | | | ANGLE ROUTINE VARIABLES |
| ANGSGN | 1149 | 4425 | SIGN OF ANGLE |
| SINVAL | 114A | 4426 | SINE OF VALUE OF ANGLE |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | DOS/VSP AREA | |
| COSVAL | 114C | 4428 | COSINE OF VALUE OF ANGLE |
| ANGCNT | 114E | 4430 | TEMPS FOR ANGLE DISTANCE ROUTINES |
| | | BASIC GRAPHIC VARIABLES. THE FOLLOWING 24 BYTES ARE MULTIPLY DEFINED. | |
| | | | CIRCLE DRAWING VARIABLES |
| XCIRCL | 1150 | 4432 | CIRCLE CENTER, X COORDINATE |
| YCIRCL | 1152 | 4434 | CIRCLE CENTER, Y COORDINATE |
| XRADUS | 1154 | 4436 | X RADIUS |
| YRADUS | 1156 | 4438 | Y RADIUS |
| ROTANG | 1158 | 4440 | ROTATION ANGLE |
| ANGBEG | 115C | 4444 | ARC ANGLE START |
| ANGEND | 115E | 4446 | ARC ANGLE END |
| XRCOS | 1160 | 4448 | X RADIUS * COS(ROTATION ANGLE) |
| YRSIN | 1162 | 4450 | Y RADIUS * SIN(ROTATION ANGLE) |
| XRSIN | 1164 | 4452 | X RADIUS * SIN(ROTATION ANGLE) |
| YRCOS | 1166 | 4454 | Y RADIUS * COS(ROTATION ANGLE) |
| | | BASIC GENERAL USE PARAMETERS | |
| XCENTR | 1150 | 4432 | |
| YCENTR | 1152 | 4434 | |
| XDIST1 | 1154 | 4436 | |
| YDIST1 | 1156 | 4438 | |
| XDIST2 | 1158 | 4440 | |
| YDIST2 | 115A | 4442 | |
| DISEEND | 115C | 4444 | PLACEHOLDER |
| COLCNT | 115E | 4446 | CHAR'S COL. COUNTER |
| ROWCNT | 115F | 4447 | |
| STRCNT | 1160 | 4448 | |
| | | | BOX-DRAWING VARIABLES |
| XCORD1 | 1150 | 4432 | POINT 1 X-COORD. |
| YCORD1 | 1152 | 4434 | POINT 1 Y-COORD. |
| BOXANG | 1154 | 4436 | ROTATION ANGLE |
| XCOUNT | 1156 | 4438 | |
| YCOUNT | 1158 | 4440 | |
| BXLENG | 115A | 4442 | LENGTH OF A SIDE |
| XCORD2 | 115C | 4444 | |
| YCORD2 | 115E | 4446 | |
| | | SHAPE AND MOVE-SHAPE VARIABLES | |
| KEYLEN | 1151 | 4433 | |
| KEYNXT | 1152 | 4434 | |
| STRSZ | 1153 | 4435 | STRING LEN |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | SHAPE AND MOVE-SHAPE VARIABLES | |
| GETTYP | 1154 | 4436 | REPLACE SHAPE MODE |
| STRPTR | 1155 | 4437 | STRING POS'N COUNTER |
| OLDBYT | 1156 | 4438 | OLD BIT MAP BYTE |
| NEWBYT | 1157 | 4439 | NEW STRING OR BIT MAP BYTE |
| | 1158 | 4440 | PLACEHOLDER |
| XSIZE | 1159 | 4441 | SHAPE COLUMN LENGTH |
| YSIZE | 115B | 4443 | SHAPE ROW LENGTH |
| XSAVE | 115D | 4445 | TEMP FOR COLUMN LENGTH |
| STRADR | 115F | 4447 | SAVE SHAPE STRING DESCRIPTOR |
| BITIDX | 1161 | 4449 | BIT INDEX INTO BYTE |
| | | BASIC GRAPHIC VARIABLES | |
| CHRPAG | 1168 | 4456 | HIGH BYTE: ADDR OF CHARROM FOR 'CHAR' CMD. |
| BITCNT | 1169 | 4457 | TEMP FOR GSHAPE |
| SCALEM | 116A | 4458 | SCALE MODE FLAG |
| WIDTH | 116B | 4459 | DOUBLE WIDTH FLAG |
| FILFLG | 116C | 4460 | BOX FILL FLAG |
| BITMSK | 116D | 4461 | TEMP FOR BIT MASK |
| NUMCNT | 116E | 4462 | |
| TRCFLG | 116F | 4463 | FLAGS TRACE MODE |
| RENUM_TMP_1 | 1170 | 4464 | A TEMP FOR RENUMBER |
| RENUM_TMP_2 | 1172 | 4466 | A TEMP FOR RENUMBER |
| T3 | 1174 | 4468 | |
| T4 | 1175 | 4469 | |
| VTEMP3 | 1177 | 4471 | GRAPHIC TEMP STORAGE |
| VTEMP4 | 1178 | 4472 | |
| VTEMP5 | 1179 | 4473 | |
| ADRAY1 | 117A | 4474 | PTR TO ROUTINE: CONVERT FLOAT → INTEGER |
| ADRAY2 | 117C | 4476 | PTR TO ROUTINE: CONVERT INTEGER → FLOAT |
| SPRITE_DATA | 117E | 4478 | SPRITE SPEED/DIRECTION TABLES ($117E–D5) |
| VIC_SAVE | 11D6 | 4566 | COPY OF VIC REG'S, USED TO UPDATE CHIP DURING RETRACE (21 BYTES, $11D6–EA) |
| UPPER_LOWER | 11EB | 4587 | POINTER TO UPPER/LOWER CHAR SET FOR CHAR |
| UPPER_GRAPHIC | 11EC | 4588 | PTR. TO UPPER/GRAPHIC CHAR. SET |
| DOSSA | 11ED | 4589 | TEMP STORAGE FOR FILE SA DURING RECORD CMD |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | BASIC GENERAL NON-ZP STORAGE | |
| OLDLIN | 1200 | 4608 | PREVIOUS BASIC LINE NUMBER |
| OLDTXT | 1202 | 4610 | POINTER: BASIC STATEMENT FOR CONTINUE |
| | | PRINT USING DECLARATIONS | |
| PUCHRS | 1204 | 4612 | |
| PUFILL | 1204 | 4612 | PRINT USING FILL SYMBOL |
| PUCOMA | 1205 | 4613 | PRINT USING COMMA SYMBOL |
| PUDOT | 1206 | 4614 | PRINT USING D.P. SYMBOL |
| PUMONY | 1207 | 4615 | PRINT USING MONETARY SYMBOL |
| ERRNUM | 1208 | 4616 | USED BY ERROR TRAPPING ROUTINE-LAST ERR NO |
| ERRLIN | 1209 | 4617 | LINE # OF LAST ERROR—$FFFF IF NO ERROR |
| TRAPNO | 120B | 4619 | LINE TO GO TO ON ERROR. $FFXX IF NONE SET |
| TMPTRP | 120D | 4621 | HOLD TRAP # TEMPOR. |
| ERRTXT | 120E | 4622 | |
| TEXT_TOP | 1210 | 4624 | TOP OF TEXT POINTER |
| MAX_MEM_0 | 1212 | 4626 | HIGHEST ADDRESS AVAILABLE TO BASIC IN RAM 0 |
| TMPTXT | 1214 | 4628 | USED BY DO-LOOP. COULD BE MULT. ASSIGNED |
| TMPLIN | 1216 | 4630 | |
| USRPOK | 1218 | 4632 | |
| RNDX | 121B | 4635 | |
| CIRCLE_SEGMENT | 1220 | 4640 | DEGREES PER CIRCLE SEGMENT |
| DEJAVU | 1221 | 4641 | 'COLD' OR 'WARM' RESET STATUS |
| | | BASIC STORAGE FOR MUSIC VECTORS | |
| TEMPO_RATE | 1222 | 4642 | |
| VOICES | 1223 | 4643 | |
| NTIME | 1229 | 4649 | |
| OCTAVE | 122B | 4651 | |
| SHARP | 122C | 4652 | |
| PITCH | 122D | 4653 | |
| VOICE | 122F | 4655 | |
| WAVE0 | 1230 | 4656 | |
| DNOTE | 1233 | 4659 | |
| FLTSAV | 1234 | 4660 | |
| FLTFLG | 1238 | 4664 | |
| NIBBLE | 1239 | 4665 | |
| TONNUM | 123A | 4666 | |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | **BASIC STORAGE FOR MUSIC VECTORS** | | |
| TONVAL | 123B | 4667 | |
| PARCNT | 123E | 4668 | |
| ATKTAB | 123F | 4669 | |
| SUSTAB | 1249 | 4681 | |
| WAVTAB | 1253 | 4691 | |
| PULSLW | 125D | 4701 | |
| PULSHI | 1267 | 4711 | |
| FILTERS | 1271 | 4721 | |
| | **INTERRUPT VECTORS** | | |
| INT_TRIP_FLAG | 1276 | 4726 | |
| INT_ADR_LO | 1279 | 4729 | |
| INT_ADR_HI | 127C | 4732 | |
| INTVAL | 127F | 4735 | |
| COLTYP | 1280 | 4736 | |
| | **BASIC SOUND COMMAND VARS** | | |
| SOUND_VOICE | 1281 | 4737 | |
| SOUND_TIME_LO | 1282 | 4738 | |
| SOUND_TIME_HI | 1285 | 4741 | |
| SOUND_MAX_LO | 1288 | 4744 | |
| SOUND_MAX_HI | 128B | 4747 | |
| SOUND_MIN_LO | 128E | 4750 | |
| SOUND_MIN_HI | 1291 | 4753 | |
| SOUND_DIRECTION | 1294 | 4756 | |
| SOUND_STEP_LO | 1297 | 4759 | |
| SOUND_STEP_HI | 129A | 4762 | |
| SOUND_FREQ_LO | 129D | 4765 | |
| SOUND_FREQ_HI | 12A0 | 4768 | |
| TEMP_TIME_LO | 12A3 | 4771 | |
| TEMP_TIME_HI | 12A4 | 4772 | |
| TEMP_MAX_LO | 12A5 | 4773 | |
| TEMP_MAX_HI | 12A6 | 4774 | |
| TEMP_MIN_LO | 12A7 | 4775 | |
| TEMP_MIN_HI | 12A8 | 4776 | |
| TEMP_DIRECTION | 12A9 | 4777 | |
| TEMP_STEP_LO | 12AA | 4778 | |
| TEMP_STEP_HI | 12AB | 4779 | |
| TEMP_FREQ_LO | 12AC | 4780 | |
| TEMP_FREQ_HI | 12AD | 4781 | |
| TEMP_PULSE_LO | 12AE | 4782 | |
| TEMP_PULSE_HI | 12AF | 4783 | |
| TEMP_WAVEFORM | 12B0 | 4784 | |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | BASIC SOUND COMMAND VARS | |
| POT_TEMP_1 | 12B1 | 4785 | TEMPORARIES FOR 'POT' FUNCTION |
| POT_TEMP_2 | 12B2 | 4786 | |
| WINDOW_TEMP | 12B3 | 4787 | |
| SAVRAM | 12B7 | 4791 | USED BY SPRDEF & SAVSPR |
| DEFMOD | 12FA | 4858 | USED BY SPRDEF & SAVSPR |
| LINCNT | 12FB | 4859 | USED BY SPRDEF & SAVSPR |
| SPRITE_NUMBER | 12FC | 4860 | USED BY SPRDEF & SAVSPR |
| IRQ_WRAP_FLAG | 12FD | 4861 | USED BY BASIC IRQ TO BLOCK ALL BUT ONE IRQ CALL |
| | 1300 | 4864 | APPLICATION PROGRAM AREA $1300–$1BFF |
| RAMBOT | 1C00 | 7168 | START OF BASIC TEXT $1C00–$EFFF (KERNAL SETS MEMBOT HERE) |
| | | | OR |
| | 1C00 | 7168 | VIDEO MATRIX #2 (1KB OF COLORS FOR BITMAP, IF ALLOCATED) $1C00–$1FFF |
| | 2000 | 8192 | VIC BITMAP (8KB, IF ALLOCATED) $2000–$3FFF |
| | | BEGINNING OF ROM OVER RAM | |
| | 4000 | 16384 | C128 BASIC LO ROM START OF BASIC TEXT IF BIT MAP IS ALLOCATED (RAM) $4000–$EFFF |
| | 8000 | 32768 | C128 BASIC HI ROM (OR FUNCTION ROM) $8000–$BFFF |

## BASIC JUMP TABLE

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | BASIC ENTRY | |
| JMP HARD_RESET | 4000 | 16384 | COLD ENTRY |
| JMP SOFT_RESET | 4003 | 16387 | WARM ENTRY |
| JMP BASIC_IRQ | 4006 | 16390 | IRQ ENTRY |
| | | FORMAT CONVERSIONS | |
| JMP AYINT | AF00 | 44800 | CONVERT F.P. TO INTEGER |
| JMP GIVAYF | AF03 | 44803 | CONVERT INTEGER TO F.P. |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | **FORMAT CONVERSIONS** | |
| JMP FOUT | AF06 | 44806 | CONVERT F.P. TO ASCII STRING |
| JMP VAL_1 | AF09 | 44809 | CONVERT ASCII STRING TO F.P. |
| JMP GETADR | AF0C | 44812 | CONVERT F.P. TO AN ADDRESS |
| JMP FLOATC | AF0F | 44815 | CONVERT ADDRESS TO F.P. |
| | | **MATH FUNCTIONS** | |
| JMP FSUB | AF12 | 44818 | MEM – FACC |
| JMP FSUBT | AF15 | 44821 | ARG – FACC |
| JMP FADD | AF18 | 44824 | MEM + FACC |
| JMP FADDT | AF1B | 44827 | ARG – FACC |
| JMP FMULT | AF1E | 44830 | MEM * FACC |
| JMP FMULTT | AF21 | 44833 | ARG * FACC |
| JMP FDIV | AF24 | 44836 | MEM / FACC |
| JMP FDIVT | AF27 | 44839 | ARG / FACC |
| JMP LOG | AF2A | 44842 | COMPUTE NATURAL LOG OF FACC |
| JMP INT | AF2D | 44845 | PERFORM BASIC INT ON FACC |
| JMP SQR | AF30 | 44848 | COMPUTE SQUARE ROOT OF FACC |
| JMP NEGOP | AF33 | 44851 | NEGATE FACC |
| JMP FPWR | AF36 | 44854 | RAISE ARG TO THE MEM POWER |
| JMP FPWRT | AF39 | 44857 | RAISE ARG TO THE FACC POWER |
| JMP EXP | AF3C | 44860 | COMPUTE EXP OF FACC |
| JMP COS | AF3F | 44863 | COMPUTE COS OF FACC |
| JMP SIN | AF42 | 44866 | COMPUTE SIN OF FACC |
| JMP TAN | AF45 | 44869 | COMPUTE TAN OF FACC |
| JMP ATN | AF48 | 44872 | COMPUTE ATN OF FACC |
| JMP ROUND | AF4B | 44875 | ROUND FACC |
| JMP ABS | AF4E | 44878 | ABSOLUTE VALUE OF FACC |
| JMP SIGN | AF51 | 44881 | TEST SIGN OF FACC |
| JMP FCOMP | AF54 | 44884 | COMPARE FACC WITH MEM |
| JMP RND 0 | AF57 | 44887 | GENERATE RANDOM F.P. NUMBER |
| | | **MOVEMENT** | |
| JMP CONUPK | AF5A | 44890 | MOVE RAM MEM TO ARG |
| JMP ROMUPK | AF5D | 44893 | MOVE ROM MEM TO ARG |
| JMP MOVFRM | AF60 | 44896 | MOVE RAM MEM TO FACC |
| JMP MOVFM | AF63 | 44899 | MOVE ROM MEM TO FACC |
| JMP MOVMF | AF66 | 44902 | MOVE FACC TO MEM |
| JMP MOVFA | AF69 | 44905 | MOVE ARG TO FACC |
| JMP MOVAF | AF6C | 44908 | MOVE FACC TO ARG |
| | | **OTHER BASIC ROUTINES** | |
| JMP OPTAB | AF6F | 44911 | |
| JMP DRAWLN | AF72 | 44914 | |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | OTHER BASIC ROUTINES | |
| JMP GPLOT | AF75 | 44917 | |
| JMP CIRSUB | AF78 | 44920 | |
| JMP RUN | AF7B | 44923 | |
| JMP RUNC | AF7E | 44926 | |
| JMP CLEAR | AF81 | 44929 | |
| JMP NEW | AF84 | 44932 | |
| JMP LNKPRG | AF87 | 44935 | |
| JMP CRUNCH | AF8A | 44938 | |
| JMP FNDLIN | AF8D | 44941 | |
| JMP NEWSTT | AF90 | 44944 | |
| JMP EVAL | AF93 | 44947 | |
| JMP FRMEVL | AF96 | 44950 | |
| JMP RUN_A_ PROGRAM | AF99 | 44953 | |
| JMP SETEXC | AF9C | 44956 | |
| JMP LINGET | AF9F | 44959 | |
| JMP GARBA2 | AFA2 | 44962 | |
| JMP EXECUTE_A_ LINE | AFA5 | 44965 | |
| | | MONITOR ENTRY | |
| JMP CALL | B000 | 45056 | MONITOR CALL ENTRY |
| JMP BREAK | B003 | 45059 | MONITOR BREAK ENTRY |
| JMP MONCMD | B006 | 45062 | MONITOR COMMAND PARSER ENTRY |
| | C000 | 49152 | KERNAL (OR FUNCTION) ROM $C000–$FFFF |

## EDITOR JUMP TABLE

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| JMP CINT | C000 | 49152 | INITIALIZE EDITOR & SCREEN |
| JMP DISPLY | C003 | 49155 | DISPLAY CHARAC IN .A, COLOR IN .X |
| JMP LP2 | C006 | 49158 | GET KEY FROM IRQ BUFFER INTO .A |
| JMP LOOP5 | C009 | 49161 | GET A CHR FROM SCRN LINE INTO.A |
| JMP PRINT | C00C | 49164 | PRINT CHARACTER IN .A |
| JMP SCRORG | C00F | 49167 | GET # OF SCRN ROWS, COLS INTO X&Y |

*Editor Jump Table (continued)*

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| JMP SCNKEY | C012 | 49170 | SCAN KEYBOARD SUBROUTINE |
| JMP REPEAT | C015 | 49173 | HANDLE REPEAT KEY&STORE DECODED KEY |
| JMP PLOT | C018 | 49176 | READ OR SET CRSR POSITION IN .X, .Y |
| JMP CURSOR | C01B | 49179 | MOVE 8563 CURSOR SUBROUTINE |
| JMP ESCAPE | C01E | 49182 | EXECUTE ESC FUNCTION USING CHR IN .A |
| JMP KEYSET | C021 | 49185 | REDEFINE A PROGRAMMABLE FUNC'N KEY |
| JMP IRQ | C024 | 49188 | IRQ ENTRY |
| JMP INIT80 | C027 | 49191 | INITIALIZE 80-COLUMN CHARACTER SET |
| JMP SWAPPER | C02A | 49194 | SWAP EDITOR LOCALS (40/80 CHANGE) |
| JMP WINDOW | C02D | 49197 | SET TOP-LEFT OR BOT-RIGHT OF WINDOW |
| | D000 | 53248 | VIC CHARACTER ROM ($D000–$DFFF) |
| | | | **VIC CHIP REGISTERS** |
| VICREG0 | D000 | 53248 | SPRITE 0, X-LOCATION |
| VICREG1 | D001 | 53249 | SPRITE 0, Y-LOCATION |
| VICREG2 | D002 | 53250 | SPRITE 1, X-LOCATION |
| VICREG3 | D003 | 53251 | SPRITE 1, Y-LOCATION |
| VICREG4 | D004 | 53252 | SPRITE 2, X-LOCATION |
| VICREG5 | D005 | 53253 | SPRITE 2, Y-LOCATION |
| VICREG6 | D006 | 53254 | SPRITE 3, X-LOCATION |
| VICREG7 | D007 | 53255 | SPRITE 3, Y-LOCATION |
| VICREG8 | D008 | 53256 | SPRITE 4, X-LOCATION |
| VICREG9 | D009 | 53257 | SPRITE 4, Y-LOCATION |
| VICREG10 | D00A | 53258 | SPRITE 5, X-LOCATION |
| VICREG11 | D00B | 53259 | SPRITE 5, Y-LOCATION |
| VICREG12 | D00C | 53260 | SPRITE 6, X-LOCATION |
| VICREG13 | D00D | 53261 | SPRITE 6, Y-LOCATION |
| VICREG14 | D00E | 53262 | SPRITE 7, X-LOCATION |
| VICREG15 | D00F | 53263 | SPRITE 7, Y-LOCATION |
| VICREG16 | D010 | 53264 | MSBIT OF X-LOCATION FOR SPRITES 0–7 |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | | **VIC CHIP REGISTERS** |
| VICREG17 | D011 | 53265 | VIC CONTROL REGISTER 1 |
| | | | 7 — RASTER COMPARE BIT<br>6 — EXTENDED COLOR TEXT MODE (1 = ON)<br>5 — BIT MAP MODE (1 = ENABLE)<br>4 — BLANK SCREEN TO BORDER CLR (0 = BLANK)<br>3 — SELECT 24/25 ROW TEXT DISPLAY (1 = 25)<br>2–0 — SMOOTH SCROLL TO Y DOT POSITION |
| VICREG18 | D012 | 53266 | READ/WRITE RASTER VALUE FOR COMPARE IRQ |
| VICREG19 | D013 | 53267 | LIGHT PEN LATCH X-POSITION |
| VICREG20 | D014 | 53268 | LIGHT PEN LATCH Y-POSITION |
| VICREG21 | D015 | 53269 | SPRITES 0–7 DISPLAY ENABLE (1 = ENABLE) |
| VICREG22 | D016 | 53270 | VIC CONTROL REGISTER 2 BITS |
| | | | 7–6 — UNUSED<br>5 — RESET<br>4 — MULTI-COLOR MODE (1 = ENABLE)<br>3 — SELECT 38/40 COLUMN DISPLAY (1 = 40 COLS)<br>2–0 — SMOOTH SCROLL TO X-POSITION |
| VICREG23 | D017 | 53271 | SPRITES 0–7 Y EXPAND |
| VICREG24 | D018 | 53272 | VIC MEMORY CONTROL REGISTER BITS ($D018) |
| | | | 7–4 — VIDEO MATRIX BASE ADDRESS<br>3–0 — CHARACTER DOT-DATA BASE ADDRESS |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | | **VIC CHIP REGISTERS** |
| VICREG25 | D019 | 53273 | VIC INTERRUPT FLAG REGISTER (1 = IRQ OCCURRED)<br>7 — SET ON ANY ENABLED VIC IRQ CONDITION<br>6–4 — NOT USED<br>3 — LIGHT PEN TRIGGERED IRQ FLAG<br>2 — SPRITE TO SPRITE COLLISION IRQ FLAG<br>1 — SPRITE TO BACKGROUND COLLISION IRQ FLAG<br>0 — RASTER COMPARE IRQ FLAG |
| VICREG26 | D01A | 53274 | IRQ ENABLE (1 = ENABLED) BITS<br>7–4 — NOT USED<br>3 — LIGHT PEN<br>2 — SPRITE TO SPRITE<br>1 — SPRITE TO BACKGROUND<br>0 — IRQ |
| VICREG27 | D01B | 53275 | SPRITES 0–7 BACKGROUND PRIORITY (1 = SPRITE) |
| VICREG28 | D01C | 53276 | SPRITES 0–7 MULTI-COLOR MODE (1 = MULTI-COLOR) |
| VICREG29 | D01D | 53277 | SPRITES 0–7 X EXPAND |
| VICREG30 | D01E | 53278 | SPRITE TO SPRITE COLLISION LATCH |
| VICREG31 | D01F | 53279 | SPRITE TO BACKGROUND COLLISION LATCH |
| VICREG32 | D020 | 53280 | BORDER COLOR |
| VICREG33 | D021 | 53281 | BACKGROUND COLOR 0 |
| VICREG34 | D022 | 53282 | BACKGROUND COLOR 1 |
| VICREG35 | D023 | 53283 | BACKGROUND COLOR 2 |
| VICREG36 | D024 | 53284 | BACKGROUND COLOR 3 |
| VICREG37 | D025 | 53285 | SPRITE MULTI-COLOR REGISTER 0 |
| VICREG38 | D026 | 53286 | SPRITE MULTI-COLOR REGISTER 1 |
| VICREG39 | D027 | 53287 | SPRITE 0 COLOR |
| VICREG40 | D028 | 53288 | SPRITE 1 COLOR |
| VICREG41 | D029 | 53289 | SPRITE 2 COLOR |
| VICREG42 | D02A | 53290 | SPRITE 3 COLOR |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | | **VIC CHIP REGISTERS** |
| VICREG43 | D02B | 53291 | SPRITE 4 COLOR |
| VICREG44 | D02C | 53292 | SPRITE 5 COLOR |
| VICREG45 | D02D | 53293 | SPRITE 6 COLOR |
| VICREG46 | D02E | 53294 | SPRITE 7 COLOR |
| VICREG47 | D02F | 53295 | KEYBOARD LINES<br>BITS<br>7–3 NOT USED<br>2–0 K2, K1 AND K0 |
| VICREG48 | D030 | 53296 | CLOCK SPEED<br>BITS<br>7–2 NOT USED<br>1 TEST<br>0 2 MHZ |
| | | | **SID REGISTERS** |
| SIDREG0 | D400 | 54272 | VOICE 1 FREQUENCY LO |
| SIDREG1 | D401 | 54273 | VOICE 1 FREQUENCY HI |
| SIDREG2 | D402 | 54274 | VOICE 1 PULSE WIDTH LO |
| SIDREG3 | D403 | 54275 | VOICE 1 PULSE WIDTH HI (0–15) |
| SIDREG4 | D404 | 54276 | VOICE 1 CONTROL REGISTER<br>7 NOISE (1 = NOISE)<br>6 PULSE (1 = PULSE)<br>5 SAW (1 = SAWTOOTH)<br>4 TRI (1 = TRIANGLE)<br>3 TEST (1 = DISABLE OSCILLATOR)<br>2 RING (1 = RING MODULATE OSC 1 WITH OSC 3 OUTPUT)<br>1 SYNC (1 = SYNCHRONIZE OSC 1 WITH OSC 3 FREQ)<br>0 GATE (1 = START ATTACK/DECAY/SUSTAIN 0 = START RELEASE) |
| SIDREG5 | D405 | 54277 | VOICE 1 ATTACK/DECAY<br>7–4 ATTACK (0–15)<br>3–0 DECAY (0–15) |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | | **SID REGISTERS** |
| SIDREG6 | D406 | 54278 | VOICE 1 SUSTAIN/RELEASE |
| | | | 7–4 SUSTAIN (0–15)<br>3–0 RELEASE (0–15) |
| SIDREG7 | D407 | 54279 | VOICE 2 FREQUENCY LO |
| SIDREG8 | D408 | 54280 | VOICE 2 FREQUENCY HI |
| SIDREG9 | D409 | 54281 | VOICE 2 PULSE WIDTH LO |
| SIDREG10 | D40A | 54282 | VOICE 2 PULSE WIDTH HI (0–15) |
| SIDREG11 | D40B | 54283 | VOICE 2 CONTROL REGISTER |
| | | | 7 NOISE (1 = NOISE)<br>6 PULSE (1 = PULSE)<br>5 SAW (1 = SAWTOOTH)<br>4 TRI (1 = TRIANGLE)<br>3 TEST (1 = DISABLE OSCILLATOR)<br>2 RING (1 = RING MODULATE OSC 2 WITH OSC 1 OUTPUT)<br>1 SYNC (1 = SYNCHRONIZE OSC 2 WITH OSC 1 FREQ)<br>0 GATE (1 = START ATTACK/DECAY/SUSTAIN 0 = START RELEASE) |
| SIDREG12 | D40C | 54284 | VOICE 2 ATTACK/DECAY |
| | | | 7–4 ATTACK (0–15)<br>3–0 DECAY (0–15) |
| SIDREG13 | D40D | 54285 | VOICE 2 SUSTAIN/RELEASE |
| | | | 7–4 SUSTAIN (0–15)<br>3–0 RELEASE (0–15) |
| SIDREG14 | D40E | 54286 | VOICE 3 FREQUENCY LO |
| SIDREG15 | D40F | 54287 | VOICE 3 FREQUENCY HI |
| SIDREG16 | D410 | 54288 | VOICE 3 PULSE WIDTH LO |
| SIDREG17 | D411 | 54289 | VOICE 3 PULSE WIDTH HI (0–15) |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | | **SID REGISTERS** |
| SIDREG18 | D412 | 54290 | VOICE 3 CONTROL REGISTER |
| | | | 7 NOISE (1 = NOISE)<br>6 PULSE (1 = PULSE)<br>5 SAW (1 = SAWTOOTH)<br>4 TRI (1 = TRIANGLE)<br>3 TEST (1 = DISABLE OSCILLATOR)<br>2 RING (1 = RING MODULATE OSC 3 WITH OSC 2 OUTPUT)<br>1 SYNC (1 = SYNCHRONIZE OSC 3 WITH OSC 2 FREQ)<br>0 GATE (1 = START ATTACK/DECAY/SUSTAIN 0 = START RELEASE) |
| SIDREG19 | D413 | 54291 | VOICE 3 ATTACK/DECAY |
| | | | 7–4 ATTACK (0–15)<br>3–0 DECAY (0–15) |
| SIDREG20 | D414 | 54292 | VOICE 3 SUSTAIN/RELEASE |
| | | | 7–4 SUSTAIN (0–15)<br>3–0 RELEASE (0–15) |
| SIDREG21 | D415 | 54293 | FILTER CUTOFF FREQUENCY LO |
| SIDREG22 | D416 | 54294 | FILTER CUTOFF FREQUENCY HI |
| SIDREG23 | D417 | 54295 | RESONANCE/FILTER |
| | | | 7–4 FILTER RESONANCE (0–15)<br>3 FILTER EXTERNAL INPUT (1 = YES)<br>2 FILTER VOICE 3 OUTPUT (1 = YES)<br>1 FILTER VOICE 2 OUTPUT (1 = YES)<br>0 FILTER VOICE 1 OUTPUT (1 = YES) |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | | **SID REGISTERS** |
| SIDREG24 | D418 | 54296 | MODE/VOLUME |
| | | | 7 — CUTOFF VOICE 3 OUTPUT (1 = OFF)<br>6 — SELECT HI-PASS FILTER (1 = ON)<br>5 — SELECT BAND-PASS FILTER (1 = ON)<br>4 — SELECT LO-PASS FILTER (1 = ON)<br>3–0 — OUTPUT VOLUME (0–15) |
| SIDREG25 | D419 | 54297 | POT X, A/D CONVERTER, PADDLE 1 |
| SIDREG26 | D41A | 54298 | POT Y, A/D CONVERTER, PADDLE 2 |
| SIDREG27 | D41B | 54299 | OSCILLATOR 3, RANDOM NUMBER GENERATOR |
| SIDREG28 | D41C | 54300 | ENVELOPE GENERATOR 3 OUTPUT |
| | | | **C128 MEMORY MANAGEMENT UNIT, PRIMARY REGISTERS IMPLEMENTS C128, C64, & CP/M 3.0 MODES** |
| MMUCR1 | D500 | 54528 | CONFIGURATION REGISTER |
| PCRA | D501 | 54529 | PRECONFIGURATION REGISTER A |
| PCRB | D502 | 54530 | PRECONFIGURATION REGISTER B |
| PCRC | D503 | 54531 | PRECONFIGURATION REGISTER C |
| PCRD | D504 | 54532 | PRECONFIGURATION REGISTER D |
| | | | BITS ($D500–$D504)<br>7–6 — RAM BANK (0–3)<br>5–4 — ROM HI (SYSTEM, INT, EXT, RAM)<br>3–2 — ROM MID (SYSTEM, INT, EXT, RAM)<br>1 — ROM LO (SYSTEM, RAM)<br>0 — I/O (I/O BLOCK, ELSE ROM-HI) |
| MMUMCR | D505 | 54533 | MODE CONFIGURATION REGISTER |
| | | | 7 — 40/80 KEY SENSE<br>6 — OS MODE, 0 = C128/1 = C64<br>5 — /EXROM LINE SENSE<br>4 — /GAME LINE SENSE<br>3 — FSDIR<br>2–1 — NOT USED<br>0 — PROCESSOR, 0 = Z80/1 = 8502 |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | | **C128 MEMORY MANAGEMENT UNIT, PRIMARY REGISTERS IMPLEMENTS C128, C64, & CP/M 3.0 MODES** |
| MMURCR | D506 | 54534 | RAM CONFIGURATION REGISTER |
| | | | 7–6 VIC RAM BANK (VA17 & VA16)<br>5–4 RAM BLOCK (FOR FUTURE EXPANSION)<br>3–2 RAM SHARE STATUS (NONE, BOT, TOP, BOTH)<br>1–0 RAM SHARE AMOUNT (1K, 4K, 8K, 16K) |
| MMUP0L | D507 | 54535 | PAGE 0 POINTER LOW |
| MMUP0H | D508 | 54536 | PAGE 0 POINTER HIGH |
| MMUP1L | D509 | 54537 | PAGE 1 POINTER LOW |
| MMUP1H | D50A | 54538 | PAGE 1 POINTER HIGH |
| | | | SWAPS PAGE0 AND/OR PAGE1 WITH ANY OTHER PAGE IN THE 256K ADDRESS SPACE |
| | | | BITS ($D508 & $D50A)<br>7–4 —<br>3–2 A19–A18 (USED IN 1MB SYSTEM)<br>1–0 A17–A16 (256K SYSTEM) |
| | | | BITS ($D507 & $D509)<br>7–0 A15–A8 |
| MMUVER | D50B | 54539 | MMU VERSION NUMBER |
| | | | 7–4 BANK VERSION<br>3–0 MMU VERSION |
| | | | **C128 80-COLUMN VIDEO DISPLAY CONTROLLER** |
| VDCADR | D600 | 54784 | 8563 ADDRESS REGISTER |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | | ADDRESS AND DATA REGISTER BITS |
| | 7 6 | 5 | 4 3 2 1 0 |
| D600——WRITE | — — | R5 | R4 R3 R2 R1 R0 |
| READ | STATUS LP | VBLANK | — — — — — |
| VDCDAT | D601 | 54785 | 8563 DATA REGISTER |
| D601——DATA | D7 D6 | D5 | D4 D3 D2 D1 D0 |
| | | | ADDITIONAL 8563 REGISTERS ARE NOT VISIBLE TO THE 8502 |
| | D700 | 55040 | RESERVED I/O BLOCK |
| VICCOL | D800 | 55296 | VIC COLOR MATRIX, 1 KB ($D800–$DBFF) |
| | | | 6526 CIA#1, COMPLEX INTERFACE ADAPTER #1 KEYBOARD, JOYSTICK, PADDLES, LIGHTPEN, FAST DISK |
| D1PRA | DC00 | 56320 | PORT A (OUTPUT KEYBOARD COLUMNS) |

BITS ($DC00)

| | |
|---|---|
| 0 | PRA0 : KEYBD O/P C0/JOY #1 DIRECTION |
| 1 | PRA1 : KEYBD O/P C1/JOY #1 DIRECTION |
| 2 | PRA2 : KEYBD O/P C2/JOY #1 DIRECTION/PADDLE FIRE BUTTON |
| 3 | PRA3 : KEYBD O/P C3/JOY #1 DIRECTION/PADDLE FIRE BUTTON |
| 4 | PRA4 : KEYBD O/P C4/JOY #1 FIRE BUTTON |
| 5 | PRA5 : KEYBD O/P C5/ |
| 6 | PRA6 : KEYBD O/P C6/ /SELECT PORT #1 PADDLES |
| 7 | PRA7 : KEYBD O/P C7/ /SELECT PORT #2 PADDLES |

| | | | |
|---|---|---|---|
| D1PRB | DC01 | 56321 | PORT B (INPUT KEYBOARD ROWS) |

BITS ($DC01)

| | |
|---|---|
| 0 | PRB0 : KEYBD I/P R0/JOY #2 DIRECTION |
| 1 | PRB1 : KEYBD I/P R1/JOY #2 DIRECTION/PADDLE FIRE BUTTON |
| 2 | PRB2 : KEYBD I/P R2/JOY #2 DIRECTION/PADDLE FIRE BUTTON |
| 3 | PRB3 : KEYBD I/P R3/JOY #2 DIRECTION/ |
| 4 | PRB4 : KEYBD I/P R4/JOY #2 FIRE BUTTON |
| 5 | PRB5 : KEYBD I/P R5/ |
| 6 | PRB6 : KEYBD I/P R6/TIMER B: TOGGLE /PULSE OUTPUT |
| 7 | PRB7 : KEYBD I/P R7/TIMER A: TOGGLE /PULSE OUTPUT |

| | | | |
|---|---|---|---|
| D1DDRA | DC02 | 56322 | DATA DIRECTION PORT A |
| D1DDRB | DC03 | 56323 | DATA DIRECTION PORT B |
| D1T1L | DC04 | 56324 | TA LO |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | | **6526 CIA#1, COMPLEX INTERFACE ADAPTER #1<br>KEYBOARD, JOYSTICK, PADDLES, LIGHTPEN, FAST DISK** |
| D1T1H | DC05 | 56325 | TA HI (TIMER A) |
| D1T2L | DC06 | 56326 | TB LO |
| D1T2H | DC07 | 56327 | TB HI (TIMER B) |
| D1TOD1 | DC08 | 56328 | TOD (TENTHS) |
| D1TODS | DC09 | 56329 | TOD (SECONDS) |
| D1TODM | DC0A | 56330 | TOD (MINUTES) |
| D1TODH | DC0B | 56331 | TOD (HOURS) |
| D1SDR | DC0C | 56332 | SERIAL DATA REGISTER |
| D1ICR | DC0D | 56333 | INTERRUPT CONTROL REGISTER |
| D1CRA | DC0E | 56334 | CONTROL REGISTER A |
| D1CRB | DC0F | 56335 | CONTROL REGISTER B |
| | | | **6526 CIA#2 COMPLEX INTERFACE ADAPTER #2<br>USER PORT,RS-232,SERIAL BUS,VIC MEMORY,NMI** |
| D2PRA | DD00 | 56576 | PORT A, SERIAL BUS, RS-232, VA14 & VA15<br>BITS<br>0 PRA0 : VA14<br>1 PRA1 : VA15<br>2 PRA2 : RS232 DATA OUTPUT<br>3 PRA3 : SERIAL ATN OUTPUT<br>4 PRA4 : SERIAL CLK OUTPUT<br>5 PRA5 : SERIAL DATA OUTPUT<br>6 PRA6 : SERIAL CLK INPUT<br>7 PRA7 : SERIAL DATA INPUT |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| D2PRB | DD01 | 56577 | PORT B, USER PORT, RS-232<br>BITS<br>0 PRB0 : USER PORT / RS-232 RECEIVED DATA<br>1 PRB1 : USER PORT / RS-232 REQUEST TO SEND<br>2 PRB2 : USER PORT / RS-232 DATA TERMINAL READY<br>3 PRB3 : USER PORT / RS-232 RING INDICATOR<br>4 PRB4 : USER PORT / RS-232 CARRIER DETECT<br>5 PRB5 : USER PORT<br>6 PRB6 : USER PORT / RS-232 CLEAR TO SEND<br>7 PRB7 : USER PORT / RS-232 DATA SET READY |
| D2DDRA | DD02 | 56578 | DATA DIRECTION PORT A |
| D2DDRB | DD03 | 56579 | DATA DIRECTION PORT B |
| D2T1L | DD04 | 56580 | TA LO |
| D2T1H | DD05 | 56581 | TA HI (TIMER A) |
| D2T2L | DD06 | 56582 | TB LO |
| D2T2H | DD07 | 56583 | TB HI (TIMER B) |
| D2TOD1 | DD08 | 56584 | TOD (TENTHS) |
| D2TODS | DD09 | 56585 | TOD (SECONDS) |
| D2TODM | DD0A | 56586 | TOD (MINUTES) |
| D2TODH | DD0B | 56587 | TOD (HOURS) |
| D2SDR | DD0C | 56588 | SERIAL DATA REGISTER |
| D2ICR | DD0D | 56589 | INTERRUPT CONTROL REGISTERS (NMI'S) |
| D2CRA | DD0E | 56590 | CONTROL REGISTER A |
| D2CRB | DD0F | 56591 | CONTROL REGISTER B |
| IO1 | DE00 | 56832 | EXPANSION I/O SLOT (RESERVED) |
| IO2 | DF00 | 57088 | EXPANSION I/O SLOT (RESERVED) |
| | | | C128 DMA CONTROLLER FOR EXPANSION RAM ACCESS OPTIONAL DEVICE MAPPED INTO IO2 BLOCK VIA SYSTEM EXPANSION PORT (PRELIMINARY) |
| DMA ST | DF00 | 57088 | DMA CONTROLLER STATUS REGISTER (READ ONLY) |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | | 7 INTERRUPT PENDING (1 = INT. WAITING TO BE SERVICED)<br>6 END OF BLOCK (1 = TRANSFER COMPLETE)<br>5 FAULT (1 = BLOCK VERIFY ERROR)<br>4 SIZE (0 = EXP. MEMORY = 128K) (1 = EXP. MEMORY = 512K)<br>3–0 VERSION |
| DMA CMD | DF01 | 57089 | DMA CONTROLLER COMMAND REGISTER<br>7 EXECUTE<br>6 RESERVED<br>5 LOAD (1 = ENABLE AUTO = LOAD)<br>4 $FF00 (1 = DISABLE $FF00 DECADES)<br>3 RESERVED<br>2 RESERVED<br>1–0 MODE (00 = TRANSFER FROM INTERNAL TO EXTERNAL, 01 = FROM EXT TO INT, 10 = SWAP, 11 = VERIFY) |
| DMA ADL | DF02 | 57090 | LSB OF INTERNAL (C128) ADDRESS TO ACCESS |
| DMA ADH | DF03 | 57091 | MSB OF INTERNAL (C128) ADDRESS TO ACCESS |
| DMA LO | DF04 | 57092 | LSB OF EXTERNAL EXPANSION RAM TO ACCESS |
| DMA HI | DF05 | 57093 | MSB OF EXTERNAL EXPANSION RAM TO ACCESS |
| DMA BNK | DF06 | 57094 | 64K EXTERNAL RAM BANK BITS ($DF06) |

| MEMORY ADDRESS LABEL | HEXADECIMAL ADDRESS | DECIMAL ADDRESS | DESCRIPTION |
|---|---|---|---|
| | | | 7–3 NOT USED<br>2–0 EXPANSION BANK NUMBER |
| DMA DAL | DF07 | 57095 | LSB OF BYTE COUNT |
| DMA DAH | DF08 | 57096 | MSB OF BYTE COUNT (BLOCK COUNT) |
| DMA SUM | DF09 | 57097 | INTERRUPT MASK REGISTER<br>7 INTERRUPT ENABLE (1 = INTERRUPTS ENABLED)<br>6 END OF BLOCK MASK (1 = INTERRUPT ON END OF BLOCK)<br>5 VERIFY ERROR (1 = INTERRUPT ON VERIFY ERROR) |
| DMA VER | DF0A | 57098 | ADDRESS CONTROL REGISTER<br>BITS 7 AND 6<br>0,0 = INCREMENT BOTH ADDRESS (DEFAULT)<br>0,1 = FIX EXPANSION ADDRESS.<br>1,0 = FIX C128 ADDRESS.<br>1,1 = FIX BOTH ADDRESSES. |
| | E000 | 57344 | KERNAL ROM (8K OPERATING SYSTEM, $E000–$FFFF) |
| | | | MMU SECONDARY REGISTERS |
| MMUCR | FF00 | 65280 | CONFIGURATION REGISTER (SECONDARY) |
| LCRA | FF01 | 65281 | LOAD CONFIGURATION REGISTER A |
| LCRB | FF02 | 65282 | LOAD CONFIGURATION REGISTER B |
| LCRC | FF03 | 65283 | LOAD CONFIGURATION REGISTER C |
| LCRD | FF04 | 65284 | LOAD CONFIGURATION REGISTER D<br>BITS ($FF00–$FF04)<br>7–6 RAM BANK (0–3)<br>5–4 ROM HI (SYSTEM, INT, EXT, RAM)<br>3–2 ROM MID (SYSTEM, INT, EXT, RAM)<br>1 ROM LO (SYSTEM, RAM)<br>0 I/O (I/O ELSE ROM-HI) |

# KERNAL JUMP TABLE

## NEW ENTRIES FOR C128

| | | | |
|---|---|---|---|
| JMP SPIN SPOUT | FF47 | 65351 | SET UP FAST SERIAL PORT FOR I/O |
| JMP CLOSE ALL | FF4A | 65354 | CLOSE ALL LOGICAL FILES FOR A DEVICE |
| JMP C64MODE | FF4D | 65357 | RECONFIGURE SYSTEM AS A C64 (NO RETURN) |
| JMP DMA CALL | FF50 | 65360 | INITIATE DMA REQUEST TO EXTERNAL RAM EXPANSION, SEND COMMAND TO DMA DEVICE |
| JMP BOOT CALL | FF53 | 65363 | BOOT LOAD PROGRAM FROM DISK |
| JMP PHOENIX | FF56 | 65366 | CALL ALL FUNCTION CARDS' COLD START ROUTINES, INITIALIZE |
| JMP LKUPLA | FF59 | 65369 | SEARCH TABLES FOR GIVEN LA |
| JMP LKUPSA | FF5C | 65372 | SEARCH TABLES FOR GIVEN SA |
| JMP SWAPPER | FF5F | 65375 | SWITCH BETWEEN 40 AND 80 COLUMNS (EDITOR) |
| JMP DLCHR | FF62 | 65378 | INIT 80-COL CHARACTER RAM (EDITOR) |
| JMP PFKEY | FF65 | 65381 | PROGRAM FUNCTION KEY (EDITOR) |
| JMP SETBNK | FF68 | 65384 | SET BANK FOR I/O OPERATIONS |
| JMP GETCFG | FF6B | 65387 | LOOKUP MMU DATA FOR GIVEN BANK |
| JMP JSRFAR | FF6E | 65390 | JSR TO ANY BANK, RTS TO CALLING BANK |
| JMP JMPFAR | FF71 | 65393 | JMP TO ANY BANK |
| JMP INDFET | FF74 | 65396 | LDA (FETVEC), Y FROM ANY BANK |
| JMP INDSTA | FF77 | 65499 | STA (STAVEC), Y TO ANY BANK |
| JMP INDCMP | FF7A | 65402 | CMP (CMPVEC),Y TO ANY BANK |
| JMP PRIMM | FF7D | 65405 | PRINT IMMEDIATE UTILITY (ALWAYS JSR TO THIS ROUTINE) |

STANDARD KERNAL JUMP TABLE

| | | | |
|---|---|---|---|
| | FF80 | 65408 | RELEASE NUMBER OF KERNAL |
| JMP CINT | FF81 | 65409 | INIT EDITOR & DISPLAY CHIPS (EDITOR) |

## STANDARD KERNAL JUMP TABLE

| | | | | |
|---|---|---|---|---|
| JMP IOINIT | FF84 | | 65412 | INIT I/O DEVICES (PORTS, TIMERS, ETC.) |
| JMP RAMTAS | FF87 | | 65415 | INITIALIZE RAM AND BUFFERS FOR SYSTEM |
| JMP RESTOR | FF8A | | 65418 | RESTORE VECTORS TO INITIAL SYSTEM |
| JMP VECTOR | FF8D | | 65421 | CHANGE VECTORS FOR USER |
| JMP SETMSG | FF90 | | 65424 | CONTROL O.S. MESSAGE |
| JMP SECND | FF93 | | 65427 | SEND SA AFTER LISTEN |
| JMP TKSA | FF96 | | 65430 | SEND SA AFTER TALK |
| JMP MEMTOP | FF99 | | 65433 | SET/READ TOP OF SYSTEM RAM |
| JMP MEMBOT | FF9C | | 65436 | SET/READ BOTTOM OF SYSTEM RAM |
| JMP KEY | FF9F | | 65439 | SCAN KEYBOARD (EDITOR) |
| JMP SETTMO | FFA2 | | 65442 | SET TIMEOUT IN IEEE (RESERVED) |
| JMP ACPTR | FFA5 | | 65445 | HANDSHAKE SERIAL BYTE IN |
| JMP CIOUT | FFA8 | | 65448 | HANDSHAKE SERIAL BYTE OUT |
| JMP UNTLK | FFAB | | 65451 | SEND UNTALK OUT SERIAL |
| JMP UNLSN | FFAE | | 65454 | SEND UNLISTEN OUT SERIAL |
| JMP LISTN | FFB1 | | 65457 | SEND LISTEN OUT SERIAL |
| JMP TALK | FFB4 | | 65460 | SEND TALK OUT SERIAL |
| JMP READSS | FFB7 | | 65463 | RETURN I/O STATUS BYTE |
| JMP SETLFS | FFBA | | 65460 | SET LA, FA, SA |
| JMP SETNAM | FFBD | | 65469 | SET LENGTH AND FILE NAME ADDRESS |
| JMP (IOPEN) | FFC0 | OPEN | 65472 | OPEN LOGICAL FILE |
| JMP (ICLOSE) | FFC3 | CLOSE | 65475 | CLOSE LOGICAL FILE |
| JMP (ICHKIN) | FFC6 | CHKIN | 65478 | SET CHANNEL IN |
| JMP (ICKOUT) | FFC9 | CKOUT | 65481 | SET CHANNEL OUT |
| JMP (ICLRCH) | FFCC | CLRCH | 65484 | RESTORE DEFAULT I/O CHANNEL |
| JMP (IBASIN) | FFCF | BASIN | 65487 | INPUT FROM CHANNEL |
| JMP (IBSOUT) | FFD2 | BSOUT | 65490 | OUTPUT TO CHANNEL |
| JMP LOADSP | FFD5 | | 65493 | LOAD FROM FILE |
| JMP SAVESP | FFD8 | | 65496 | SAVE TO FILE |
| JMP SETTIM | FFDB | | 65599 | SET INTERNAL CLOCK |
| JMP RDTIM | FFDE | | 65502 | READ INTERNAL CLOCK |
| JMP (ISTOP) | FFE1 | STOP | 65505 | SCAN STOP KEY |
| JMP (IGETIN) | FFE4 | GETIN | 65508 | READ BUFFERED DATA |
| JMP (ICLALL) | FFE7 | CLALL | 65511 | CLOSE ALL FILES AND CHANNELS |

STANDARD KERNAL JUMP TABLE

| | | | | |
|---|---|---|---|---|
| JMP UDTIM | FFEA | CLOCK | 65514 | INCREMENT INTERNAL CLOCK |
| JMP SCRORG | FFED | | 65517 | RETURN SCREEN WINDOW SIZE (EDITOR) |
| JMP PLOT | FFF0 | | 65520 | READ/SET X,Y CURSOR COORD (EDITOR) |
| JMP IOBASE | FFF3 | | 65523 | RETURN I/O BASE |
| SYSTEM | FFF8 | | 65528 | OPERATING SYSTEM VECTOR (RAM1) |
| NMI | FFFA | | 65530 | PROCESSOR NMI VECTOR |
| RESET | FFFC | | 65532 | PROCESSOR RESET VECTOR |
| IRQ | FFFE | | 65534 | PROCESSOR IRQ/BRK VECTOR |

# KERNAL/EDITOR FLAGS AND SHADOW REGISTERS

The following symbols are used by the C128 Editor. Note that the Editor IRQ VIC screen handler depends upon them. In most cases the contents of these locations will be placed directly into the appropriate register and should be used instead of the actual register. For example, to change the location of the character set used by VIC, use VM1 ($0A2C) instead of VIC register 24 ($D018). VM1 will be used by the editor to update VIC register 24.

| ADDRESS/NAME | EXPLANATION |
|---|---|
| $00D8/GRAPHM | SEE BELOW. IF = $FF THEN EDITOR LEAVES VIC ALONE. |
| $00D9/CHAREN | MASK FOR 8502 /CHAREN BIT. |
| $0A2C/VM1 | VIC TEXT MODE VIDEO MATRIX & CHARACTER BASE POINTER. |
| $0A2D/VM2 | VIC GRAPHIC MODE VIDEO MATRIX & BIT MAP POINTER. |
| $0A2E/VM3 | 8563 TEXT DISPLAY BASE ADDRESS. |
| $0A2F/VM4 | 8563 ATTRIBUTE BASE ADDRESS. |
| $0A34/SPLIT | IN SPLIT SCREEN MODE, CONTAINS VALUE FOR MIDDLE RASTER IRQ. |
| $0A2B/CURMOD | 8563 CURSOR MODE. |
| $0A21/PAUSE | CONTROL S FLAG (IN EFFECT = $13) |

### EXPLANATION OF VARIOUS KERNAL/EDITOR FLAG BYTES, ETC.

| ADDRESS | SYMBOL | DESCRIPTION | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0000 | D6510 | — | (IN) | (OUT) | (IN) | (OUT) | (OUT) | (OUT) | (OUT) |
| 0001 | R6510 | — | CAPKEY | CASMTR | CASSEN | CASWRT | CHAREN | HIRAM | LORAM |
| 00F7 | LOCKS | CASE | CTL S | — | — | — | — | — | — |
| 00F8 | SCROLL | OFF | LINKER | | | | | | |
| 00D3 | SHFLAG | — | — | — | ALT | ALPHA | CTRL | | SHIFT |
| 0A22 | RPTFLG | ALL | NONE | — | — | — | — | — | — |
| 0A26 | BLNON | ON | BLNK | — | — | — | — | — | — |
| 00F9 | BEEPER | ON | — | — | — | — | — | — | — |
| 00D8 | GRAPHM | MCM | SPLIT | BMM | — | — | — | — | — |
| 00D7 | MODE | 40/80 | — | — | — | — | — | — | — |
| 0A04 | INIT_ STATUS | CHRSET | CINT | — | — | — | — | — | BASIC |

**Notes on Kernal Symbols:**

**Init_Status.** See also above. Lets system know what has been initialized and what hasn't. Set to $00 by a reset but untouched by NMI.

**System_Vector.** Where the Kernal goes when it has to go somewhere. It's set to BASIC cold at reset. BASIC itself sets it to BASIC warm after it has initialized. The monitor respects it too.

**System.** Vector in RAM1 at $FFF8. Set at power-up to C128MODE, user may redirect it to his code. Taken at reset always providing user with control (protection) from reset.

# COMMODORE 64 MEMORY MAP

| LABEL | HEX ADDRESS | DECIMAL LOCATION | DESCRIPTION |
|---|---|---|---|
| D6510 | 0000 | 0 | 6510 On-Chip Data-Direction Register |
| R6510 | 0001 | 1 | 6510 On-Chip 8-Bit Input/Output Register |
| | 0002 | 2 | Unused |
| ADRAY1 | 0003–0004 | 3–4 | Jump Vector: Convert Floating—Integer |
| ADRAY2 | 0005–0006 | 5–6 | Jump Vector: Convert Integer—Floating |
| CHARAC | 0007 | 7 | Search Character |
| ENDCHR | 0008 | 8 | Flag: Scan for Quote at End of String |

| LABEL | HEX ADDRESS | DECIMAL LOCATION | DESCRIPTION |
|---|---|---|---|
| TRMPOS | 0009 | 9 | Screen Column From Last TAB |
| VERCK | 000A | 10 | Flag: 0 = Load, 1 = Verify |
| COUNT | 000B | 11 | Input Buffer Pointer/No. of Subscripts |
| DIMFLG | 000C | 12 | Flag: Default Array DIMension |
| VALTYP | 000D | 13 | Data Type: $FF = String, $00 = Numeric |
| INTFLG | 000E | 14 | Data Type: $80 = Integer, $00 = Floating |
| GARBFL | 000F | 15 | Flag: DATA scan/LIST quote/ Garbage Coll |
| SUBFLG | 0010 | 16 | Flag: Subscript Ref / User Function Call |
| INPFLG | 0011 | 17 | Flag: $00 = INPUT, $40 = GET, $98 = READ |
| TANSGN | 0012 | 18 | Flag: TAN sign / Comparison Result |
| | 0013 | 19 | Flag: INPUT Prompt |
| LINNUM | 0014–0015 | 20–21 | Temp: Integer Value |
| TEMPPT | 0016 | 22 | Pointer: Temporary String Stack |
| LASTPT | 0017–0018 | 23–24 | Last Temp String Address |
| TEMPST | 0019–0021 | 25–33 | Stack for Temporary Strings |
| INDEX | 0022–0025 | 34–37 | Utility Pointer Area |
| RESHO | 0026–002A | 38–42 | Floating-Point Product of Multiply |
| TXTTAB | 002B–002C | 43–44 | Pointer: Start of BASIC Text |
| VARTAB | 002D–002E | 45–46 | Pointer: Start of BASIC Variables |
| ARYTAB | 002F–0030 | 47–48 | Pointer: Start of BASIC Arrays |
| STREND | 0031–0032 | 49–50 | Pointer: End of BASIC Arrays (+1) |
| FRETOP | 0033–0034 | 51–52 | Pointer: Bottom of String Storage |
| FRESPC | 0035–0036 | 53–54 | Utility String Pointer |
| MEMSIZ | 0037–0038 | 55–56 | Pointer: Highest Address Used by BASIC |
| CURLIN | 0039–003A | 57–58 | Current BASIC Line Number |
| OLDLIN | 003B–003C | 59–60 | Previous BASIC Line Number |
| OLDTXT | 003D–003E | 61–62 | Pointer: BASIC Statement for CONT |
| DATLIN | 003F–0040 | 63–64 | Current DATA Line Number |
| DATPTR | 0041–0042 | 65–66 | Pointer: Current DATA Item Address |
| INPPTR | 0043–0044 | 67–68 | Vector: INPUT Routine |
| VARNAM | 0045–0046 | 69–70 | Current BASIC Variable Name |
| VARPNT | 0047–0048 | 71–72 | Pointer: Current BASIC Variable Data |
| FORPNT | 0049–004A | 73–74 | Pointer: Index Variable for FOR/NEXT |
| | 004B–0060 | 75–96 | Temp Pointer / Data Area |
| FACEXP | 0061 | 97 | Floating-Point Accumulator #1: Exponent |
| FACHO | 0062–0065 | 98–101 | Floating Accum. #1: Mantissa |
| FACSGN | 0066 | 102 | Floating Accum. #1: Sign |

*Commodore 64 Memory Map (continued)*

| LABEL | HEX ADDRESS | DECIMAL LOCATION | DESCRIPTION |
|---|---|---|---|
| SGNFLG | 0067 | 103 | Pointer: Series Evaluation Constant |
| BITS | 0068 | 104 | Floating Accum. #1: Overflow Digit |
| ARGEXP | 0069 | 105 | Floating-Point Accumulator #2: Exponent |
| ARGHO | 006A–006D | 106–109 | Floating Accum. #2: Mantissa |
| ARGSGN | 006E | 110 | Floating Accum. #2: Sign |
| ARISGN | 006F | 111 | Sign Comparison Result: Accum. #1 vs #2 |
| FACOV | 0070 | 112 | Floating Accum. #1. Low-Order (Rounding) |
| FBUFPT | 0071–0072 | 113–114 | Pointer: Cassette Buffer |
| CHRGET | 0073–008A | 115–138 | Subroutine: Get Next Byte of BASIC Text |
| CHRGOT | 0079 | 121 | Entry to Get Same Byte of Text Again |
| TXTPTR | 007A–007B | 122–123 | Pointer: Current Byte of BASIC Text |
| RNDX | 008B–008F | 139–143 | Floating RND Function Seed Value |
| STATUS | 0090 | 144 | Kernal I/O Status Word: ST |
| STKEY | 0091 | 145 | Flag: STOP key / RVS key |
| SVXT | 0092 | 146 | Timing Constant for Tape |
| VERCK | 0093 | 147 | Flag: 0 = Load, 1 = Verify |
| C3PO | 0094 | 148 | Flag: Serial Bus—Output Char. Buffered |
| BSOUR | 0095 | 149 | Buffered Character for Serial Bus |
| SYNO | 0096 | 150 | Cassette Sync No. |
| | 0097 | 151 | Temp Data Area |
| LDTND | 0098 | 152 | No. of Open Files / Index to File Table |
| DFLTN | 0099 | 153 | Default Input Device (0) |
| DFLTO | 009A | 154 | Default Output (CMD) Device (3) |
| PRTY | 009B | 155 | Tape Character Parity |
| DPSW | 009C | 156 | Flag: Tape Byte-Received |
| MSGFLG | 009D | 157 | Flag: $80 = Direct Mode, $00 = Program |
| PTR1 | 009E | 158 | Tape Pass 1 Error Log |
| PTR2 | 009F | 159 | Tape Pass 2 Error Log |
| TIME | 00A0–00A2 | 160–162 | Real-Time Jiffy Clock (approx) 1/60 Sec |
| | 00A3–00A4 | 163–164 | Temp Data Area |
| CNTDN | 00A5 | 165 | Cassette Sync Countdown |
| BUFPNT | 00A6 | 166 | Pointer: Tape I/O Buffer |
| INBIT | 00A7 | 167 | RS-232 Input Bits / Cassette Temp |
| BITCI | 00A8 | 168 | RS-232 Input Bit Count / Cassette Temp |
| RINONE | 00A9 | 169 | RS-232 Flag: Check for Start Bit |

| LABEL | HEX ADDRESS | DECIMAL LOCATION | DESCRIPTION |
|---|---|---|---|
| RIDATA | 00AA | 170 | RS-232 Input Byte Buffer/Cassette Temp |
| RIPRTY | 00AB | 171 | RS-232 Input Parity / Cassette Short Cnt |
| SAL | 00AC–00AD | 172–173 | Pointer: Tape Buffer/Screen Scrolling |
| EAL | 00AE–00AF | 174–175 | Tape End Addresses / End of Program |
| CMPO | 00B0–00B1 | 176–177 | Tape Timing Constants |
| TAPE1 | 00B2–00B3 | 178–179 | Pointer: Start of Tape Buffer |
| BITTS | 00B4 | 180 | RS-232 Out Bit Count / Cassette Temp |
| NXTBIT | 00B5 | 181 | RS-232 Next Bit to Send / Tape EOT Flag |
| RODATA | 00B6 | 182 | RS-232 Out Byte Buffer |
| FNLEN | 00B7 | 183 | Length of Current File Name |
| LA | 00B8 | 184 | Current Logical File Number |
| SA | 00B9 | 185 | Current Secondary Address |
| FA | 00BA | 186 | Current Device Number |
| FNADR | 00BB–00BC | 187–188 | Pointer: Current File Name |
| ROPRTY | 00BD | 189 | RS-232 Out Parity / Cassette Temp |
| FSBLK | 00BE | 190 | Cassette Read/Write Block Count |
| MYCH | 00BF | 191 | Serial Word Buffer |
| CAS1 | 00C0 | 192 | Tape Motor Interlock |
| STAL | 00C1–00C2 | 193–194 | I/O Start Address |
| MEMUSS | 00C3–00C4 | 195–196 | Tape Load Temps |
| LSTX | 00C5 | 197 | Current Key Pressed: CHR$(n) 0 = No Key |
| NDX | 00C6 | 198 | No. of Chars. in Keyboard Buffer (Queue) |
| RVS | 00C7 | 199 | Flag: Print Reverse Chars.—1 = Yes, 0 = Not Used |
| INDX | 00C8 | 200 | Pointer: End of Logical Line for INPUT |
| LXSP | 00C9–00CA | 201–202 | Cursor X-Y Pos. at Start of INPUT |
| SFDX | 00CB | 203 | Flag: Print Shifted Chars. |
| BLNSW | 00CC | 204 | Cursor Blink enable: 0 = Flash Cursor |
| BLNCT | 00CD | 205 | Timer: Countdown to Toggle Cursor |
| GDBLN | 00CE | 206 | Character Under Cursor |
| BLNON | 00CF | 207 | Flag: Last Cursor Blink On/Off |
| CRSW | 00D0 | 208 | Flag: INPUT or GET from Keyboard |
| PNT | 00D1–00D2 | 209–210 | Pointer: Current Screen Line Address |
| PNTR | 00D3 | 211 | Cursor Column on Current Line |
| QTSW | 00D4 | 212 | Flag: Editor in Quote Mode, $00 = NO |
| LNMX | 00D5 | 213 | Physical Screen Line Length |

*Commodore 64 Memory Map (continued)*

| LABEL | HEX ADDRESS | DECIMAL LOCATION | DESCRIPTION |
|---|---|---|---|
| TBLX | 00D6 | 214 | Current Cursor Physical Line Number |
| | 00D7 | 215 | Temp Data Area |
| INSRT | 00D8 | 216 | Flag: Insert Mode, >0 = # INSTs |
| LDTB1 | 00D9–00F2 | 217–242 | Screen Line Link Table / Editor Temps |
| USER | 00F3–00F4 | 243–244 | Pointer: Current Screen Color RAM loc. |
| KEYTAB | 00F5–00F6 | 245–246 | Vector: Keyboard Decode Table |
| RIBUF | 00F7–00F8 | 247–248 | RS-232 Input Buffer Pointer |
| ROBUF | 00F9–00FA | 249–250 | RS-232 Output Buffer Pointer |
| FREKZP | 00FB–00FE | 251–254 | Free 0-Page Space for User Programs |
| BASZPT | 00FF | 255 | BASIC Temp Data Area |
| | 0100–01FF | 256–511 | Micro-Processor System Stack Area |
| | 0100–010A | 256–266 | Floating to String Work Area |
| BAD | 0100–013E | 256–318 | Tape Input Error Log |
| BUF | 0200–0258 | 512–600 | System INPUT Buffer |
| LAT | 0259–0262 | 601–610 | KERNAL Table: Active Logical File No's. |
| FAT | 0263–026C | 611–620 | KERNAL Table: Device No. for Each File |
| SAT | 026D–0276 | 621–630 | KERNAL Table: Second Address Each File |
| KEYD | 0277–0280 | 631–640 | Keyboard Buffer Queue (FIFO) |
| MEMSTR | 0281–0282 | 641–642 | Pointer: Bottom of Memory for O.S. |
| MEMSIZ | 0283–0284 | 643–644 | Pointer: Top of Memory for O.S. |
| TIMOUT | 0285 | 645 | Flag: Kernal Variable for IEEE Timeout |
| COLOR | 0286 | 646 | Current Character Color Code |
| GDCOL | 0287 | 647 | Background Color Under Cursor |
| HIBASE | 0288 | 648 | Top of Screen Memory (Page) |
| XMAX | 0289 | 649 | Size of Keyboard Buffer |
| RPTFLG | 028A | 650 | Flag: REPEAT Key Used, $80 = Repeat |
| KOUNT | 028B | 651 | Repeat Speed Counter |
| DELAY | 028C | 652 | Repeat Delay Counter |
| SHFLAG | 028D | 653 | Flag: Keyb'rd SHIFT Key / CTRL Key / Key |
| LSTSHF | 028E | 654 | Last Keyboard Shift Pattern |
| KEYLOG | 028F–0290 | 655–656 | Vector: Keyboard Table Setup |
| MODE | 0291 | 657 | Flag: $00 = Disable SHIFT Keys, $80 = Enable SHIFT Keys |
| AUTODN | 0292 | 658 | Flag: Auto Scroll Down, 0 = ON |
| M51CTR | 0293 | 659 | RS-232: 6551 Control Register Image |
| M51CDR | 0294 | 660 | RS-232: 6551 Command Register Image |

| LABEL | HEX ADDRESS | DECIMAL LOCATION | DESCRIPTION |
|---|---|---|---|
| M51AJB | 0295–0296 | 661–662 | RS-232 Non-Standard BPS (Time/2-100) USA |
| RSSTAT | 0297 | 663 | RS-232: 6551 Status Register Image |
| BITNUM | 0298 | 664 | RS-232 Number of Bits Left to Send |
| BAUDOF | 0299–029A | 665–666 | RS-232 Baud Rate: Full Bit Time (μs) |
| RIDBE | 029B | 667 | RS-232 Index to End of Input Buffer |
| RIDBS | 029C | 668 | RS-232 Start of Input Buffer (Page) |
| RODBS | 029D | 669 | RS-232 Start of Output Buffer (Page) |
| RODBE | 029E | 670 | RS-232 Index to End of Output Buffer |
| IRQTMP | 029F–02A0 | 671–672 | Holds IRQ Vector During Tape I/O |
| ENABL | 02A1 | 673 | RS-232 Enables |
| | 02A2 | 674 | TOD Sense During Cassette I/O |
| | 02A3 | 675 | Temp Storage For Cassette Read |
| | 02A4 | 676 | Temp D1IRQ Indicator For Cassette Read |
| | 02A5 | 677 | Temp for Line Index |
| | 02A6 | 678 | PAL/NTSC Flag, 0 = NTSC, 1 = PAL |
| | 02A7–02FF | 697–767 | Unused |
| IERROR | 0300–0301 | 768–769 | Vector: Print BASIC Error Message |
| IMAIN | 0302–0303 | 770–771 | Vector: BASIC Warm Start |
| ICRNCH | 0304–0305 | 772–773 | Vector: Tokenize BASIC Text |
| IQPLOP | 0306–0307 | 774–775 | Vector: BASIC Text LIST |
| IGONE | 0308–0309 | 776–777 | Vector: BASIC Char. Dispatch |
| IEVAL | 030A–030B | 778–779 | Vector: BASIC Token Evaluation |
| SAREG | 030C | 780 | Storage for 6502 .A Register |
| SXREG | 030D | 781 | Storage for 6502 .X Register |
| SYREG | 030E | 782 | Storage for 6502 .Y Register |
| SPREG | 030F | 783 | Storage for 6502 .SP Register |
| USRPOK | 0310 | 784 | USR Function Jump Instr (4C) |
| USRADD | 0311–0312 | 785–786 | USR Address Low Byte / High Byte |
| | 0313 | 787 | Unused |
| CINV | 0314–0315 | 788–789 | Vector: Hardware IRQ Interrupt |
| CBINV | 0316–0317 | 790–791 | Vector: BRK Instr. Interrupt |
| NMINV | 0318–0319 | 792–793 | Vector: Non-Maskable Interrupt |
| IOPEN | 031A–031B | 794–795 | KERNAL OPEN Routine Vector |
| ICLOSE | 031C–031D | 796–797 | KERNAL CLOSE Routine Vector |
| ICHKIN | 031E–031F | 798–799 | KERNAL CHKIN Routine Vector |
| ICKOUT | 0320–0321 | 800–801 | KERNAL CHKOUT Routine Vector |
| ICLRCH | 0322–0323 | 802–803 | KERNAL CLRCHN Routine Vector |
| IBASIN | 0324–0325 | 804–805 | KERNAL CHRIN Routine Vector |
| IBSOUT | 0326–0327 | 806–807 | KERNAL CHROUT Routine Vector |
| ISTOP | 0328–0329 | 808–809 | KERNAL STOP Routine Vector |
| IGETIN | 032A–032B | 810–811 | KERNAL GETIN Routine Vector |
| ICLALL | 032C–032D | 812–813 | KERNAL CLALL Routine Vector |
| USRCMD | 032E–032F | 814–815 | User-Defined Vector |

*Commodore 64 Memory Map (continued)*

| LABEL | HEX ADDRESS | DECIMAL LOCATION | DESCRIPTION |
|---|---|---|---|
| ILOAD | 0330–0331 | 816–817 | KERNAL LOAD Routine Vector |
| ISAVE | 0332–0333 | 818–819 | KERNAL SAVE Routine Vector |
| | 0334–033B | 820–827 | Unused |
| TBUFFR | 033C–03FB | 828–1019 | Tape I/O Buffer |
| | 03FC–03FF | 1020–1023 | Unused |
| VICSCN | 0400–07FF | 1024–2047 | 1024 Byte Screen Memory Area |
| | 0400–07E7 | 1024–2023 | Video Matrix: 25 Lines × 40 Columns |
| | 07F8–07FF | 2040–2047 | Sprite Data Pointers |
| | 0800–9FFF | 2048–40959 | Normal BASIC Program Space |
| | 8000–9FFF | 32768–40959 | VSP Cartridge ROM—8192 Bytes |
| | A000–BFFF | 40960–49151 | BASIC ROM—8192 Bytes (or 8K RAM) |
| | C000–CFFF | 49152–53247 | RAM—4096 Bytes |
| | D000–DFFF | 53248–57343 | Input/Output Devices and Color RAM<br>or Character Generator ROM<br>or RAM—4096 Bytes |
| | E000–FFFF | 57344–65535 | KERNAL ROM—8192 Bytes (or 8K RAM) |

# COMMODORE 64 INPUT/OUTPUT ASSIGNMENTS

| HEX | DECIMAL | BITS | DESCRIPTION |
|---|---|---|---|
| 0000 | 0 | 7–0 | MOS 6510 Data Direction Register (xx101111)<br>Bit = 1: Output, Bit = 0: Input, x = Either |
| 0001 | 1 | | MOS 6510 Micro-Processor On-Chip I/O Port |
| | | 0 | /LORAM Signal (0 = Switch BASIC ROM Out) |
| | | 1 | /HIRAM Signal (0 = Switch Kernal ROM Out) |
| | | 2 | /CHAREN Signal (0 = Switch Char. ROM In) |
| | | 3 | Cassette Data Output Line |
| | | 4 | Cassette Switch Sense<br>1 = Switch Closed |
| | | 5 | Cassette Motor Control<br>0 = ON, 1 = OFF |
| | | 6–7 | Undefined |

| HEX | DECIMAL | BITS | DESCRIPTION |
|---|---|---|---|
| D000–D02E | 53248–54271 | | MOS 6566 VIDEO INTERFACE CONTROLLER (VIC) |
| D000 | 53248 | | Sprite 0 X Pos |
| D001 | 53249 | | Sprite 0 Y Pos |
| D002 | 53250 | | Sprite 1 X Pos |
| D003 | 53251 | | Sprite 1 Y Pos |
| D004 | 53252 | | Sprite 2 X Pos |
| D005 | 53253 | | Sprite 2 Y Pos |
| D006 | 53254 | | Sprite 3 X Pos |
| D007 | 53255 | | Sprite 3 Y Pos |
| D008 | 53256 | | Sprite 4 X Pos |
| D009 | 53257 | | Sprite 4 Y Pos |
| D00A | 53258 | | Sprite 5 X Pos |
| D00B | 53259 | | Sprite 5 Y Pos |
| D00C | 53260 | | Sprite 6 X Pos |
| D00D | 53261 | | Sprite 6 Y Pos |
| D00E | 53262 | | Sprite 7 X Pos |
| D00F | 53263 | | Sprite 7 Y Pos |
| D010 | 53264 | | Sprites 0–7 X Pos (msb of X coord.) |
| D011 | 53265 | | VIC Control Register |
| | | 7 | Raster Compare: (Bit 8) See 53266 |
| | | 6 | Extended Color Text Mode: 1 = Enable |
| | | 5 | Bit-Map Mode: 1 = Enable |
| | | 4 | Blank Screen to Border Color: 0 = Blank |
| | | 3 | Select 24/25 Row Text Display: 1 = 25 Rows |
| | | 2–0 | Smooth Scroll to Y Dot-Position (0–7) |
| D012 | 53266 | | Read Raster / Write Raster Value for Compare IRQ |
| D013 | 53267 | | Light-Pen Latch X Pos |
| D014 | 53268 | | Light-Pen Latch Y Pos |
| D015 | 53269 | | Sprite Display Enable: 1 = Enable |
| D016 | 53270 | | VIC Control Register |
| | | 7–6 | Unused |
| | | 5 | ALWAYS SET THIS BIT TO 0 |
| | | 4 | Multi-Color Mode: 1 = Enable (Text or Bit-Map) |
| | | 3 | Select 38/40 Column Text Display: 1 = 40 Cols |
| | | 2–0 | Smooth Scroll to X Pos |
| D017 | 53271 | | Sprites 0–7 Expand 2 × Vertical (Y) |
| D018 | 53272 | | VIC Memory Control Register |
| | | 7–4 | Video Matrix Base Address (inside VIC) |

*Commodore 64 Input/Output Assignments (continued)*

| HEX | DECIMAL | BITS | DESCRIPTION |
|---|---|---|---|
| | | 3–1 | Character Dot-Data Base Address (inside VIC) |
| D019 | 53273 | | VIC Interrupt Flag Register (Bit = 1: IRQ Occurred) |
| | | 7 | Set on Any Enabled VIC IRQ Condition |
| | | 3 | Light-Pen Triggered IRQ Flag |
| | | 2 | Sprite to Sprite Collision IRQ Flag |
| | | 1 | Sprite to Background Collision IRQ Flag |
| | | 0 | Raster Compare IRQ Flag |
| D01A | 53274 | | IRQ Mask Register: 1 = Interrupt Enabled |
| D01B | 53275 | | Sprite to Background Display Priority: 1 = Sprite |
| D01C | 53276 | | Sprites 0–7 Multi-Color Mode Select: 1 = M.C.M. |
| D01D | 53277 | | Sprites 0–7 Expand 2× Horizontal (X) |
| D01E | 53278 | | Sprite to Sprite Collision Detect |
| D01F | 53279 | | Sprite to Background Collision Detect |
| D020 | 53280 | | Border Color |
| D021 | 53281 | | Background Color 0 |
| D022 | 53282 | | Background Color 1 |
| D023 | 53283 | | Background Color 2 |
| D024 | 53284 | | Background Color 3 |
| D025 | 53285 | | Sprite Multi-Color Register 0 |
| D026 | 53286 | | Sprite Multi-Color Register 1 |
| D027 | 53287 | | Sprite 0 Color |
| D028 | 53288 | | Sprite 1 Color |
| D029 | 53289 | | Sprite 2 Color |
| D02A | 53290 | | Sprite 3 Color |
| D02B | 53291 | | Sprite 4 Color |
| D02C | 53292 | | Sprite 5 Color |
| D02D | 53293 | | Sprite 6 Color |
| D02E | 53294 | | Sprite 7 Color |
| D400–D7FF | 54272–55295 | | MOS 6581 SOUND INTERFACE DEVICE (SID) |
| D400 | 54272 | | Voice 1: Frequency Control—Low-Byte |
| D401 | 54273 | | Voice 1: Frequency Control—High-Byte |
| D402 | 54274 | | Voice 1: Pulse Waveform Width—Low-Byte |

| HEX | DECIMAL | BITS | DESCRIPTION |
|---|---|---|---|
| D403 | 54275 | 7–4 | Unused |
| | | 3–0 | Voice 1: Pulse Waveform Width—High-Nybble |
| D404 | 54276 | | Voice 1: Control Register |
| | | 7 | Select Random Noise Waveform, 1 = On |
| | | 6 | Select Pulse Waveform, 1 = On |
| | | 5 | Select Sawtooth Waveform, 1 = On |
| | | 4 | Select Triangle Waveform, 1 = On |
| | | 3 | Test Bit: 1 = Disable Oscillator 1 |
| | | 2 | Ring Modulate Osc. 1 with Osc. 3 Output, 1 = On |
| | | 1 | Synchronize Osc. 1 with Osc. 3 Frequency, 1 = On |
| | | 0 | Gate Bit: 1 = Start Att/Dec/Sus, 0 = Start Release |
| D405 | 54277 | | Envelope Generator 1: Attack / Decay Cycle Control |
| | | 7–4 | Select Attack Cycle Duration: 0–15 |
| | | 3–0 | Select Decay Cycle Duration: 0–15 |
| D406 | 54278 | | Envelope Generator 1: Sustain / Release Cycle Control |
| | | 7–4 | Select Sustain Cycle Duration: 0–15 |
| | | 3–0 | Select Release Cycle Duration: 0–15 |
| D407 | 54279 | | Voice 2: Frequency Control—Low-Byte |
| D40B | 54280 | | Voice 2: Frequency Control—High-Byte |
| D409 | 54281 | | Voice 2: Pulse Waveform Width—Low-Byte |
| D40A | 54282 | 7–4 | Unused |
| | | 3–0 | Voice 2: Pulse Waveform Width—High-Nybble |
| D40B | 54283 | | Voice 2: Control Register |
| | | 7 | Select Random Noise Waveform, 1 = On |
| | | 6 | Select Pulse Waveform, 1 = On |
| | | 5 | Select Sawtooth Waveform, 1 = On |
| | | 4 | Select Triangle Waveform, 1 = On |
| | | 3 | Test Bit: 1 = Disable Oscillator 2 |
| | | 2 | Ring Modulate Osc. 2 with Osc. 1 Output, 1 = On |
| | | 1 | Synchronize Osc. 2 with Osc. 1 Frequency, 1 = On |
| | | 0 | Gate Bit: 1 = Start Att/Dec/Sus, 0 = Start Release |
| D40C | 54284 | | Envelope Generator 2: Attack / Decay Cycle Control |
| | | 7–4 | Select Attack Cycle Duration: 0–15 |

*Commodore 64 Input/Output Assignments (continued)*

| HEX | DECIMAL | BITS | DESCRIPTION |
|---|---|---|---|
| | | 3–0 | Select Decay Cycle Duration: 0–15 |
| D40D | 54285 | | Envelope Generator 2: Sustain / Release Cycle Control |
| | | 7–4 | Select Sustain Cycle Duration: 0–15 |
| | | 3–0 | Select Release Cycle Duration: 0–15 |
| D40E | 54286 | | Voice 3: Frequency Control—Low-Byte |
| D40F | 54287 | | Voice 3: Frequency Control—High-Byte |
| D410 | 54288 | | Voice 3: Pulse Waveform Width—Low-Byte |
| D411 | 54289 | 7–4 | Unused |
| | | 3–0 | Voice 3: Pulse Waveform Width—High-Nybble |
| D412 | 54290 | | Voice 3: Control Register |
| | | 7 | Select Random Noise Waveform, 1 = On |
| | | 6 | Select Pulse Waveform, 1 = On |
| | | 5 | Select Sawtooth Waveform, 1 = On |
| | | 4 | Select Triangle Waveform, 1 = On |
| | | 3 | Test Bit: 1 = Disable Oscillator 3 |
| | | 2 | Ring Modulate Osc. 3 with Osc. 2 Output, 1 = On |
| | | 1 | Synchronize Osc. 3 with Osc. 2 Frequency, 1 = On |
| | | 0 | Gate Bit: 1 = Start Att/Dec/Sus, 0 = Start Release |
| D413 | 54291 | | Envelope Generator 3: Attack / Decay Cycle Control |
| | | 7–4 | Select Attack Cycle Duration: 0–15 |
| | | 3–0 | Select Decay Cycle Duration: 0–15 |
| D414 | 54292 | | Envelope Generator 3: Sustain / Release Cycle Control |
| | | 7–4 | Select Sustain Cycle Duration: 0–15 |
| | | 3–0 | Select Release Cycle Duration: 0–15 |
| D415 | 54293 | | Filter Cutoff Frequency: Low-Nybble (Bits 2–0) |
| D416 | 54294 | | Filter Cutoff Frequency: High-Byte |
| D417 | 54295 | | Filter Resonance Control / Voice Input Control |
| | | 7–4 | Select Filter Resonance: 0–15 |
| | | 3 | Filter External Input: 1 = Yes, 0 = No |
| | | 2 | Filter Voice 3 Output: 1 = Yes, 0 = No |
| | | 1 | Filter Voice 2 Output: 1 = Yes, 0 = No |

| HEX | DECIMAL | BITS | DESCRIPTION |
|---|---|---|---|
| | | 0 | Filter Voice 1 Output: 1 = Yes, 0 = No |
| D418 | 54296 | | Select Filter Mode and Volume |
| | | 7 | Cut-Off Voice 3 Output: 1 = Off, 0 = On |
| | | 6 | Select Filter High-Pass Mode: 1 = On |
| | | 5 | Select Filter Band-Pass Mode: 1 = On |
| | | 4 | Select Filter Low-Pass Mode: 1 = On |
| | | 3–0 | Select Output Volume: 0–15 |
| D419 | 54297 | | Analog/Digital Converter: Game Paddle 1 (0–255) |
| D41A | 54298 | | Analog/Digital Converter: Game Paddle 2 (0–255) |
| D41B | 54299 | | Oscillator 3 Random Number Generator |
| D41C | 54300 | | Envelope Generator 3 Output |
| D500–D7FF | 54528–55295 | | SID IMAGES |
| D800–DBFF | 55296–56319 | | Color RAM (Nybbles) |
| DC00–DCFF | 56320–56575 | | MOS 6526 Complex Interface Adapter (CIA) #1 |
| DC00 | 56320 | | Data Port A (Keyboard, Joystick, Paddles, Light-Pen) |
| | | 7–0 | Write Keyboard Column Values for Keyboard Scan |
| | | 7–6 | Read Paddles on Port A / B (01 = Port A, 10 = Port B) |
| | | 4 | Joystick A Fire Button: 1 = Fire |
| | | 3–2 | Paddle Fire Buttons |
| | | 3–0 | Joystick A Direction (0–15) |
| DC01 | 56321 | | Data Port B (Keyboard, Joystick, Paddles): Game Port 1 |
| | | 7–0 | Read Keyboard Row Values for Keyboard Scan |
| | | 7 | Timer B: Toggle/Pulse Output |
| | | 6 | Timer A: Toggle/Pulse Output |
| | | 4 | Joystick 1 Fire Button: 1 = Fire |
| | | 3–2 | Paddle Fire Buttons |
| | | 3–0 | Joystick 1 Direction |
| DC02 | 56322 | | Data Direction Register—Port A (56320) |
| DC03 | 56323 | | Data Direction Register—Port B (56321) |
| DC04 | 56324 | | Timer A: Low-Byte |
| DC05 | 56325 | | Timer A: High-Byte |
| DC06 | 56326 | | Timer B: Low-Byte |

*Commodore 64 Input/Output Assignments (continued)*

| HEX | DECIMAL | BITS | DESCRIPTION |
|---|---|---|---|
| DC07 | 56327 | | Timer B: High-Byte |
| DC08 | 56328 | | Time-of-Day Clock: 1/10 Seconds |
| DC09 | 56329 | | Time-of-Day Clock: Seconds |
| DC0A | 56330 | | Time-of-Day Clock: Minutes |
| DC0B | 56331 | | Time-of-Day Clock: Hours + AM/PM Flag (Bit 7) |
| DC0C | 56332 | | Synchronous Serial I/O Data Buffer |
| DC0D | 56333 | | CIA Interrupt Control Register (Read IRQ's/Write Mask) |
| | | 7 | IRQ Flag (1 = IRQ Occurred) / Set-Clear Flag |
| | | 4 | FLAG1 IRQ (Cassette Read / Serial Bus SRQ Input) |
| | | 3 | Serial Port Interrupt |
| | | 2 | Time-of-Day Clock Alarm Interrupt |
| | | 1 | Timer B Interrupt |
| | | 0 | Timer A Interrupt |
| DC0E | 56334 | | CIA Control Register A |
| | | 7 | Time-of-Day Clock Frequency: 1 = 50 Hz, 0 = 60 Hz |
| | | 6 | Serial Port I/O Mode: 1 = Output, 0 = Input |
| | | 5 | Timer A Counts: 1 = CNT Signals, 0 = System 02 Clock |
| | | 4 | Force Load Timer A: 1 = Yes |
| | | 3 | Timer A Run Mode: 1 = One-Shot, 0 = Continuous |
| | | 2 | Timer A Output Mode to PB6: 1 = Toggle, 0 = Pulse |
| | | 1 | Timer A Output on PB6: 1 = Yes, 0 = No |
| | | 0 | Start/Stop Timer A: 1 = Start, 0 = Stop |
| DC0F | 56335 | | CIA Control Register B |
| | | 7 | Set Alarm/TOD-Clock: 1 = Alarm, 0 = Clock |
| | | 6–5 | Timer B Mode Select:<br>00 = Count System 02 Clock Pulses<br>01 = Count Positive CNT Transitions<br>10 = Count Timer A Underflow Pulses<br>11 = Count Timer A Underflows While CNT Positive |
| | | 4–0 | Same as CIA Control Reg. A—for Timer B |

| HEX | DECIMAL | BITS | DESCRIPTION |
|---|---|---|---|
| DD00–DDFF | 56576–56831 | | MOS 6526 Complex Interface Adapter (CIA) #2 |
| DD00 | 56576 | | Data Port A (Serial Bus, RS-232, VIC Memory Control) |
| | | 7 | Serial Bus Data Input |
| | | 6 | Serial Bus Clock Pulse Input |
| | | 5 | Serial Bus Data Ouput |
| | | 4 | Serial Bus Clock Pulse Output |
| | | 3 | Serial Bus ATN Signal Output |
| | | 2 | RS-232 Data Output (User Port) |
| | | 1–0 | VIC Chip System Memory Bank Select (Default = 11) |
| DD01 | 56577 | | Data Port B (User Port, RS-232) |
| | | 7 | User / RS-232 Data Set Ready |
| | | 6 | User / RS-232 Clear to Send |
| | | 5 | User |
| | | 4 | User / RS-232 Carrier Detect |
| | | 3 | User / RS-232 Ring Indicator |
| | | 2 | User / RS-232 Data Terminal Ready |
| | | 1 | User / RS-232 Request to Send |
| | | 0 | User / RS-232 Received Data |
| DD02 | 56578 | | Data Direction Register—Port A |
| DD03 | 56579 | | Data Direction Register—Port B |
| DD04 | 56580 | | Timer A: Low-Byte |
| DD05 | 56581 | | Timer A: High-Byte |
| DD06 | 56582 | | Timer B: Low-Byte |
| DD07 | 56583 | | Timer B: High-Byte |
| DD08 | 56584 | | Time-of-Day Clock: 1/10 Seconds |
| DD09 | 56585 | | Time-of-Day Clock: Seconds |
| DD0A | 56586 | | Time-of-Day Clock: Minutes |
| DD0B | 56587 | | Time-of-Day Clock: Hours + AM/PM Flag (Bit 7) |
| DD0C | 56588 | | Synchronous Serial I/O Data Buffer |
| DD0D | 56589 | | CIA Interrupt Control Register (Read NMI's/Write Mask) |
| | | 7 | NMI Flag (1 = NMI Occurred) / Set-Clear Flag |
| | | 4 | FLAG1 NMI (User/RS-232 Received Data Input) |
| | | 3 | Serial Port Interrupt |
| | | 1 | Timer B Interrupt |
| | | 0 | Timer A Interrupt |
| DD0E | 56590 | | CIA Control Register A |
| | | 7 | Time-of-Day Clock Frequency: 1 = 50 Hz, 0 = 60 Hz |
| | | 6 | Serial Port I/O Mode: 1 = Output, 0 = Input |

*Commodore 64 Input/Output Assignments (continued)*

| HEX | DECIMAL | BITS | DESCRIPTION |
|---|---|---|---|
| | | 5 | Timer A Counts: 1 = CNT Signals, 0 = System 02 Clock |
| | | 4 | Force Load Timer A: 1 = Yes |
| | | 3 | Timer A Run Mode: 1 = One-Shot, 0 = Continuous |
| | | 2 | Timer A Output Mode to PB6: 1 = Toggle, 0 = Pulse |
| | | 1 | Timer A Output on PB6: 1 = Yes, 0 = No |
| | | 0 | Start/Stop Timer A: 1 = Start, 0 = Stop |
| DD0F | 56591 | | CIA Control Register B |
| | | 7 | Set Alarm/TOD-Clock: 1 = Alarm, 0 = Clock |
| | | 6–5 | Timer B Mode Select:<br>00 = Count System 02 Clock Pulses<br>01 = Count Positive CNT Transitions<br>10 = Count Timer A Underflow Pulses<br>11 = Count Timer A Underflows While CNT Positive |
| | | 4–0 | Same as CIA Control Reg. A—for Timer B |
| DE00–DEFF | 56832–57087 | | Reserved for Future I/O Expansion |
| DF00–DFFF | 57088–57343 | | Reserved for Future I/O Expansion |

# 16

# C128 HARDWARE SPECIFICATIONS

This chapter describes the Commodore C128 hardware, VLSI integrated circuit requirements, and the relationship between the hardware configuration and the operating system.

The C128 personal computer is compatible with C64 software and peripherals. In addition to C64 compatibility, a C128 mode exists in which 128K of RAM is available for system/user use. BASIC version 7.0 is the default language. The Commodore Kernal is supported in a compatible fashion.

The C128 also has a Z80 coprocessor that can make full use of system RAM and Kernal utilities, intended for use with such powerful operating systems as CP/M version 3.0. The Banking ROM scheme allows function key software to be installed internal to the system or added externally as an expansion cartridge.

Another major feature is the 8563 80-column display capability, available in C128 mode as an addition to the 40-column mode.

Peripheral support includes the Commodore mouse, joystick, paddle, light pen interface (both 40- and 80-column light pens); the Commodore Datassette; the User Port, which supports RS232; modems; the Expansion Bus, which supports external memory expansion; and the Commodore standard Serial Bus, which supports all existing Serial Bus components. There are also several features intended to reduce software overhead, such as relocatable zero page and system stack.

In C64 mode, the standard sixty-six keys are available. In C128 mode, twenty-six extra keys are available, including separate cursor keys, a **HELP** key, additional function keys, and a true **CAPS LOCK** key. The additional keys, grouped into the alternate keypad, are user-definable, increasing the flexibility and user friendliness of the system.

The following is a summary of C128 features:

- C64 compatibility
- 80-column display capability
- Z80 coprocessor (CP/M version 3.0 (2 MHz))
- 2 MHz 8502 operation in 80-column mode
- 128K standard system RAM
- 48K standard system ROM
- 32K internal function ROM (optional)
- 32K external cartridge ROM
- Fast serial disk drive interface
- Full keyboard, ninety-two keys with **CAPS LOCK** key, **HELP** key and separate cursor-control keys.

# SYSTEM ARCHITECTURE

The C128 computer utilizes a shared bus structure similar to that of the C64. The shared bus emulates dual-port RAM and ROM, which allows the character ROM, color RAM and system RAM to be shared by both the microprocessor and the 8564 VIC video controller, with no interference to each other. This requires that the RAM be fast enough to supply valid data in half the time of a normal microprocessor machine cycle. Normal sharing is controlled by a coprocessor that will enable or disable the processor during alternate halves of the machine cycle.

The C128 system splits the address bus into shared and nonshared sections. All normal 8502 I/O parts are on the nonshared address bus; the VIC chip and its associated support chips are located on the shared bus. The VIC chip will gate processor addresses onto the shared bus based upon its AEC control line. The data bus is common to both sides of the address bus.

The processor interfaces with most of the system chips like a standard 6502 bus cycle, where a machine cycle is equal to a clock cycle. This allows the use of 1 MHz parts for a 1 MHz clock and eliminates the need to create valid address and data strobes, as this information is now supplied by the edges of the master clock, Φ0. Chip selects for the I/O and system ROM are generated by the PLA that tracks the microprocessor addresses during Φ1.

For system RAM access, the row address is the address from the microprocessor, and the column address is the MMU output address (called the **Translated Address**). The Translated Address outputs are calculated by considering the contents of the MMU's Configuration Register and RAM Configuration Register. From these values, the MMU generates either normal or translated addresses, a CAS selection. The CAS-gating circuitry in the MMU enables either one of the two banks of 64K RAM in a 128K system. For VIC cycle access of RAM, the RAM bank is set independently of the processor's bank. A write to ROM will result in the write "bleeding through" to RAM underneath, while a read from ROM will always disable CAS in both banks. The MMU allows custom arrangements of RAM for both banks 0 and 1.

Banking ROM is effected through the setting of bits in the Configuration Register contained in the MMU and communicated to the PLA decoder. This allows Banking Function ROM and any attached C128 cartridge to be included in the basic system configuration.

The PLA generates chip selects for the color RAM, VIC control registers and character ROM, which are used during processor and VIC cycles, as well as all chip selects needed for the processor-only ROM and peripherals. To avoid bus contention, the PLA must also generate CAS disable for any accesses to ROM or I/O devices.

The VIC chip generates the signals used to control Dynamic Memory and provides macro-control functions such as RAM refresh. The VIC's primary purpose is to fetch screen data from memory, using either cycle sharing or DMA, and create an NTSC- or PAL-compatible video output that is applied to a monitor or modulated and applied to a TV set. The C128 provides outputs for Composite, Chroma/Luminance and RF video

outputs from the VIC chip, as well as an edge-triggered light pen input latch going to the VIC chip.

The output from the SID chip sound generator is buffered and applied directly to an external amplifier, like that found in an external monitor, or modulated and reproduced in the user's television set. The SID chip also has an external input for mixing another sound source.

The 8563 video control chip fetches screen data from a dedicated section of RAM referred to as the display RAM and creates an RGBI (Red-Green-Blue-Intensity) output for use with an external 80-column monitor. The 8563 also creates all needed signals for dynamic refresh of its dedicated display RAM. The C128 provides RGBI and composite monochrome outputs from the 8563 chip.

The cassette port is implemented using the zero page ports available on the 8502 and software control of hardware handshaking. The Commodore serial bus port is implemented in a similar manner using a 6526 CIA for I/O. The serial bus works with Commodore serial components, and in C64 mode is actually driven by the software routines contained in the C64. The User Port is a multipurpose port comprising several parallel port lines that support peripherals such as slow RS232, modem, etc. The joystick ports are identical to those on the C64 and are implemented using a 6526 CIA to read/write the port.

The video connector has composite video as well as separate chroma and luminance outputs for use with monitors. The 1701 and 1702 Commodore monitors interface directly to this connector. The RF output jack supplies an RF signal compatible with the regulations for TV interface devices and is switch selectable between channels 3 and 4. Both NTSC and PAL television standards are supported. The RGBI connector and signal are similar to the ones used by IBM, and are compatible with most monitors supporting Type I RGBI. Additionally, a composite monochrome signal is available on the RGBI connector and is generally compatible with NTSC (or PAL) composite. Audio is available only from the 40-column video/audio connector.

# SYSTEM SPECIFICATION

This section discusses various features and constraints of the C128 system. Included are descriptions of the system and its configurations and limiting factors such as power, loading and environment.

Figure 16–1 shows the C128 system.

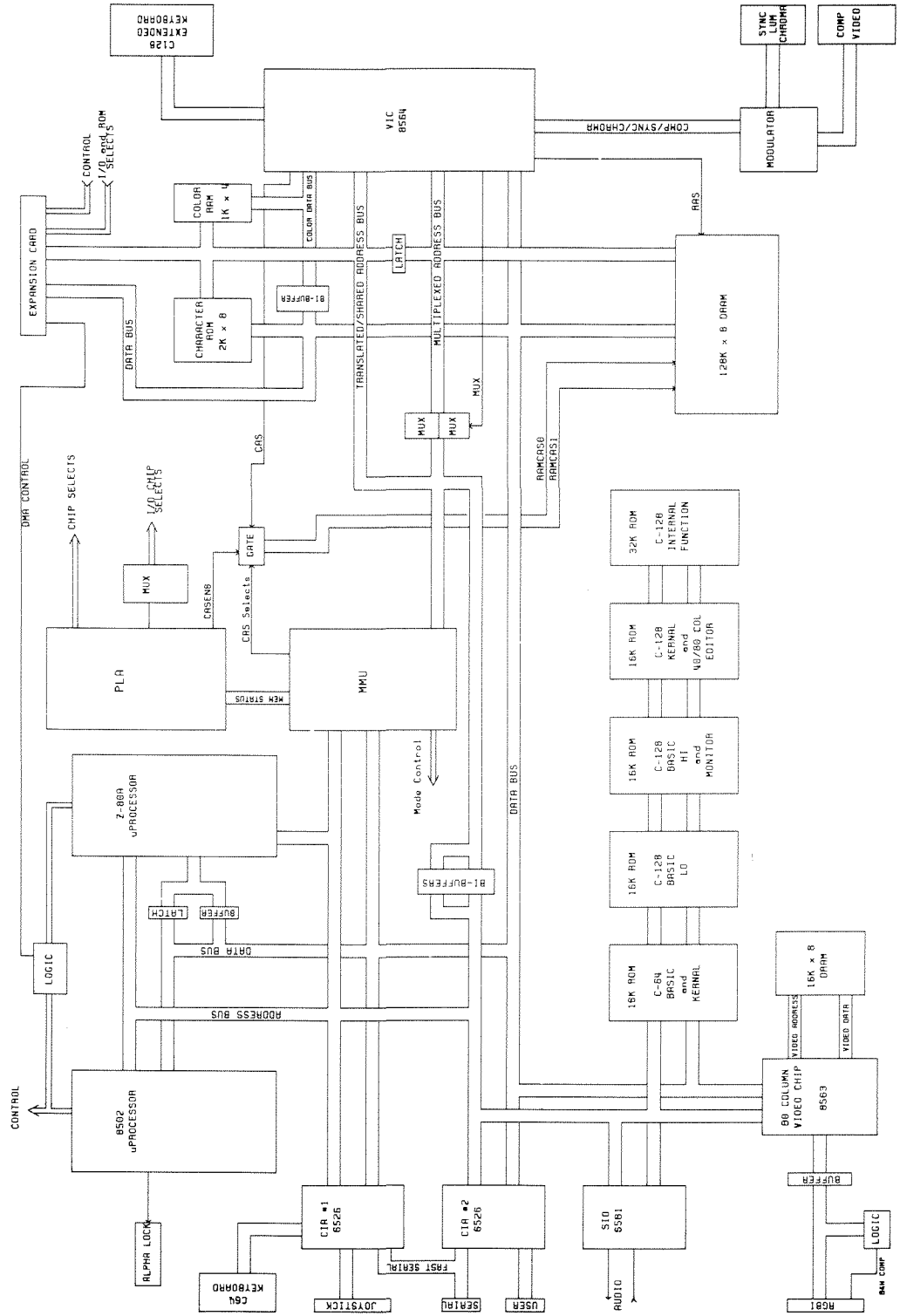



**Figure 16–1. The C128 System**

# SYSTEM BUS ARCHITECTURE

The buses described include:

- Processor bus
- Translated address bus
- Multiplexed address bus
- Shared address bus
- Color data bus
- Display bus

## THE PROCESSOR BUS

The **Processor Bus** includes the data and address buses that are connected directly to the 8502 processor. These buses are designated $D_0$-$D_7$ for the 8-bit data bus and $A_0$-$A_{15}$ for the 16-bit address bus. These buses tie the processor to most of the system ROM and I/O devices, including at least part of all system ROM, all built-in function ROM, the MMU, the PLA, the 8563 video processor, the SID and both CIA chips.

The processor bus is in communication with the Z80 coprocessor as well. All address lines are shared directly by both processors. In order to allow the Z80 to operate on a 6502 family bus, it is necessary to latch data going into the Z80 and gate the data leaving the Z80. Thus, the Z80 has a small local data bus, designated $ZD_0$-$ZD_7$. During a write cycle, when AEC is high, Z80 data is gated to the processor bus. During a read cycle, processor bus data is gated to the Z80 data bus. This read data is transparently latched by the 1 MHz system clock.

> **NOTE:** Read and write cycles referred to in this document, unless otherwise specified, are 8502-type bus cycles. The Z80 Read Enable and Write Enable outputs are conditioned using logic to interface with an 8502 bus cycle, so no distinction is made as to the differences between cycles of the different processors. For more information on this logic, consult the section on the Z80 processor and the C128 Schematic, Commodore Part No. 310378.

As mentioned above, the Z80 is not in direct communication with the processor data bus, because of the need to adapt the Z80 to 8502 bus protocol. Note, however, that every other device and every other bus (except two that will be explained later) shares the processor data bus as a common data bus.

## THE TRANSLATED ADDRESS BUS

Another C128 system bus is the **Translated Address Bus** produced by the MMU during AEC high. This bus consists only of high-order addressing lines, designated $TA_8$-$TA_{15}$. These lines reflect the action of the MMU on the normal high-order address lines, which may or may not include some sort of translation. The MMU can translate the address of page zero or page one in normal operation, and it translates the Z80 address from $0000 through $0FFF in order to direct it to read the Z80 BIOS. A more complete description of MMU translations can be found in the MMU section of this document.

Normally, the translated address bus indirectly drives the system RAM and the VIC

chip by driving the multiplexed address buses. It directly drives system ROM 4 address line 12 to allow the Z80 ROM relocation. Finally, this bus becomes address lines 8 through 15 of the C64 compatible expansion port.

During a VIC cycle or a DMA, the MMU pulls $TA_{12}$–$TA_{15}$ high, while $TA_8$–$TA_{11}$ are tri-stated. This allows the VIC chip to drive $TA_8$–$TA_{11}$ as VIC addresses $VA_8$–$VA_{11}$. During an external DMA cycle, the TA lines of the MMU are tri-stated, and the TA bus, presumably driven by the DMA source, in turn drives the processor address bus from $A_8$ to $A_{15}$. Thus allows the DMA source to access any peripheral chip except the MMU. The action of the VIC during a VIC cycle is described below.

## THE MULTIPLEXED ADDRESS BUS

This section describes two related address buses, the **Multiplexed Address Bus** and the **VIC Multiplexed Address Bus,** known respectively as $MA_0$–$MA_7$ and $VMA_0$–$VMA_7$. The VIC multiplexed address bus is created during AEC high by multiplexing the high-order translated address bus ($TA_8$–$TA_{15}$) with the low-order processor address bus ($A_0$–$A_7$), controlled via the MUX signal. This bus, driven by a hardware multiplexer through series resistors, is called the Multiplexed Address Bus. The VIC multiplexed address bus is used for processor access of the VIC chip registers. It is also used for VIC access of system RAM.

During a VIC cycle (AEC low), the VIC chip address lines will be asserted. There is no completely separate address bus for the VIC addresses, so it shares the $VMA_0$–$VMA_7$ and address lines that are tri-stated during AEC high. Most of the VIC addresses come out of the VIC chip already multiplexed, but two of them, $VA_6$ and $VA_7$, do not supply column information, as the VIC chip supplies only 14 bits of addressing. The higher-order address bits $VA_{14}$ and $VA_{15}$ come from CIA-2, as in the C64. This means the VIC supplies complete $VMA_0$–$VMA_7$ for a VIC DRAM access or DRAM refresh. The $TA_8$–$TA_{11}$ supplied by VIC are used in conjunction with another addressing bus for nonmultiplexed VIC cycle addresses, such as character ROM and color RAM accesses.

## THE SHARED ADDRESS BUS

The **Shared Address Bus** is a nonmultiplexed address bus used by both the processor and the VIC chip to communicate with common resources, namely the character ROM and color RAM (and the 8563 system RAM indirectly). During AEC high, the shared address bus, designated $Sa_0$–$SA_7$, is driven by $A_0$–$A_7$, the lower-order processor address bits. The higher-order bits needed are supplied by the translated address bus, which is also a shared address bus. Thus, the processor is able to access both shared items.

During AEC low, the VIC addresses $VA_0$–$VA_7$ ($VMA_0$–$VMA_7$) must come onto the shared address bus. Since $VA_0$–$VA_8$ are actually multiplexed, the row address only must be sent to the shared address bus. Thus, the multiplexed VIC addresses are transparently gated when either /RAS or MUX are high, but latched and held afterward so that when combined with the column address, the full address is presented. The high-order address bits here are supplied by the shared translated address bus. Note that the shared address bus provides the lower 8 bits of the expansion port address, allowing VIC access to cartridges and some additional drive capability by way of the TTL chips used to

drive the shared address bus. During DMA, the SA lines, like the TA, are driven backward to drive the processor bus. As noted above, this allows peripheral chips, ROM and RAM to be accessed by a DMA source, like the RAM expansion module. Only the MMU and the 8563 video controller cannot be accessed during DMA. See the auto-start ROM section in Chapter 13 for more details on initializing both external and internal expansion ROMS.

### THE COLOR DATA BUS

The color RAM is written to or read from a nybble data bus called the **Color Data Bus.** During AEC high, the color data bus is connected to the lower half of the processor data bus via an analog switch, allowing the processor full access to the color RAM. During AEC low, that switch is opened, effectively isolating the color data bus from the processor data bus. In this state, it is driven by the VIC extended data bus $D_8$–$D_{11}$. Since the color RAM is banked, only one-half appears here at a time.

### THE DISPLAY BUS

The **Display Bus** is a bus local to the 8563 video controller, consisting of the **Display Address** ($DA_0$–$DA_7$) and the **Display Data Bus** ($DD_0$–$DD_7$). This local bus supports the 8563 display RAM, which is completely isolated from the rest of the C128 system. The display address bus is a multiplexed address bus providing addressing to the display DRAM. The display data bus provides communication between this DRAM and the 8563. The 8563 also provides row and column strobes and dynamic refresh to this DRAM.

## SYSTEM MEMORY ORGANIZATION

This section describes the C128 memory system. Figure 16–2 is a detailed diagram of the C128 Memory Map.

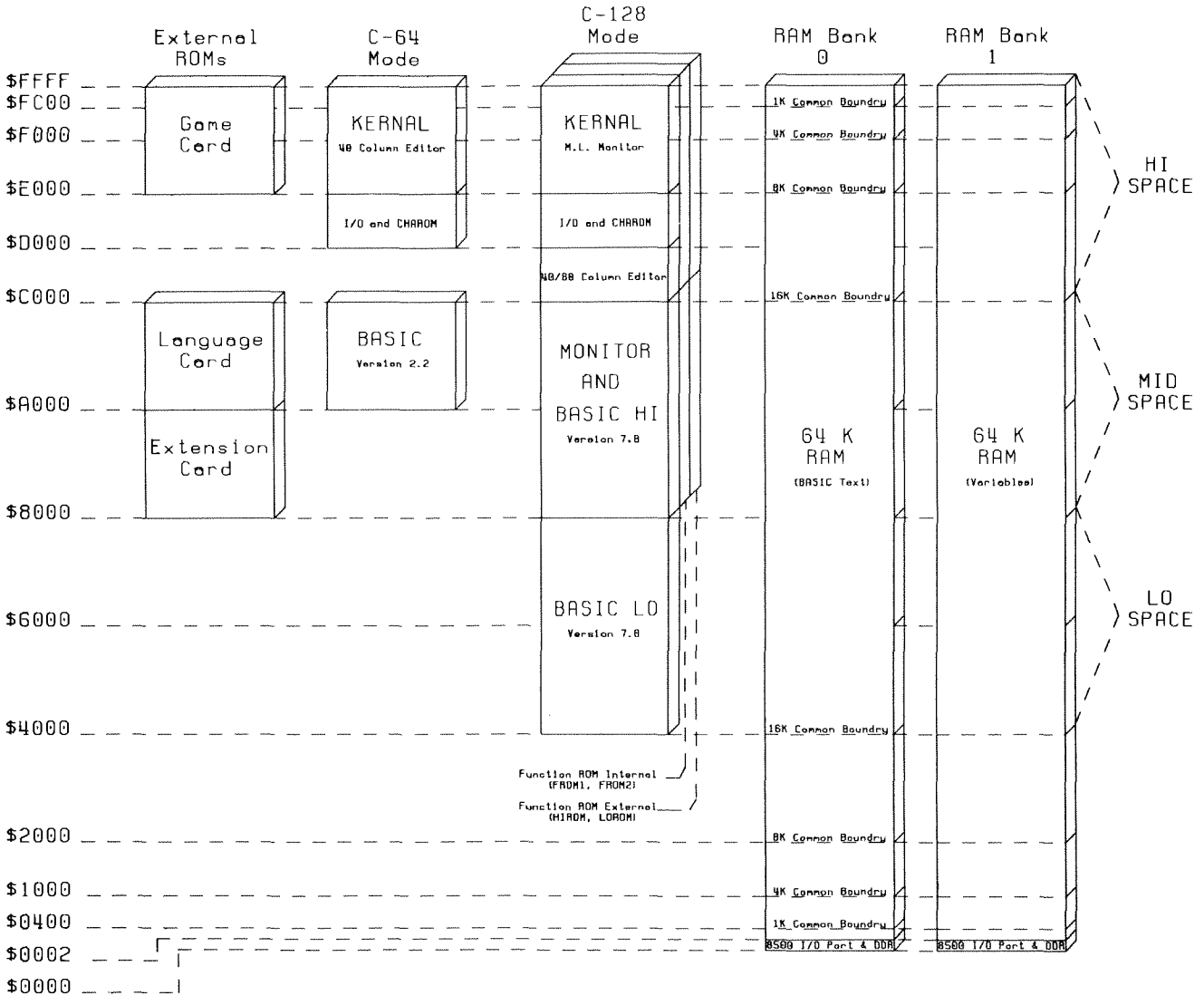



**Figure 16–2. C128 Memory Map (C64 Mode)**

## C128 ROM MEMORY ORGANIZATION

C128 mode is achieved at system reset and is controlled by a bit in the MMU Configuration Register. In C128 mode, the MMU asserts itself in the C128 memory map at $FF00 and in the I/O space, starting at $D500. Use of MMU registers located at $FF00 allows memory management without actually having the I/O block banked in at the time and with a minimum loss of contiguous RAM. The MMU is completely removed from the memory map in C64 mode. It is, however, still used by the hardware to manage memory.

Figure 16–2 presents the standard map for the C64 mode. Some of the alternate modes are shown in Figure 16–3. All C64 modes are compatible with the C64 computer, as the C128 basically becomes a C64 when in C64 mode. The details of MMU register location/operation are discussed in Chapter 13.

The ROMs in C64 mode look like Commodore 64 ROMs. The internal BASIC and Kernal provide the C64 mode with the normal Commodore 64 operating system in ROM. This ROM actually duplicates some of the ROM used in C128 mode, but it is necessary, as it is not accessible from C128 mode. In C128 mode, up to 48K of the operating system is present, with the exact amount being set by software control. This allows quicker access to underlying RAM by turning off unneeded sections of the operating system.

The external ROMs represented on the memory map are those used in C64 mode. They obey the Commodore 64 rules for mapping; i.e., cartridges assert themselves in hardware via the /EXROM and /GAME lines. External ROMs in C128 mode (i.e., C128 cartridges) are mapped as banked ROMs; when the system is initialized, all ROM slots are polled for the existence of a ROM, and the ROM's priority, if one exists. This allows much more flexibility than the hard-wired ROM substitution method, since the Kernal and BASIC ROMs can be swapped for an application program, or for external program control, or can be turned off altogether. This banking manipulation is accomplished by writing to the Configuration Register at location $D500 or $FF00 in the MMU.

The hardware also features the ability to store preset values for the configuration and force a load of the Configuration Register by writing to one of the LCRs (Load Configuration Registers). This allows the programmer to imply that ROM does not appear in the bank (by default) any time an access occurs to a bank where data is stored.

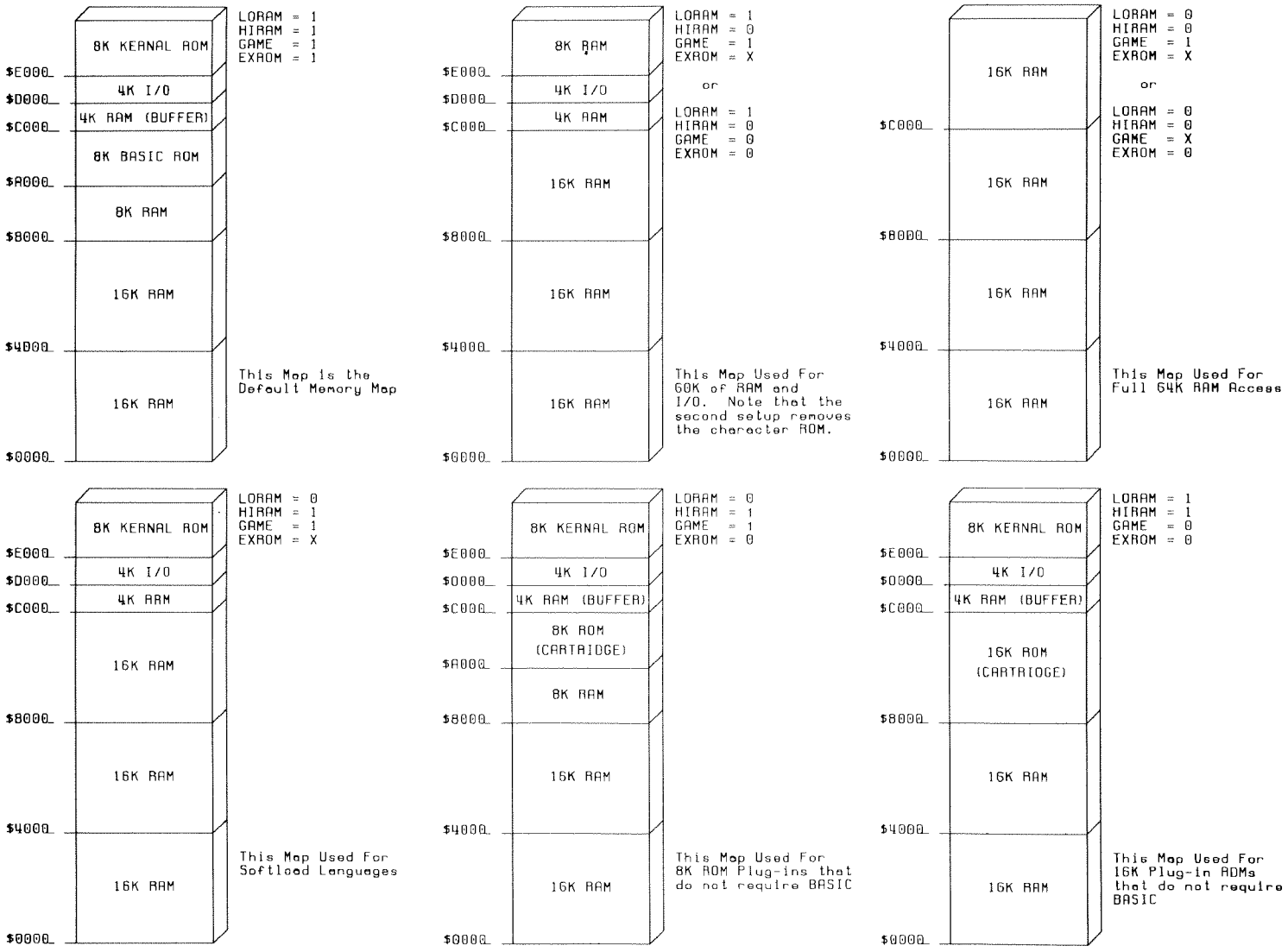



**Figure 16–3. C64 Alternate Memory Configurations**

## C128 RAM MEMORY ORGANIZATION

As shown in Figure 16–2, the RAM present in the system is actually composed of two banks of 64K by 8 bytes of DRAM. The RAM is accessed by selecting one of the two 64K banks, the maximum address range of the 8502 and Z80, according to the RAM banking rules set in the RAM configuration register of the MMU. This area shown as RAM represents what the processor would see if all ROM were disabled. Bank switching can be accomplished in one of two ways.

The bank in use is a function of the value stored in the configuration register. A store to this register will always take effect immediately. An indirect store to this register, using programmed bank configuration values, can be accomplished by writing to one of the indirect load registers known as LCRs, located in the $FF00 region of memory. By writing to an LCR, the contents of its corresponding PCR (PreConfiguration Register) will be latched into the configuration register. This allows the programmer to set up four different preprogrammed configurations that allow each bank to be personalized ahead of time; e.g., bank 1 as a data bank might be strictly a RAM bank with no ROM or I/O enabled, while bank 0 as the system bank can have the system ROM and I/O enabled by default. Additionally, reading any LCR will return the value of its corresponding PCR.

When dealing with 64K banks of memory at once, it may be desirable to bank in bank 1 but still retain the system RAM (stack, zero page, screen, etc.). The MMU has provisions for what is referred to as **common** RAM. This RAM does not bank and is programmable in size and position (top, bottom, or both) in the memory map. The size is set by bits 0 and 1 in the RAM Configuration Register (RCR). If the value of the bits is 0, 1K will be common. Values of 1, 2 and 3 produce common areas of 4K, 8K and 16K respectively. If bit 2 of the RCR is set, bottom memory is held common; if bit 3 is set, then top memory is common. In all cases, common RAM is physically located in bank 0.

Zero page and page one can be located (or relocated) independently of the RCR. When the processor accesses an address that falls within zero page or page one, the MMU adds to the high-order processor address the contents of the P0 register pair or the P1 register pair, respectively, and puts this new address on the bus, including the extended addressing bit A16. RAM banking will occur as appropriate to access the new address. Writes to the P0 and P1 registers will be stored in prelatch until a write to the respective low byte page pointer occurs. This prevents a $P_{xH}$ high byte page pointer from affecting the translated address until both high and low bytes have been written. Common memory overrides the page pointers.

At the same time, the contents of the P0 and P1 registers are applied to a digital comparator, and a reverse substitution occurs if the address from the 8502 falls within the page pointed to by the register. This results in a swapping of the zero page or page one with the memory that it replaced. Swapping occurs only if the swapped area is defined as RAM; i.e., system or function ROM must always be at its assigned addresses and thus should not be back-substituted but of course will not cause contention of any kind. Note that upon system reset, the pointers are set to true zero page and true page one.

**NOTE:** There are actually several memory modes that override parts of the bank as selected here. These modes are mentioned below, and are covered in detail in a later section.

For VIC-chip access, one bit in the MMU status register substitutes for extended address line $A_{16}$, selecting the proper CAS enable to make it possible to steer the VIC to anywhere in the 128K range. Note that AEC is the mechanism the MMU uses to steer a VIC space address; i.e., when AEC is low, a VIC access is assumed. This results in the VIC bank being selected as well for an outside DMA, since this, too, will pull the AEC line to the MMU low.

# PERFORMANCE SPECIFICATIONS

## POWER CONSUMPTION

Table 16–1 contains values for C128 power consumption, including various add-on options.

| I.C. | | $I_{Typ}$ (5v) | $I_{Max}$ | $I_{Typ}$ (9v) | $I_{Max}$ | $I_{Typ}$ (12v) | $I_{Max}$ | Unit |
|---|---|---|---|---|---|---|---|---|
| SUBTOTAL | 61 | 2.63 | 3.83 | 0.03 | 0.05 | 0 | 0 | A |
| PERIPHERALS | | | | | | | | |
| Magic Desk | 1 | 0.24 | 0.35 | 0 | 0 | 0 | 0 | A |
| RS-232 | 1 | 0.07 | 0.10 | 0 | 0 | 0 | 0 | A |
| Auto Modem | 1 | 0 | 0 | 0.07 | 0.10 | 0 | 0 | A |
| Cassette | 1 | 0 | 0 | 0.28 | 0.40 | 0 | 0 | A |
| PERIPHERAL TOTAL | 4 | 0.31 | 0.45 | 0.35 | 0.50 | 0 | 0 | A |
| C128 AND PERIPHERALS | | 2.94 | 4.28 | 0.38 | 0.55 | 0 | 0 | A |

TOTAL POWER, WORST CASE (WATTS)

| | 5V | 12V | | |
|---|---|---|---|---|
| Voltage | 5.25 | 9.45 | 12.60 | TOTAL |
| Power | 22.47 | 5.20 | 0.00 | 27.67 |

**Table 16–1. C128 Power Consumption**

## BUS LOADING

Table 16–2 details AC and DC device loading for the C128 system. All capacitances are in picofarads and currents are in microampheres.

| Dev: Signal | DRAM | ROM | VIC | 8563 | 4016 | 2332 | LS TTL | MMU | PLA | 8502 | Z80 | 4066 | SID CIA | TOTAL LOADS |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $TA_8$-$TA_{15}$ | | | | | | | | | | | | | | |
| # | --- | 1 | 1 | --- | 1 | 1 | 2 | --- | --- | --- | --- | --- | --- | |
| I | --- | 10 | 2.5 | --- | 10 | 10 | 800 | --- | --- | --- | --- | --- | --- | 833 |
| C | --- | 8 | 8 | --- | 4 | 8 | 30 | --- | --- | --- | --- | --- | --- | 58 |
| $SA_0$-$SA_7$ | | | | | | | | | | | | | | |
| # | --- | --- | --- | --- | 1 | 1 | 1 | --- | --- | --- | --- | --- | --- | |
| I | --- | --- | --- | --- | 10 | 10 | 400 | --- | --- | --- | --- | --- | --- | 420 |
| C | --- | --- | --- | --- | 4 | 8 | 15 | --- | --- | --- | --- | --- | --- | 27 |
| $D_0$-$D_7$ | | | | | | | | | | | | | | |
| # | 2 | 5 | 1 | 1 | --- | --- | 1 | 1 | --- | 1 | --- | 1 | 3 | |
| I | 20 | 50 | 2.5 | 2.5 | --- | --- | 400 | 2.5 | --- | 2.5 | --- | 0.1 | 7.5 | 488 |
| C | --- | 40 | 8 | 8 | --- | --- | 15 | 8 | --- | 8 | --- | 8 | 24 | 129 |
| R/W | | | | | | | | | | | | | | |
| # | --- | --- | 1 | 1 | --- | --- | 2 | 1 | 1 | 1 | --- | --- | 3 | |
| I | --- | --- | 2.5 | 2.5 | --- | --- | 800 | 2.5 | 2.5 | 2.5 | --- | --- | 7.5 | 820 |
| C | --- | --- | 8 | 8 | --- | --- | 30 | 8 | 8 | 8 | --- | --- | 24 | 94 |
| RES | | | | | | | | | | | | | | |
| # | --- | --- | --- | 1 | --- | --- | --- | 1 | --- | 1 | 1 | --- | 3 | |
| I | --- | --- | --- | 2.5 | --- | --- | --- | 2.5 | --- | 2.5 | 2.5 | --- | 7.5 | 18 |
| C | --- | --- | --- | 8 | --- | --- | --- | 8 | --- | 8 | 8 | --- | 24 | 56 |
| 1MHz | | | | | | | | | | | | | | |
| # | --- | --- | --- | --- | --- | --- | 5 | --- | --- | --- | --- | --- | 3 | |
| I | --- | --- | --- | --- | --- | --- | 2000 | --- | --- | --- | --- | --- | 7.5 | 2008 |
| C | --- | --- | --- | --- | --- | --- | 75 | --- | --- | --- | --- | --- | 24 | 99 |

**Table 16–2. Bus Loading Power Requirements (Capacitances and Currents)**

## ENVIRONMENTAL SPECIFICATIONS

The C128 is rated to operate at from 10 to 40 degrees Celsius (50 to 104 degrees Fahrenheit) and at a noncondensing relative humidity of 5 to 95 percent.

# THE 8502 MICROPROCESSOR

## GENERAL DESCRIPTION

The 8502 is an HMOSII Technology microprocessor, similar to the 6510/6502. It is the normal operating processor used in C64 and C128 modes. Software-compatible with the 6510, hence the 6502, the 8502 also features a zero page port used in memory management and cassette implementation.

The 8502 is also specified for operation at 2 MHz. The 2 MHz operation is made possible by removing the VIC from the system as a display chip. (The VIC chip is never completely removed from the C128 system, as it continues to function as clock generator and refresh controller.) What this refers to is that the VIC is removed as a display chip and co-processor; thus the full clock cycle can be devoted to processor functioning instead of sharing the cycle with the VIC.

The task of the video display processor is taken over by the 8563, which can function without the need for bus sharing. Since the I/O devices, SID, etc., are rated at 1 MHz only, stretching of the 2 MHz clock is used to allow these parts to be accessed directly by the 2 MHz processor and still keep throughput to a maximum.

The I/O devices are not affected by the 2 MHz operation, as they are still driven by a 1 MHz source (as such, all timer operations remain unchanged), and clock stretching is used only to synchronize the 2 MHz machine cycle to the 1 MHz Φ0 high time. The clock sources and clock-stretching capabilities are generated by the VIC chip.

## ELECTRICAL SPECIFICATION

This section describes some of the electrical constraints and specifications of the system.

### MAXIMUM RATINGS

Table 16–3 gives the absolute maximum ratings of the 8502 microprocessor.

| RATING | SYMBOL | VALUE | UNIT |
|---|---|---|---|
| **Supply Voltage** | $V_{cc}$ | **–0.5 to +7.0** | **Vdc** |
| **Input Voltage** | $V_{in}$ | **–0.5 to +7.0** | **Vdc** |
| **Storage Temperature** | $T_{stg}$ | **–55 to +150** | **°C** |
| **Operating Temperature** | $T_a$ | **0 to +70** | **°C** |

**Table 16–3. 8502 Absolute Maximum Ratings**

### ELECTRICAL CHARACTERISTICS

Table 16–4 gives the 8502's basic electrical specifications for minimum, typical and worst-case operation, valid over the range of operation T.

| CHARACTERISTIC | SYMBOL | MIN | TYP | MAX | UNIT |
|---|---|---|---|---|---|
| **Input High Voltage** | $V_{IH}$ | | | | |
| $\phi_0$ (in) | | $V_{ss}+2.4$ | — | $V_{cc}$ | Vdc |
| /RES, $P_0$–$P_7$,/IRQ, Data | | $V_{ss}+2.2$ | — | — | Vdc |
| **Input Low Voltage** | $V_{IL}$ | | | | |
| $\phi_0$ (in) | | $V_{ss}$–0.3 | — | $V_{ss}+0.5$ | Vdc |
| /RES, $P_0$–$P_7$,/IRQ, Data | | — | — | $V_{ss}+0.8$ | Vdc |
| **Input Leakage Current** | $I_{IN}$ | | | | |
| ($V_{in}$ = 0 to 5.25V, $V_{cc}$ = 5.25V) | | | | | |
| Logic | | — | — | 2.5 | μA |
| $\phi_0$ (in) | | — | — | 10.0 | μA |
| **3-State (Off) Inp. Cur.** | $I_{TSI}$ | | | | |
| ($V_{in}$ = 0.4 to 2.4V, $V_{cc}$ = 5.25V) | | | | | |
| Data Lines | | — | — | 10.0 | μA |
| **Output High Voltage** | $V_{OH}$ | | | | |
| ($I_{OH}$ = –100μAdc, $V_{cc}$ = 4.75V) | | | | | |
| Data, $A_0$–$A_{15}$, R/W, $P_0$–$P_7$ | | $V_{ss}+2.4$ | — | — | Vdc |
| **Output Low Voltage** | | | | | |
| ($I_{OL}$ = 1.6mAdc, $V_{cc}$ = 4.75V) | | | | | |
| Data, $A_0$–$A_{15}$, R/W, $P_0$–$P_7$ | | — | — | $V_{ss}+0.4$ | Vdc |
| **Power Supply Current** | $I_{cc}$ | — | 125 | — | mA |
| **Capacitance** | C | | | | |
| ($V_{in}$ = 0, $T_a$ = 25 C, f = 1MHz) | | | | | |
| Logic, $P_0$–$P_7$ | $C_{in}$ | — | — | 10 | pF |
| Data | $C_{out}$ | — | — | 15 | pF |
| $A_0$–$A_7$ | $C_{out}$ | — | — | 12 | pF |
| $\phi_0$ | C$\phi$o | — | 30 | 50 | pF |

***Table 16–4. 8502 Basic Electrical Specifications***

## SIGNAL DESCRIPTION

Below is a description of all the 8502 signals from a functional and electrical point of view:

**CLOCK ($\Phi_0$)**—This is the dual speed system clock. Note that the input level required is above worst-case TTL; thus, extra precautions must be taken when attempting to drive this input from a standard TTL level input.

**ADDRESS BUS ($A_0$–$A_{15}$)**—TTL output. Capable of driving 2 TTL loads at 130 pF.

**DATA BUS ($D_0$–$D_7$)**—Bidirectional bus for transferring data to and from the device and the peripherals. The outputs are tri-state buffers capable of driving 2 standard TTL loads at 130 pF.

**RESET**—This input is used to reset or start the processor from a power down condition. During the time that this line is held low, writing to or from the processor is inhibited. When a positive edge is detected on the input, the processor will immediately begin the reset sequence. After a system initialization time of 6 cycles, the mask interrupt flag will be set and the processor will load the program counter from the contents of memory locations $FFFC and $FFFD. This is the start location for program control. After Vcc reaches 4.75 volts in a power up routine, reset must be held low for at least 2 cycles. At this time the R/W line will become valid.

**INTERRUPT REQUEST (IRQ)**—TTL input; requests that the processor initiate an interrupt sequence. The processor will complete execution of the current instruction before recognizing the request. At that time, the interrupt mask in the Status Code Register will be examined. If the interrupt mask is not set, the processor will begin an interrupt sequence. The Program Counter and the Processor Status Register will be stored on the stack and the interrupt disable flag is set so that no other interrupts can occur. The processor will then load the program counter from the memory locations $FFFE and $FFFF.

**NON-MASKABLE INTERRUPT REQUEST (NMI)**—TTL input, negative edge sensitive request that the processor initiate an interrupt sequence. The processor will complete execution of the current instruction before recognizing the request. The Program Counter and the Processor Status Register will be stored on the stack. The processor will then load the program counter from the memory locations $FFFA and $FFFB. Since NMI is non-maskable, care must be taken to insure that the NMI request will not result in system fatality.

**ADDRESS ENABLE CONTROL (AEC)**—The Address Bus is only valid when the AEC line is high. When low, the address bus is in a high impedance state. This allows DMA's for dual processor systems.

**I/O PORT ($P_0$–$P_6$)**—Bidirectional port used for transferring data to and from the processor directly. The Data Register is located at location $0001 and the Data Direction Register is located at location $0000.

**R/W**—TTL level output from processor to control the direction of data transfer between the processor and memory, peripherals, etc. This line is high for reading memory and low for writing.

**RDY**—Ready. TTL level input, used to DMA the 8502. The processor operates normally while RDY is high. When RDY makes a transition to the low state, the processor will finish the operation it is on, and any subsequent operation if it is a write cycle. On the next occurrence of read cycle the processor will halt, making it possible to gain complete access to the system bus.

## PROCESSOR TIMING

This section explores the timing considerations of the 8502 processor unit. Table 16–5 is a processor timing chart. Figure 16–4 presents timing diagrams that show both general timing and the particular method of clock stretching used in 2 MHz mode.

**Electrical Characteristics $V_{cc}$ = 5v ± 5%, $V_{ss}$ = Ov, $T_s$ = 0 °C to 70 °C**

| CHARACTERISTIC | SYMBOL | MIN | MAX | UNITS |
|---|---|---|---|---|
| AEC setup time | $T_{AEC}$ | 25 | 60 | ns |
| Up data setup from ϕo | $T_{MDS}$ | | 100 | ns |
| Up write data hold | $T_{HW}$ | 40 | | ns |
| Data bus to tri-state from AEC | $T_{AEDT}$ | | 120 | ns |
| Read data stable | $T_{DSU}$ | 40 | | ns |
| Read data hold | $T_{HR}$ | 40 | | ns |
| Address setup from ϕo | $T_{ADS}$ | 40 | 75 | ns |
| Address hold | $T_{HA}$ | 40 | | ns |
| Address setup from AEC | $T_{AADS}$ | | 60 | ns |
| Address tri-state from AEC | $T_{AEAT}$ | | 120 | ns |
| Port input setup | $T_{PDSU}$ | 105 | | ns |
| Port input half | $T_{PDH}$ | 65 | | ns |
| Port output data valid | $T_{PDW}$ | | 195 | ns |
| Cycle time | $T_{CYC}$ | 489 | | ns |
| ϕo (in) pulse width @1.5v (crystal clock) | $P_{WH}$ϕo | 235 | 265 | ns |
| ϕo (in) rise time | TRϕo | | 10 | ns |
| ϕo (in) fall time | TFϕo | | 10 | ns |
| RDY setup time | $T_{RDY}$ | 80 | | ns |

**Table 16–5. Processor Timing Chart**

## CLOCK STRETCHING

When running in 2 MHz mode, the processor clock sometimes must be stretched. This is handled by the VIC chip, the processor and the PLA working together. When an I/O operation is decoded during a 2 MHz cycle, the phase relationship between the 2 MHz and 1 MHz clocks must be considered. If the 2 MHz access occurs during 1MHz Φ1, the access to a clocked I/O chip would be out of synchronization with the 1 MHz clock that drives all I/O chips. Thus, during this phase relationship, /IOACC from the PLA signals the VIC chip to extend the 2 MHz clock. Should the 2 MHz cycles take place during the 1 MHz Φ2 cycle, no special attention is necessary.

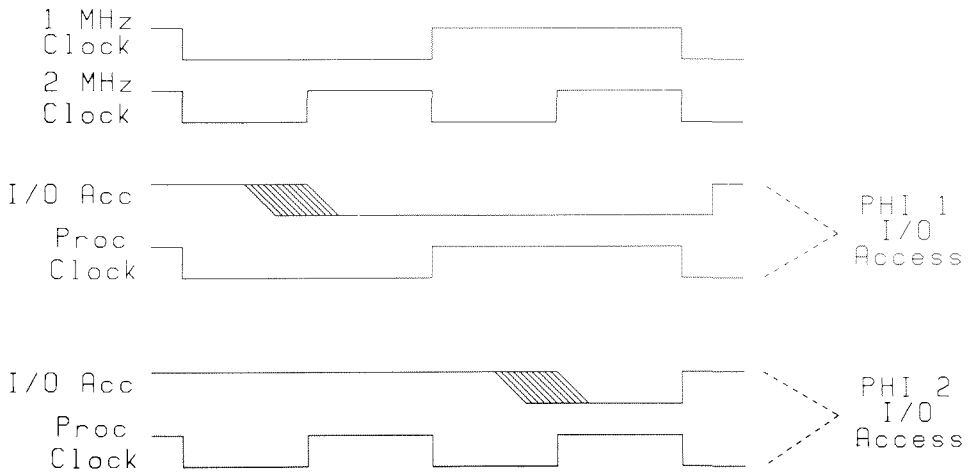



**Figure 16–4. Clock Stretching in 2 MHz Mode**

Note the speed implications of this. In 2 MHz mode, half the I/O access given will occur at an effective speed of 1MHz.

Figure 16–5 is a diagram of the 8502 microprocessor pin configuration.

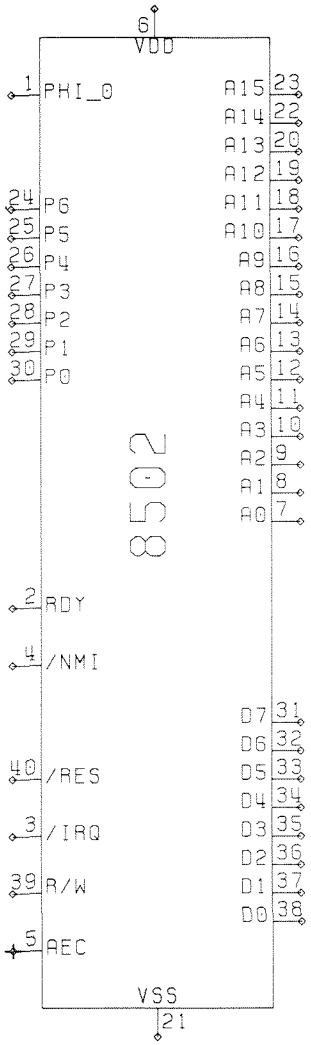



**Figure 16–5. The 8502 Microprocessor Pin Configuration**

For information on programming 8502 Machine Language, see Chapter 5, Machine Language on the C128.

# Z80 MICROPROCESSOR HARDWARE SPECIFICATION

The Z80 microprocessor is used as a secondary processor in the C128 to run CP/M based programs. Covered in this section is not only the operation of the Z80 as part of

the C128 system, but some important electrical and timing specifications of the Z80. For more information on Z80 bus interfacing, consult the Zilog Z80 Data Book.

# SYSTEM DESCRIPTION

The Z80A, a 4MHz version of Zilog's standard Z80 processor, is included as an alternate processor in the C128 system. This allows the C128 to run the CPM 3.0 operating system at an effective speed of 2 MHz. The Z80 is interfaced to the 8502 bus interface and can access all the devices that the Z80 can access. The bus interface for the Z80 (the most complex part of the Z80 implementation) is described in this section, along with Z80's operation as a coprocessor in the C128 system.

> **NOTE:** See the Signal Description section later in this chapter for definitions of the signals mentioned in the following paragraphs.

## BUS INTERFACE

Because a Z80 bus cycle is much different than a 65xx family bus cycle, a certain amount of interfacing is required for a Z80 to control a 65xx-type bus. Since the Z80 has built-in bus arbitration control lines, it is possible to isolate the Z80 by tri-stating its address lines. Thus, both the Z80 and the 8502 share common address lines.

The interfacing of the data lines is more complex. Because of the shared nature of the bus during Z80 mode, the Z80 must be isolated from the bus during AEC low. Thus, a tri-statable buffer must drive the processor bus during Z80 data writes. The reverse situation occurs during a Z80 read—the Z80 must not read things that are going on during AEC low; it must latch the data that was present during AEC high. Thus, a transparent latch drives the data input to the Z80. It is gated by the Z80 read-enable output, and latched when the 1 MHz clock is low. It will be seen that the Z80 actually runs during AEC low, but that the data bus interfaces with it only during AEC high.

## CONTROL INTERFACE

The Z80 control read-enable interfacing must provide useful clock pulses to the Z80, and must tailor the Z80 and write-enable signals for the 8502-type bus protocol. The Z80 clock is provided by the VIC chip, and is basically a 4MHz clock that occurs only during AEC low, as seen in Figure 16–6. This ensures that the Z80 is clocked only when it is actively on the bus. One additional consideration in clocking the Z80 is that while all of the 8502 levels and most of the Z80 levels are TTL-compatible, the Z80 clock input expects levels very close to 5 volts. For that reason, the output from the VIC chip is processed by the 9-volt supply; thus, the 9-volt circuit must be operational for the Z80, and the system, to function. The most common power-up failure for the C128 is a blown 9-volt fuse.

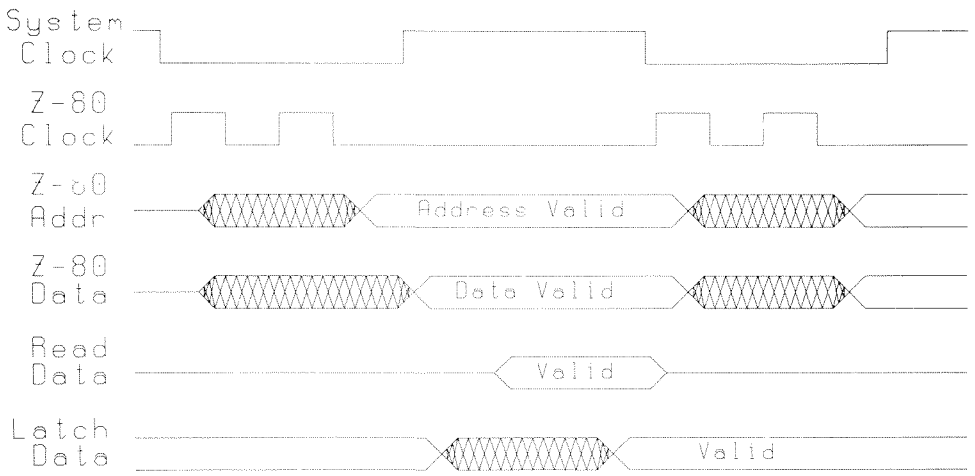



**Figure 16–6. Z80 Bus Timing**

## PROCESSOR SWITCHING

It is important in normal operation for the Z80 and the 8502 to operate as coprocessors, communicating with each other. Since only one processor may have the bus at any one time, this is only serial coprocessing, not parallel coprocessing or multiprocessing. This is important in several ways:

First, the C128 system must power-up with the Z80 as master processor. This is to prevent the Z80 from accidentally accessing the bus when powering up. Thus, the Z80 is made master on power-up and can do anything it likes to the bus. Also, because the Z80 can start-up certain C64 applications that would cause the 8502 to crash, the Z80 is the logical choice as start-up processor. After some initializations, the Z80 starts-up the 8502 in either C128 or C64 mode, depending upon whether a cartridge is present, and upon the type of cartridge, if one is present. The operating system also allows C64 mode to be forced on power-up.

Second, processor switching allows the Z80 to access 8502 Kernal routines. For standardized programs or for any I/O operation not supported in the Z80 BIOS, the Z80 can pass on the task of I/O to the 8502. Since the Z80 sees BIOS ROM where the 8502 sees its pages 0 through F, the Z80 can operate without fear of disrupting any 8502 pointers or the stack in RAM Bank 0. The Z80 ROM BIOS physically overlays that critical section of RAM Bank 1.

The Z80 can receive a bus grant request from the MMU via /Z80EN, or from the VIC chip via BA. Since the VIC control line is used for DMAs, the latter request is not of immediate concern. The /Z80EN action, however, is important, since it is the mechanism by which the two processors exchange control.

When the /Z80EN line goes high, it triggers a Z80 /BUSRQ. The Z80 then relinquishes the bus by pulling /BUSACK low. This action drives the 8502 AEC high and (providing VIC does not request a DMA) also drives the 8502 RDY line high, enabling the 8502. To switch back, a low on the Z80 /BUSRQ will result in Z80 /BUSACK going high, tri-stating and halting the 8502.

See Appendix K on CP/M for interchip communication details.

## SIGNAL DESCRIPTION

The list below defines each Z80 signal. The Z80 pin configuration is shown in Figure 16–7.

**Address Bus ($A_0$–$A_{15}$):** 16-bit tri-stating address bus. Used for 16-bit I/O cycles. This allows up to 256 input or 256 output ports. During refresh time, the lower 7 bits contain a valid refresh address. (This signal is not used in the C128 system.)

**Data Bus ($D_0$–$D_7$):** Input/output bus capable of tri-stating; used for 8-bit exchanges with memory and I/O devices.

**Machine Cycle One (/$M_1$):** Output, active low. This signal indicates that the current machine cycle is the operation code fetch of an instruction execution. During execution of a two-byte opcode, /$M_1$ is generated, as each byte is fetched. /$M_1$ also occurs with an input/output request (/IORQ) to indicate an interrupt acknowledge cycle. The $M_1$ line is used to disable the I/O decoder during an interrupt acknowledge cycle (See Input/Output Request).

**Memory Request (/MREQ):** Active low, tri-state output that indicates that the address bus holds a valid address for a memory read or write operation.

**Input/Output Request (/INRQ):** Active low, tri-state output. The /INRQ signal indicates that the lower half of the address bus holds a valid address for an I/O read or write operation. An /INRQ signal is also generated with a /$M_1$ signal when an interrupt is being acknowledged to indicate that an interrupt response vector can be placed on the data bus. An interrupt can acknowledge during /$M_1$; I/O operations never occur during /$M_1$.

**Memory Read (/RD):** Active low, tri-state output. /RD indicates that the CPU wants to read data from memory or from an I/O device. This signal is generally used to gate-read data onto the data bus.

**Memory Write (/WR):** Active low tri-state output. /WR indicates that the data bus holds valid data to be processed by memory or by an I/O device.

**Refresh (/RFSH):** Active low output used to indicate that the address bus holds a refresh address in its lower 7 bits. Thus, the current /MREQ signal should be used to do a refresh read to all dynamic memories not refreshed from an alternate source. A7 is set to 0 and the upper 8 bits contain the I register at this time.

**Halt State (/HALT):** Active low output, indicating that the Z80 has executed a halt instruction and is awaiting some kind of interrupt before execution can continue. While in the halt state, the Z80 continuously executes NOPs to continue refresh activity.

**Wait (/WAIT):** Active low input, used to drive the Z80 into wait states. As long as this signal is low, the Z80 executes wait states; thus, this signal can be used to access slow memory and I/O devices.

> **NOTE:** While the Z80 is in either a wait state or a bus acknowledge (/BUSAK) state, a dynamic memory refresh cannot be performed. See Bus Acknowledge.

**Interrupt Request (/INT):** Active low input, driven by external devices. If the interrupt flag IFF is enabled and the bus request (/BUSRQ) line is not active, the processor honors the request interrupt at the end of the current instruction. When the Z80 acknowledges an interrupt, it generates an interrupt acknowledge signal (/INRQ during $/M_1$) at the beginning of the next instruction cycle. There are three different modes of response to a given interrupt. See Bus Request.

**Non-Maskable Interrupt (/NMI):** Active low input. This interrupt is edge-triggered and cannot be masked against. It is always recognized at the end of the current instruction, forcing the Z80 to restart at location $0066. The program counter is automatically saved in the stack to allow a return from the interrupt program. Note that continuous cycles can delay an /NMI by preventing the end of the current cycle, and that /BUSRQ will override /NMI.

**Reset (/RESET):** Active low input that forces the program counter to zero and initializes the Z80, which will set interrupt mode 0, disable interrupts, and set register I and R to 0. During /RESET, the address and data buses go tri-state and all other signals go inactive.

**Bus Request (/BUSRQ):** Active low input that requests the CPU address, data and tri-statable output control signals to go tri-state for sharing and DMA's. The lines go tri-state upon termination of the current machine cycle.

**Bus Acknowledge (/BUSAK):** Active low output, used to indicate to any device taking over the bus that the Z80 has gone into tri-state and the bus has been granted.

**Clock ($\phi$):** Single phase system clock.

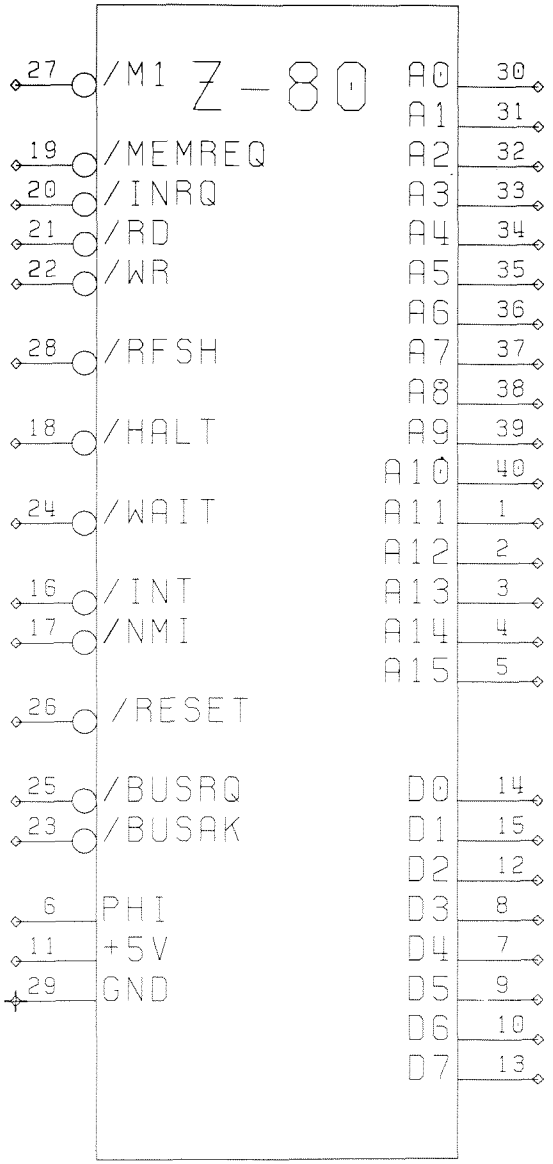



**Figure 16–7. Z80 Microprocessor Pin Configuration**

# ELECTRICAL SPECIFICATIONS

## ABSOLUTE MAXIMUM RATINGS

Table 16–6 gives the absolute maximum temperature, voltage and power dissipation ratings for the Z80. Permanent damage is likely to occur if these ratings are exceeded.

| PARAMETER | SYMBOL | RANGE | UNITS |
|---|---|---|---|
| Operating Temperature | $T_a$ | 0 to +70 | °C |
| Storage Temperature | $T_{st}$ | –65 to +150 | °C |
| Input Voltage | $V_{in}$ | –0.3 to 7.0 | Vdc |
| Power Dissipation | $P_{cc}$ | 1.5 | W |

**Table 16–6. Z80 Absolute Maximum Ratings**

## DC OPERATING CHARACTERISTICS

Table 16-7 shows the maximum DC operating ratings for the Z80. Except as noted, these ratings apply over the full rated temperature and voltage ranges.

| PARAMETER | SYMBOL | RANGE | UNITS |
|---|---|---|---|
| Power Supply Variance | $V_{cc}$ | 5 ± 5% | Vdc |
| Clock Input Low Voltage | $V_{ILC}$ | –0.3 to 0.8 | Vdc |
| Clock Input High Voltage | $V_{IHC}$ | $V_{cc}$–0.6 to $V_{cc}$ + 0.3 | Vdc |
| Input Low Voltage | $V_{IL}$ | –0.3 to 0.8 | Vdc |
| Input High Voltage | $V_{IH}$ | 2.0 to $V_{cc}$ | Vdc |
| Output Low Voltage ($I_{OL}$ = 1.8mA) | $V_{OL}$ | $V_{ss}$ to 0.4 | Vdc |
| Output High Voltage ($I_{OH}$ = –250μA) | $V_{OH}$ | 2.4 to $V_{cc}$ | Vdc |
| Power Supply Current | $I_{cc}$ | 200 | mA |
| Input Leakage Current ($V_{in}$ = 0 to $V_{cc}$) | $I_{LI}$ | 10 | μA |
| Tri-State Leakage Current ($V_{OUT}$ $V_{OH}$, $V_{OUT}$ = $V_{OL}$) | $I_{LO}$ | +10, –10 | μA |
| Data Bus Input Leakage Current ($V_{ss} \leq V_{in} \leq V_{cc}$) | $I_{LD}$ | ± 10 | μA |

**Table 16–7. Z80 DC Operating Characteristics**

## CAPACITANCE

The line capacitance values for the Z80 are given in Table 16-8. All measurements are at T = 25 degrees, F = 1 MHz.

| PARAMETER | SYMBOL | MAXIMUM | UNIT |
|---|---|---|---|
| Clock Capacitance | $C\phi$ | 35 | pF |
| Input Capacitance | $C_{in}$ | 5 | pF |
| Output Capacitance | $C_{out}$ | 10 | pF |

**Table 16–8. Z80 Capacitance Values**

# THE PROGRAMMED LOGIC ARRAY (PLA)

The 8721 C128 PLA is a programmed version of the Commodore 48 Pin Programmable Logic Array (Commodore Part #315011). It provides all the chip selects and other decoded signals that are necessary for the C64, along with a number of such signals new in the C128 system. Figure 16–8 shows the PLA chip.

The PLA does a number of things vital to the operation of the C128, including:

- All ROM selects (Kernal, BASIC, function, external) in all operating modes.
- VIC chip select.
- Color RAM chip select.
- Character RAM chip select.
- Write enable to color RAM.
- Latched write enable to DRAMs.
- Z80 select decoding.
- Z80 I/O decoding, for Z80 I/O cycle and Z80 memory mapping.
- Data bus direction signal.
- I/O group chip select (includes I/O-1, I/O-2, CIA-1, CIA-2, SID, 8563).
- I/O access signal indicating an I/O operation is occurring.
- CAS ENaBle for DRAM enable.

## CHIP SELECT GENERATION

This PLA device is responsible for defining the banking rules for ROM and RAM that the system will follow. This chip generates chip selects for all ROM and the VIC chip. It generates an enable for any other I/O device in the map, and can enable or disable CAS based upon what else is enabled. In C128 mode, decisions are made using the processor addresses and the four mode status lines: ROMBANKLO, ROMBANKHI, I/O SELECT, and C128/64. The C128 mode banking scheme is quite straightforward and simple. In Z80 mode, the selection mechanism becomes even simpler, thanks to the I/O cycle of the Z80 processor.

C64 chip selects account for the bulk of the PLA font. The C64 selects I/O, RAM and ROM based upon internal control lines: BA, HIRAM, LORAM, and CHAREN. The status of these lines, and decoded addresses, determine for any given time which (if any) chip is selected. When a cartridge is inserted, two additional control lines come into play: /EXROM and /GAME. Various combinations of these lines cause different memory maps to be asserted, all based upon the PLA font.

## OTHER FUNCTIONS

The PLA performs a variety of functions other than chip selects. It creates the write enable strobes for both DRAM and Color RAM. In C128 mode, the C64 control lines (HIRAM, LORAM and CHAREN) are not needed, since the MMU controls the more sophisticated C128 method of banking. Thus, these lines are used to extend the functionality of the C128 at little or no additional cost in hardware. The CHAREN line is

used in C128 mode to turn the Character ROM on and off in the VIC bank selected; in C128 mode the ROM can appear or disappear in any VIC bank.

The second of the new functions uses LORAM and HIRAM to select one of two Color RAM banks. The level of LORAM selects the bank that will be seen during processor time; the level of HIRAM selects the bank that will be seen during VIC time. Thus, a program can swap between two full-color pictures very cleanly, or the processor can modify one full-color picture while displaying another.

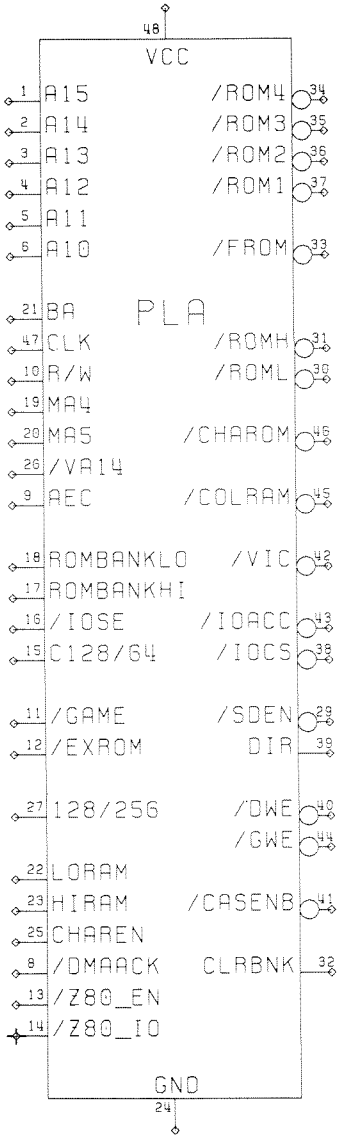



**Figure 16–8. The C128 PLA Chip**

# THE MEMORY MANAGEMENT UNIT (MMU)

The MMU is designed to allow complex control of the C128 system memory resources. Because of the way it handles all the standard C64 modes of operation, it is completely compatible with the C64. Additionally, it controls the management of particular C128 modes including the Z80 mode. The MMU's features include:

- Generation of translated address bus (TA8–TA15).
- Generation of control signals for different processor modes (C128, C64, Z80).
- Generation of CAS selects lines for RAM banking.
- Generation of ROMBANK lines for ROM banking.

Programming the MMU is described in detail in Chapter 13.

## PHYSICAL DESCRIPTION

Many of the MMU input and output signals have been discussed informally so far. This section contains descriptions of the MMU as a physial 48-pin device, including a description of all pin requirements, input and output signals, and electrical requirements.

### PIN REQUIREMENTS

Table 16–9 lists the required MMU pins, indicating the number in each signal category and the total number.

| SIGNAL NAMES | DESCRIPTION | NUMBER OF PINS |
|---|---|---|
| $A_0$–$A_3$, $A_8$–$A_{15}$ | Address Lines In | 12 |
| $A_{4/5}$, $A_{6/7}$ | Combined Address Lines In | 2 |
| $D_0$–$D_7$ | Data Lines In/Out | 8 |
| $TA_8$–$TA_{15}$ | Translated Addr. Lines Out | 8 |
| $V_{cc}$ | + 5V | 1 |
| GND | Ground | 1 |
| $PHI_0$ | 2 MHz $\phi_0$ Clock In | 1 |
| RESET | System Reset In | 1 |
| R/W | Read/Write Line In | 1 |
| /$CAS_0$–/$CAS_1$ | DRAM CAS, 64K Bank Out | 2 |
| AEC | Address Enable Control In | 1 |
| /Z8OEN | Z80 Enable Out | 1 |
| /GAME | Game ROM Enable In, Control Out | 1 |
| /EXROM | External ROM Enable In, Control Out | 1 |
| $MS_0$–$MS_1$ | Memory Status Out | 2 |
| I/O ($MS_2$) | I/O Select Out | 1 |
| C128/64 ($MS_3$) | C128 or C64 Mode Out | 1 |
| 40/80 | 40/80 Status In | 1 |
| /FSDIR | Fast Serial Direction Out | 1 |
| MUX | Memory Multiplex In | 1 |
| | TOTAL | 48 |

**Table 16–9. MMU Pin Requirements**

## PIN DESCRIPTION

The following comprises a signal-by-signal description of the MMU input and output signals. Figure 16–9 shows the MMU pin configuration. Included here are any available bond options.

The MMU input signals are:

- $A_0$–$A_3$, $A_8$–$A_{15}$: Addresses from the microprocessor. Used to derive chip selects as well as multiplexed address lines. $A_0$–$A_3$ are found at pins 18–21 on the MMU, while $A_8$–$A_{15}$ are located at pins 24–31.
- $A_{4/5}$, $A_{6/7}$: Combined addresses from the microprocessors. Used along with simple addresses, combined in this fashion to lower the pin count of the MMU. Located at pins 22 and 23 respectively.
- $PHI_0$: Presystem clock. Used for early transition of gated signals on write operations. Processor address is valid on the rising edge, and data is valid on the falling edge. This is found at pin 33.
- R/W: System Read/Write control line. This input is high for a processor read, low for a processor write. This signal is located at pin 32 on the MMU chip.
- RESET: System Reset. This input initializes internal registers on a power-up or hardware reset. It can be found at pin 2.
- AEC: Address Enable Control. Indicates whether the 8502 processor or the VIC has access to the shared bus. When low, VIC or an external DMA has the

bus and $VA_{16}$ have the processor bus, and no pointer or BIOS translation takes place. This signal occupies pin 16.

- MUX: The memory multiplex signal, used to clock various sections of the MMU. It is located at pin 17.
- $V_{dd}$: System +5 Vdc supply, connected at pin 1.
- $V_{ss}$: System Ground, connected at pin 34.

The following represents the MMU bidirectional lines. Some of the port bits detailed here are left for future expansion in a one-directional sense.

- $D_0$–$D_7$: Data inputs from microprocessor. Used for writing to internal registers. Located at pins 35 to 42.
- /EXROM: This signal is used to sense the /EXROM line on the expansion connector in C64 mode and as an expansion control line in C128 mode. Located at pin 46. This line will drive one TTL load on output, and has a passive depletion mode pull-up on input. This signal can be pulled down, but not up, by an external driver.
- /GAME: This signal is used to sense the /GAME line on the expansion connector in C64 mode and as the color RAM bank control line in C128 mode. Located at pin 45. This line will drive one TTL load on output, and has a passive depletion mode pull-up on input. External hardware can pull this line down, but not up.
- 40/80: This port in input mode senses the 40/80 column switch. It detects whether or not this switch is closed. Its output function is open for expansion. Located at pin 48. This line will drive one TTL load on output, and has a passive depletion mode pull-up on input. External hardware can pull this line down, but not up.
- FSDIR: This port in output mode is used to control the data direction of the fast serial disk interface. It is a general-purpose port signal, and is connected at pin 44. This line will drive one TTL load on output, and has a passive depletion mode pull-up on input. External hardware can pull this line down, but not up.

The following list represents the MMU output signals, their physical locations on the MMU and their logical levels if applicable.

- $TA_8$–$TA_{15}$: Translated address outputs. Tri-stated for VIC cycles during AEC, they provide translated physical addresses for use on the Multiplexed Address Bus and the Shared Static Bus. $TA_{12}$ to $TA_{15}$ are each defined to have an internal, depletion mode pull-up with an equivalent resistance of 3.3KΩ. $TA_8$ to $TA_{11}$ each go tri-state during VIC time (AEC low). These are located on the MMU at pins 10 to 3.
- $MS_0$–$MS_1$: Also called ROMBANK0 and ROMBANK1, these outputs control ROM banking for all ROM slots. They are located on pins 15 and 14. These lines are used to decode ROM bank selection for any ROM access in C128 mode. If they are both low, a system ROM has been selected. If $MS_1$ alone is

high, then a built-in function ROM has been selected. If $MS_0$ alone is high, then an external function ROM has been selected. Finally, if both are high, the RAM that occupies the particular slot has been selected. In C64 mode, the PLA completely ignores these lines.

- I/O: This output is used to select memory mapped I/O in C128 mode. It is on pin 13, and is also known as $MS_2$. In C128 mode, this line always reflects the polarity of the I/O bit. It is ignored by the PLA in C64 mode, and remains high throughout C64 mode.
- C128: This output directs the system to act in either C128 or C64 mode. It is located on pin 47, and is also known at MS3. It goes low to indicate C64 mode, high for C128 mode.
- /Z80EN: This output is used to enable the Z80 processor and disable the normal operation of the 8502 processor. It can be found at pin 43. It goes low to indicate Z80 mode, high for all other modes.
- /$CAS_0$–$CAS_1$: CAS enables to control RAM banking. $CAS_0$ enables the first bank of 64K; $CAS_1$ enables the second bank of 64K. These are pins 12 and 11, respectively.

## ABSOLUTE MAXIMUM RATINGS

| ITEM | SYMBOL | RANGE | UNITS |
|---|---|---|---|
| Input Voltage | $V_{in}$ | –2.0 to +7.0 | Vdc |
| Supply Voltage | $V_{cc}$ | –2.0 to +7.0 | Vdc |
| Operating Temperature | $T_a$ | 0 to 70 | °C |
| Storage Temperature | $T_{st}$ | –55 to 150 | °C |

**Table 16–10. Absolute Maximum Ratings for the MMU**

## MAXIMUM OPERATING CONDITIONS

| ITEM | SYMBOL | RANGE | UNITS |
|---|---|---|---|
| Voltage Variance | $V_{cc}$ | 5.0 ± 5% | Vdc |
| Input Leakage Current | $I_i$ | –1.0 | μA |
| Input High Voltage | $V_{IH}$ | $V_{ss}$ + 2.4 to $V_{cc}$ + 1.0 | Vdc |
| Input Low Voltage | $V_{IL}$ | $V_{ss}$–2.0 to $V_{ss}$ + 0.8 | Vdc |
| Output High Voltage ($I_{OH}$ = –200μA. $V_{cc}$ = 5V ± 5%) | $V_{OH}$ | $V_{ss}$ + 2.4 | Vdc |
| Output Low Voltage ($I_{OL}$ = –3.2mA, $V_{cc}$ = 5V ± 5%) | $V_{OL}$ | $V_{ss}$ + 0.4 | Vdc |
| Maximum Input Current | $I_{cc}$ | 250 | mA |

**Table 16–11. Maximum Operating Conditions for the MMU**

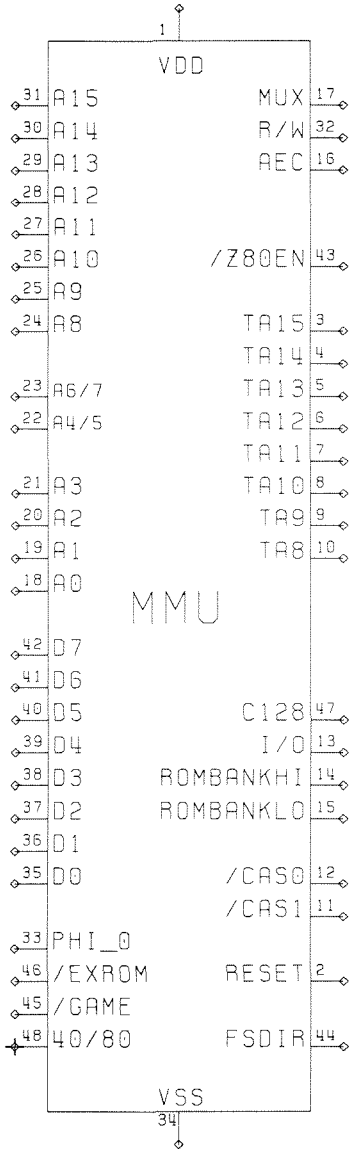



**Figure 16–9. The Memory Management Unit Pin Configuration**

# THE 8564 VIDEO INTERFACE CHIP

The 8564 VIC chip used in the C128 is an updated version of the VIC chip used in current C64 systems. It contains all the video capabilities of the earlier 6567 VIC chip, including high-resolution bit-mapped graphics and movable image blocks. It also supports new features used by the C128 system, including extended keyboard scanning. Its register map is upwardly compatible with the old VIC, allowing compatibility in C64 mode. It is powered by a single, 5V DC source instead of the two sources required by the old VIC chip. The 8564 pin configuration is shown in Figure 16–10.

## GENERAL DESCRIPTION

The 8564 VIC chip is similar to the 6567 VIC chip, yet it supports many new features unique to the C128 system requirements. It will run on a single 5V DC supply, and is packaged in a 48-pin dual-in-line package.

### 8564 VIC OPERATIONS

The 8564 VIC supports all the operations of the previous VIC chip. These functions, quickly summarized, are:

- Standard color character display mode.
- Multicolor character display mode.
- Extended color character display mode.
- Standard bit map mode.
- Multicolor bit map mode.
- Movable image blocks.
- Movable image block magnification.
- Movable image block priority
- Movable image block collision detection
- Screen blanking
- Row/column display select
- Smooth scrolling
- Light pen
- Raster compare interrupt

As these functions exist in the previous VIC, their description is purposely kept to a minimum here, while the VIC programming information is contained in Chapter 8. The new functions, however, are described in detail below.

### EXTENDED KEYBOARD SCANNING

The 8564 contains a register called the **Keyboard Control Register.** This register allows scanning of three additional keyboard control lines on the C128 keyboard. Thus, the C128 keyboard can have advanced additional keys in C128 mode while still

retaining C64 keyboard compatibility in C64 mode. In this register (register 53295 ($D02F)), bits 0–2 are directly reflected in output lines $K_0$ to $K_2$, while bits 3–7 are unused, returning high when read.

## 2 MHz OPERATION

The VIC chip contains a register that allows the C128 system to operate at 2 MHz instead of the standard 1 MHz speed of the C64. This operating speed, however, disallows the use of the VIC chip as a display processor. This is bit 0 in 53296 ($D030), and setting this bit enables 2 MHz mode. During 2 MHz operations, the VIC is disabled as a video processor. The processor spends the full time cycle on the bus, while VIC is responsible only for dynamic RAM refresh. Clearing this bit will bring back 1 MHz operation and allow the use of the VIC as a video display chip. During refresh and I/O access, the system clock is forced to 1 MHz regardless of the setting of this bit.

The 2MHz speed is available in C64 mode by setting bit 0 of location 53296 ($D030). Prior to this, blank the screen by clearing bit 4 of location 53265 ($D011). You can then process at 2MHz (to perform number crunching, for example); however, you will have no visible VIC screen. To return, set bit 4 of 53265 ($D011) and clear bit 0 of 53296 ($D030).

Bit 1 of this register contains a chip-testing facility. For normal operation, this bit must be clear. None of the other bits in this register is connected.

## SYSTEM CLOCK CONTROL

The new VIC chip generates several clocks used by the C128 system. The main clock is the 1 MHz clock, which operates at approximately 1 MHz at all times. Most bus operations and all I/O operations take place in reference to this clock. The next clock to consider is the 2 MHz clock. This clock clocks selected system components, such as the processor, at 2 MHz when in 2 MHz mode. The VIC chip monitors the /IOACC input, which indicates the access of an I/O chip, and when asserted, will stretch the 2 MHz clock to synchronize all 2 MHz parts with the 1 MHz I/O parts. Finally, the last clock is the Z80 clock, which is a 4 MHz clock that takes place only during the low half of the 1 MHz clock. Since all timed I/O parts look only at the 1 MHz clock, all I/O timings remain the same no matter what the 2 MHz clock is doing.

# SIGNAL DESCRIPTION

The VIC chip is mounted in a 48-pin dual-in-line package. The following lists describe the electrical signals that it generates.

The signals used in the system interface are:

- **$D_0$–$D_7$**: These are the bidirectional data bus signals. They are for communication between the VIC and the processor, and can be accessed only during AEC high. They occupy pins 7 through 1 and 47, respectively.
- **$D_8$–$D_{11}$**: These are the extended data bus signals. They are used for VIC communication with the color RAM. They occupy pins 47 to 43 in order.
- **/CS**: Chip select, used by the processor to select the VIC chip. Found at pin 13.
- **R/W**: Standard 8502 bus read/write for interface between the processor and the various VIC registers. Pin 14.

- **$A_0$–$A_6$**: Multiplexed address lines, pins 32 through 38. During row address time, $A_0$–$A_8$ are driven on $A_0$–$A_5$. During column address time, $A_8$–$A_{13}$ are driven on $A_0$– $A_5$ and $A_6$ is held at 1. During a processor read or write, $A_0$–$A_5$ serve as address inputs that latch on the low edge or /RAS.
- **A–$A_{10}$:** Static address lines, pins 39 through 42. These address lines are used for non-multiplexed VIC memory accesses, such as to character ROM and color RAM.
- **1MHz**: The 1 MHz system clock, pin 18. All system bus activity is referenced to this clock.
- **2MHz**: This is the changing system clock, which will be either 1 MHz or 2 MHz. If the 2 MHz bit is clear, no VIC or external DMA is taking place, and no I/O operation is occurring, the clock will be 2 MHz; otherwise it will be 1 MHz. Found at pin 23.
- **Z80 Phi**: The special 4 MHz Z80 clock, pin 25.
- **/IOACC**: From PLA, indicating an I/O chip access for clock stretching. Pin 22 of the VIC.
- **/RAS**: Row address strobe for DRAMs, pin 19.
- **/CAS**: Column address strobe for DRAMs, pin 20.
- **MUX**: Address multiplexing control for DRAMs, pin 21.
- **/IRQ**: Interrupt output, used to signal one of the various internal interrupt sources has taken place. Requires a pull-up, found at pin 8 on the chip.
- **AEC**: Address Enable Control, high for processor enable on the shared bus, low for VIC cycle and VIC or external DMA. Found at pin 12 of the VIC.
- **BA**: Bus Available signal, used to DMA processor. Pin 10.
- **$K_0$–$K_2$**: Extended keyboard strobe bits, pins 26 to 28.
- **LP**: Edge triggered latch for light pen input. Pin 9 of the VIC chip.

The signals comprised in the video interface, i.e., all the signals required to create a color video image, are:

- PH IN: The fundamental shift rate clock, also called the dot clock. Used as the reference for all system clocks. Located at pin 30.
- PH CL: The color clock, used to derive the chroma signal. Pin 29.
- SYNC: Output containing composite sync information, video data and luminance information. Requires a pull-up, pin 17.
- COLOR: Output containing all color-based video information. Open source output, should be tied through a resistor to ground. Found at pin 16.

## ELECTRICAL SPECIFICATION

Tables 16-12 and 16-13 specify the electrical operation of the VIC chip in its new form. These specs include absolute maximum ratings and maximum operating conditions.

| ITEM | SYMBOL | RANGE | UNIT |
|---|---|---|---|
| Input Voltage | $V_{in}$ | –0.5 to +7.0 | Vdc |
| Supply Voltage | $V_{cc}$ | –0.5 to +7.0 | Vdc |
| Operating Temperature | $T_a$ | 0 to 75 | °C |
| Storage Temperature | $T_{st}$ | –65 to 150 | °C |

**Table 16–12. Absolute Maximum Ratings for VIC Chip**

Below is a list of the maximum operating specifications for the new VIC chip:

| ITEM | SYMBOL | RANGE | UNIT |
|---|---|---|---|
| Power Supply Variance | $V_{cc}$ | 5.0 ± 5% | Vdc |
| Input Leakage Current | $I_I$ | –1.0 | μA |
| Input High Voltage | $V_{IH}$ | $V_{ss}$ + 2.0 to $V_{cc}$ | Vdc |
| Input Low Voltage | $V_{IL}$ | $V_{ss}$–0.5 to $V_{ss}$ + 0.8 | Vdc |
| Output High Voltage ($I_{OH}$ = –200μA, $V_{cc}$ = 5.0 ± 5% Vdc) | $V_{OH}$ | $V_{ss}$ + 2.4 | Vdc |
| Output Low Voltage ($I_{OL}$ = –3.2mA, $V_{cc}$ = 5.0 ± 5% Vdc) | $V_{OL}$ | $V_{SS}$ + 0.4 | Vdc |
| Max Power Supply Current | $I_{cc}$ | 200 | mA |

**Table 16–13. Maximum Operating Specifications for VIC Chip**

## RASTER REGISTER

The **Raster Register** is a dual function register. A read of the raster register 53266 ($D012) returns the lower 8 bits of the current raster position [the MSB-RC8 is located in register 53265 ($D011)]. A write to the raster bits (including RC8) is latched for use in an internal raster compare. When the current raster matches the written value, the raster interrupt latch is set. The raster register should be interrogated to prevent display flicker by delaying display changes to occur outside the visible area. The visible area of the display is from raster 51 to raster 251 ($033–$0FB).

## INTERRUPT REGISTER

The **Interrupt Register** indicates the status of the four sources of interrupts. An interrupt latch in register 53273 ($D019) is set to 1 when an interrupt source has generated an interrupt request.

| LATCH BIT | ENABLE BIT | WHEN SET |
|---|---|---|
| IRST | ERST | Actual raster count = stored raster count (bit 0) |
| IMDC | EMDC | MOB-DATA collision (first bit only) (bit 1) |
| IMMC | EMMC | MOB-MOB collision (first bit only) (bit 2) |
| ILP | ELP | First negative transition of LP per frame (bit 3) |
| IRQ | | When IRQ/ output low (bit 7) |

**Table 16–14. Interrupt Register Definitions**

To enable an interrupt request to set the IRQ/ output to 0, the corresponding interrupt enable bit in register 53274 ($D01A) must be set to "1". Once an interrupt latch has been set, the latch may be cleared only by writing a "1" to the associated bit in the interrupt register. This feature allows selective handling of video interrupts without software storing the active interrupts.

## SCREEN BLANKING

The display screen can be blanked by clearing the **BLNK** bit (bit 4) in register 53265 ($D011) to zero (0). When the screen is blanked, the entire screen displays the exterior color specified by register 53280 ($D020). When blanking is enabled, only transparent (phase 1) memory accesses are required, permitting full processor utilization of the system bus. However, sprite data will be accessed if the sprites are not also disabled.

## DISPLAY ROW/COLUMN SELECT

The normal display screen consists of 25 rows of 40 character regions per row. For special display purposes, the display window is reduced to 24 rows of 38 characters. There is no change in the format of the display information, except that characters (bits) adjacent to the exterior border area are covered by the border.

| RSEL | NUMBER OF ROWS | CSEL | NUMBER OF COLUMNS |
|---|---|---|---|
| 0 | 24 rows | 0 | 38 columns |
| 1 | 25 rows | 1 | 40 columns |

The **RSEL** bit (bit 3) is in register 53265 ($D011), and the **CSEL** bit (bit 3) is in register 53270 ($D016). For standard display, the larger display window is normally used, while the smaller display window is normally used in conjunction with scrolling.

## SCROLLING

The display data may be scrolled up to one character in both the horizontal and vertical directions. When used in conjunction with the smaller display window (above), scrolling can be used to create a smooth panning motion of display data while updating the system memory only when a new character row (or column) is required. Scrolling is also used for centering a display within the border. An example of horizontal smooth scrolling is found in Chapter 8.

| BITS | REGISTER | FUNCTION |
|---|---|---|
| 0–2 | 53270 ($D016) | Horizontal Position |
| 0–2 | 53265 ($D011) | Vertical Position |

## LIGHT PEN

The light pen input latches the current screen position into a pair of registers (LPX, LPY) on a low-going edge. Since the X position is defined by a 9-bit counter (53267 ($D013)), resolution to 2 horizontal dots is provided. Similarly, the Y position is latched in register 53268 ($D014) with 8 bits providing unique raster resolution within the visible display. The light pen latch may be triggered only once per frame, and subsequent triggers within the same frame will have no effect.

For more information on programming the VIC (8564) chip, see Chapter 8, The Power Behind Commodore 128 Graphics.

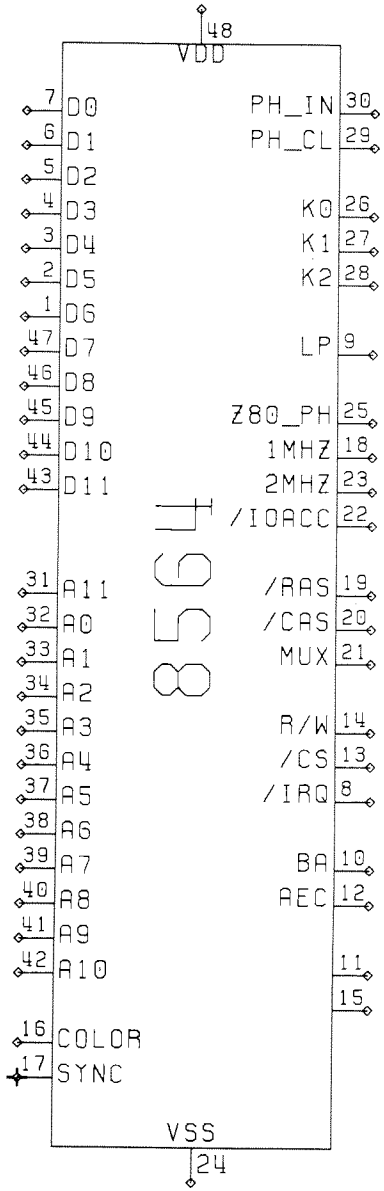



**Figure 16–10. The 8564 VIC Chip**

# THE 8563 VIDEO CONTROLLER

The 8563 is a HMOSII technology custom 80-column, color video display controller. The 8563 supplies all necessary signals to interface directly to 16K of DRAM, including refresh, and generated RGBI for use with an external RGBI monitor. For more information on the 8563 video controller, see Chapter 10, Programming the 80-Column (8563) Chip.

## GENERAL DESCRIPTION

The 8563 is a text display chip designed to implement an 80-column display system with a minimum of parts and cost. The chip contains the high-speed pixel frequency logic necessary for 80-column RGBI video. It can drive loads directly, though some buffering is desirable in most real-world applications. The chip can address up to 64K of DRAM for character font, character pointer, and attribute information. The chip provides RAS, CAS, write enable, address, data and refresh for its subordinate DRAMs. A programmable bit selects either two 4416 DRAMs (16K total) or eight 4164 DRAMs (64K total) for the display RAM. The C128 system uses the 4416 DRAMs.

### EXTERNAL REGISTERS

The 8563, which sits at $D600 in the C128, appears to the user as a device consisting of only two registers. These two registers are indirect registers that must be programmed to access the internal set of thirty-seven programming registers. The first register, located at $D600, is called the **Address Register.** Bit 7 of $D600 is the Update Ready Status Bit. When written to, the five least significant bits convey the address of an internal register to access in some way. On a read of this register, a status byte is returned. Bit 7 of this register is low while display memory is being updated, and goes high when ready for the next operation. The sixth bit will return low for an invalid light pen register condition and high for a valid light pen address. The final register indicates with a low that the scan is not in vertical blanking, and with a high that it is in vertical blanking.

The other register is the **Data Register.** It can be read from and written to. Its purpose is to write data to the internal register selected by the address register. All internal registers can be read from and written to through this register, though not all of them are a full 8 bits wide.

### INTERNAL REGISTERS

There are thirty-seven internal registers in the 8563, used for a variety of operations. They fall into two basic groups: setup registers and display registers. Setup registers are used to define internal counts for proper video display. By varying these registers, the user can configure the 8563 for NTSC, PAL or other video standards.

The display registers are used to define and manipulate characters on the screen. Once a character set has been downloaded to this chip, it is possible to display 80-column text in 4-bit digital color. There are also block movement commands that remove the time overhead needed to load large amounts of data to the chip through the two levels of indirection. Figure 16–11 is a display of the 8563 internal register map.

**Figure 16–11. 8563 Register Map**

# SIGNAL DESCRIPTION

There are many different signals involved with the 8563 chip, but they can be divided into three general categories. The CPU interface signals serve as an interface to the 8502 bus. The local bus management signals serve to maintain the local memory bus. Finally, the video interface signals are the signals that are necessary to provide an RGBI image on an RGBI monitor. The 8563 pin configuration is shown in Figure 16–12.

## THE CPU INTERFACE

The 8563 chip interfaces to the 8502 bus using a minimum of signals. This is due mainly to the local memory used by the 8563. The CPU interface signals are as follows:

- **$D_0$–$D_7$**: Bidirectional data bus allowing data to be passed between the 8563 and 8502. Found on pins 18 to 13, 11 and 10.
- **CS**: Chip select input. This input must be high for selection and proper operation of the chip. Located at pin 4.
- **/CS**: Chip select input. This input must be low for selection and proper operation of the chip. Located at pin 7.

- **/RS**: Register Select input. A high allows reads and writes to the selected data register. A low allows reads of the status register and writes to the address register. In the system, this line is tied to A0. It is located at pin 8.
- **R/W**: This line controls the data direction for the data bus. This is a typical 8502 control signal. Found at pin 9.
- **INIT**: Active low input for clearing internal control latches, allowing the chip to begin operation following initial power-on. Connect to /RES in the C128, at pin 23.
- **DISPEN**: Display Enable output signal, unused in the C128. Found at pin 19.
- **RES**: This input initializes all internal scan counters, but not control registers. It is not actively used in the C128 circuit, and is not found at pin 22.
- **TST**: Pin used for testing only, tied to ground in the C128. Located at pin 24 of the chip.

## THE LOCAL BUS MANAGEMENT INTERFACE

The local bus management interface is a group of signals generated by the 8563 for the management of local video DRAM. This local DRAM both simplifies the addition of an 80-column video display to a system and enables a computer system with a limited address space to support an 80-column display without taxing its limited memory resources.

- **$DD_0$–$DD_7$**: Bidirectional local display DRAM data bus, comprising pins 35–36 and 38–42.
- **$DA_0$–$DA_7$**: Local display DRAM multiplexed address bus. Takes up pins 26–33.
- **DR/W**: Local display DRAM Read/Write, pin 21.
- **/RAS**: Row Address Strobe for local DRAM, pin 47.
- **/CAS**: Column Address Strobe for local DRAM, pin 48.

## THE VIDEO INTERFACE

The final set of 8563 signals are the video interface signals. These signals are directly related to the display video image.

- **DCLK**: Video Dot Clock, determines the pixel width and is used internally as the timing basis for all synchronized signals, such as character clock and DRAM timing. Found at pin 2.
- **CCLK**: The character clock output, unused in the C128 system, and found at pin 1.
- **LP2**: Input for light pen; a positive going transition on this input latches the vertical and horizontal position of the character being displayed at that time. Found at pin 25.
- **HSYNC**: The horizontal sync signal, fully programmable via internal 8563 registers, and found at pin 20.
- **R,G,B,I**: The pixel data outputs. They form a 4-bit code associated with each pixel, containing color/intensity information, allowing a total of sixteen colors or gray shades to be displayed. Located at pins 46, 45, 44, 43, respectively.

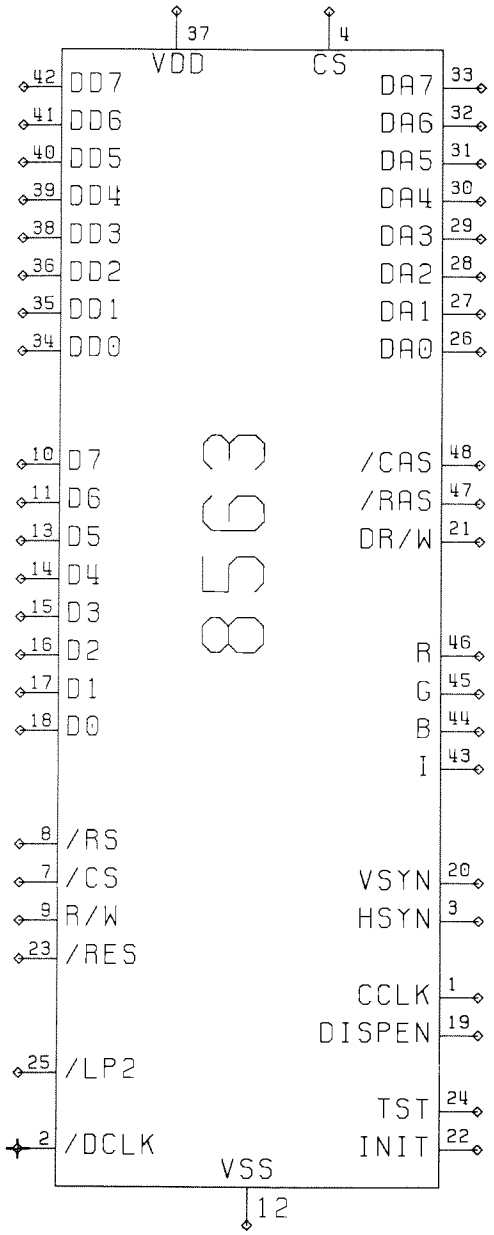



**Figure 16–12. The 8563 Chip Pin Configuration**

# 6581 SOUND INTERFACE DEVICE (SID) CHIP SPECIFICATIONS

## CONCEPT

The 6581 Sound Interface Device (SID) is a single-chip, three-voice electronic music synthesizer/sound effects generator compatible with the 8502 and similar microprocessor families. SID provides wide-range, high-resolution control of pitch (frequency), tone color (harmonic content), and dynamics (volume). Specialized control circuitry minimizes software overhead, facilitating use in arcade/home video games and low-cost musical instruments.

## FEATURES

- 3 TONE OSCILLATORS
  Range: 0–4 kHz
- 4 WAVEFORMS PER OSCILLATOR
  Triangle, Sawtooth,
  Variable Pulse, Noise
- 3 AMPLITUDE MODULATORS
  Range: 48 dB
- 3 ENVELOPE GENERATORS
  Exponential response
  Attack Rate: 2ms–8 s
  Decay Rate: 6 ms–24 s
  Sustain Level: 0–peak volume
  Release Rate: 6 ms–24 s
- OSCILLATOR SYNCHRONIZATION
- RING MODULATION
- PROGRAMMABLE FILTER
  Cutoff range: 30 Hz–12 kHz
  12 dB/octave Rolloff
  Low pass, Bandpass,
  High pass, Notch outputs
  Variable Resonance
- MASTER VOLUME CONTROL
- 2 A/D POT INTERFACES
- RANDOM NUMBER/MODULATION GENERATOR
- EXTERNAL AUDIO INPUT

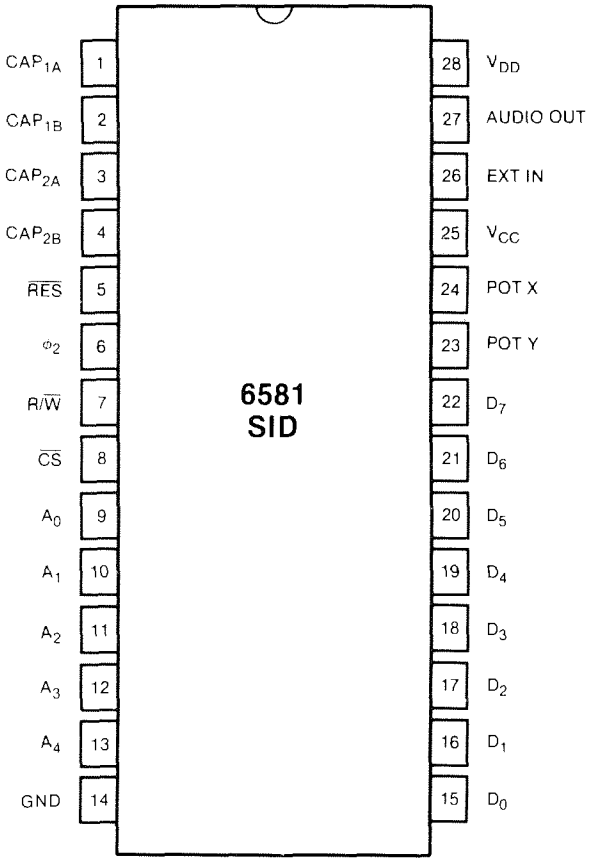



**Figure 16–13. 6581 SID Pin Configuration**

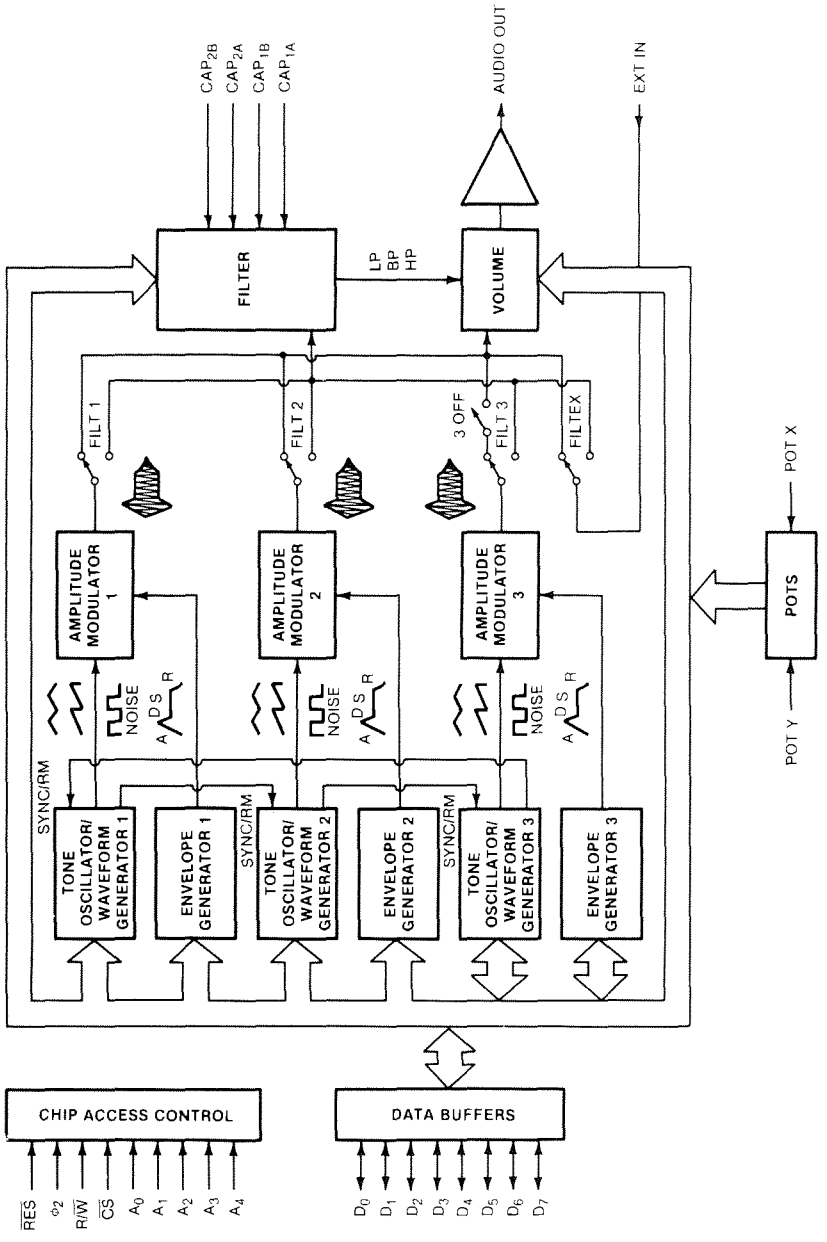



**Figure 16–14. 6581 Block Diagram**

# DESCRIPTION

The 6581 consists of three synthesizer "voices" which can be used independently or in conjunction with each other (or external audio sources) to create complex sounds. Each voice consists of a Tone Oscillator/Waveform Generator, an Envelope Generator and an Amplitude Modulator. The Tone Oscillator controls the pitch of the voice over a wide range. The Oscillator produces four waveforms at the selected frequency, with the unique harmonic content of each waveform providing simple control of tone color. The volume dynamics of the oscillator are controlled by the Amplitude Modulator under the direction of the Envelope Generator. When triggered, the Envelope Generator creates an amplitude envelope with programmable rates of increasing and decreasing volume. In addition to the three voices, a programmable Filter is provided for generating complex, dynamic tone colors via subtractive synthesis.

SID allows the microprocessor to read the changing output of the third Oscillator and third Envelope Generator. These outputs can be used as a source of modulation information for creating vibrato, frequency/filter sweeps and similar effects. The third oscillator can also act as a random number generator for games. Two A/D converters are provided for interfacing SID with potentiometers. These can be used for "paddles" in a game environment or as front panel controls in a music synthesizer. SID can process external audio signals, allowing multiple SID chips to be daisy-chained or mixed in complex polyphonic systems. For full register descriptions, see Chapter 11, Sound and Music on the Commodore 128.

# SID PIN DESCRIPTION

## CAP1A, CAP1B, (PINS 1,2)/ CAP2A,CAP2B (PINS 3,4)

These pins are used to connect the two integrating capacitors required by the programmable filter. C1 connects between pins 1 and 2, C2 between pins 3 and 4. Both capacitors should be the same value. Normal operation of the filter over the audio range (approximately 30 Hz–12 kHz) is accomplished with a value of 2200 pF for C1 and C2. Polystyrene capacitors are preferred and in complex polyphonic systems, where many SID chips must track each other, matched capacitors are recommended.

The frequency range of the filter can be tailored to specific applications by the choice of capacitor values. For example, a low-cost game may not require full high-frequency response. In this case, large values for C1 and C2 could be chosen to provide more control over the bass frequencies of the filter. The maximum cutoff frequency of the filter is given by:

$$FC_{max} = 2.6E\text{–}5/C$$

where C is the capacitor value. The range of the filter extends nine octaves below the maximum cutoff frequency.

## RES (PIN 5)

This TTL-level input is the reset control for SID. When brought low for at least ten $\phi 2$ cycles, all internal registers are reset to 0 and the audio output is silenced. This pin is normally connected to the reset line of the microprocessor or a power-on-clear circuit.

## $\phi$2 (PIN 6)

This TTL-level input is the master clock for SID. All oscillator frequencies and envelope rates are referenced to this clock. $\phi$2 also controls data transfers between SID and the microprocessor. Data can only be transferred when $\phi$2 is high. Essentially, $\phi$2 acts as a high-active chip select as far as data transfers are concerned. This pin is normally connected to the system clock, with a nominal operating frequency of 1.0 MHz.

## R/W (PIN 7)

This TTL-level input controls the direction of data transfers between SID and the microprocessor. If the chip select conditions have been met, a high on this line allows the microprocessor to Read data from the selected SID register and a low allows the microprocessor to Write data into the selected SID register. This pin is normally connected to the system Read/Write line.

## CS (PIN 8)

This TTL-level input is a low active chip select which controls data transfers between SID and the microprocessor. CS must be low for any transfer. A Read from the selected SID register can only occur if CS is low, $\phi$2 is high and R/W is high. A Write to the selected SID register can only occur if CS is low, $\phi$2 is high and R/W is low. This pin is normally connected to address decoding circuitry, allowing SID to reside in the memory map of a system.

## A0-A4 (PINS 9–13)

These TTL-level inputs are used to select one of the 29 SID registers. Although enough addresses are provided to select one of thirty-two registers, the remaining three register locations are not used. A Write to any of these three locations is ignored and a Read returns invalid data. These pins are normally connected to the corresponding address lines of the microprocessor so that SID may be addressed in the same manner as memory.

## GND (PIN 14)

For best results, the ground line between SID and the power supply should be separate from ground lines to other digital circuitry. This will minimize digital noise at the audio output.

## D0–D7 (PINS 15–22)

These bidirectional lines are used to transfer data between SID and the microprocessor. They are TTL compatible in the input mode and capable of driving two TTL loads in the output mode. The data buffers are usually in the high-impedance off state. During a Write operation, the data buffers remain in the off (input) state and the microprocessor supplies data to SID over these lines. During a Read operation, the data buffers turn on and SID supplies data to the microprocessor over these lines. The pins are normally connected to the corresponding data lines of the microprocessor.

## POTX,POTY (PINS 24,23)

These pins are inputs to the A/D converters used to digitize the position of potentiometers. The conversion process is based on the time constant of a capacitor tied from the POT pin to ground, charged by a potentiometer tied from the POT pin to +5 volts. The component values are determined by:

$$RC = 4.7E{-}4$$

Where R is the maximum resistance of the pot and C is the capacitor.

The larger the capacitor, the smaller the POT value jitter. The recommended values of R and C are 470 kΩ and 1000 pF. Note that a separate pot and cap are required for each POT pin.

## $V_{cc}$ (PIN 25)

As with the GND line, a separate +5 VDC line should be run between SID $V_{cc}$ and the power supply in order to minimize noise. A bypass capacitor should be located close to the pin.

## EXT IN (PIN 26)

This analog input allows external audio signals to be mixed with the audio output of SID or processed through the Filter. Typical sources include voice, guitar, and organ. The input impedance of this pin is on the order of 100 kΩ. Any signal applied directly to the pin should ride at a DC level of 6 volts and should not exceed 3 volts p–p. In order to prevent any interference caused by DC level differences, external signals should be AC-coupled to EXT IN by an electrolytic capacitor in the 1–10 μf range. As the direct audio path (FILTEX=0) has unity gain, EXT IN can be used to mix outputs of many SID chips by daisy-chaining. The number of chips that can be chained in this manner is determined by the amount of noise and distortion allowable at the final output. Note that the output volume control will affect not only the three SID voices, but also any external inputs.

## AUDIO OUT (PIN 27)

This open-source buffer is the final audio output of SID, comprised of the three SID voices, the filter and any external input. The output level is set by the output volume control and reaches a maximum of 2 volts p–p at a DC level of 6 volts. A source resistor from Audio Out to ground is required for proper operation. The recommended resistance is 1 kΩ for a standard output impedance.

As the output of SID rides at a 6-volt DC level, it should be AC-coupled to any audio amplifier with an electrolytic capacitor in the 1–10 μf range.

## $V_{DD}$ (PIN 28)

As with $V_{cc}$, a separate +12 VDC line should be run to SID $V_{DD}$ and a bypass capacitor should be used.

# 6581 SID CHARACTERISTICS

## ABSOLUTE MAXIMUM RATINGS

| RATING | SYMBOL | VALUE | UNITS |
|---|---|---|---|
| Supply Voltage | $V_{DD}$ | –0.3 to +17 | VDC |
| Supply Voltage | $V_{cc}$ | –0.3 to +7 | VDC |
| Input Voltage (analog) | $V_{ina}$ | –0.3 to +17 | VDC |
| Input Voltage (digital) | $V_{ind}$ | –0.3 to +7 | VDC |
| Operating Temperature | $T_A$ | 0 to +70 | °C |
| Storage Temperature | $T_{STG}$ | –55 to +150 | °C |

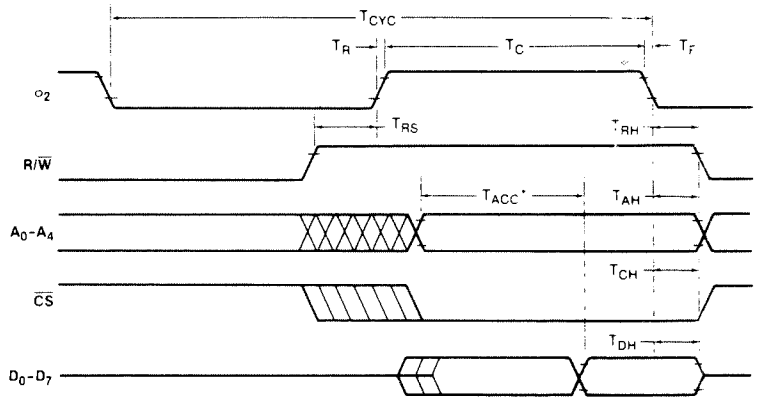
## ELECTRICAL CHARACTERISTICS

($V_{DD}$ = 12 VDC ± 5%, $V_{cc}$ = 5 VDC ± 5%, $T_A$ = 0 to 70° C)

| CHARACTERISTIC | | SYMBOL | MIN | TYP | MAX | UNITS |
|---|---|---|---|---|---|---|
| Input High Voltage | (RES, ϕ2, R/W, CS | $V_{IH}$ | 2 | — | $V_{cc}$ | VDC |
| Input Low Voltage | A0–A4, D0–D7) | $V_{IL}$ | –0.3 | — | 0.8 | VDC |
| Input Leakage Current | (RES, ϕ2, R/W, CS, A0–A4; $V_{in}$ = 0–5 VDC) | $I_{in}$ | — | — | 2.5 | μA |
| Three-State (Off) | (D0–D7; $V_{cc}$ = max) | $I_{TSI}$ | — | — | 10 | μA |
| Input Leakage Current | $V_{in}$ = 0.4–2.4 VDC | | | | | |
| Output High Voltage | (D0–D7; $V_{cc}$ = min, 1 load = 200 μA) | $V_{OH}$ | 2.4 | — | $V_{cc}$–0.7 | VDC |
| Output Low Voltage | (D0–D7; $V_{cc}$ = max, 1 load = 3.2 mA) | $V_{OL}$ | GND | — | 0.4 | VDC |
| Output High Current | (D0–D7; Sourcing, $V_{OH}$ = 2.4 VDC) | $I_{OH}$ | 200 | — | — | μA |
| Output Low Current | (D0–D7; Sinking $V_{OL}$ = 0.4 VDC) | $I_O$ | 3.2 | — | — | mA |
| Input Capacitance | (RES, ϕ2, R/W, CS, A0–A4, D0–D7) | $C_{in}$ | — | — | 10 | pF |
| Pot Trigger Voltage | (POTX, POTY) | $V_{pot}$ | — | $V_{cc}/2$ | — | VDC |
| Pot Sink Current | (POTX, POTY) | $I_{pot}$ | 500 | — | — | μA |

| CHARACTERISTIC | | SYMBOL | MIN | TYP | MAX | UNITS |
|---|---|---|---|---|---|---|
| Input Impedance | (EXT IN) | $R_n$ | 100 | 150 | — | kΩ |
| Audio Input Voltage | (EXT IN) | $V_{in}$ | 5.7 | 6 | 6.3 | VDC |
| | | | — | 0.5 | 3 | VAC |
| Audio Output Voltage | (AUDIO OUT; 1 kΩ | | | | | |
| | load, volume = max) | $V_{out}$ | 5.7 | 6 | 6.3 | VDC |
| | One Voice on: | | 0.4 | 0.5 | 0.6 | VAC |
| | All Voices on: | | 1.0 | 1.5 | 2.0 | VAC |
| Power Supply Current | ($V_{DO}$) | $I_{DD}$ | — | 20 | 25 | mA |
| Power Supply Current | ($V_{cc}$) | $1_{cc}$ | — | 70 | 100 | mA |
| Power Dissipation | (Total) | $P_D$ | — | 600 | 1000 | mW |

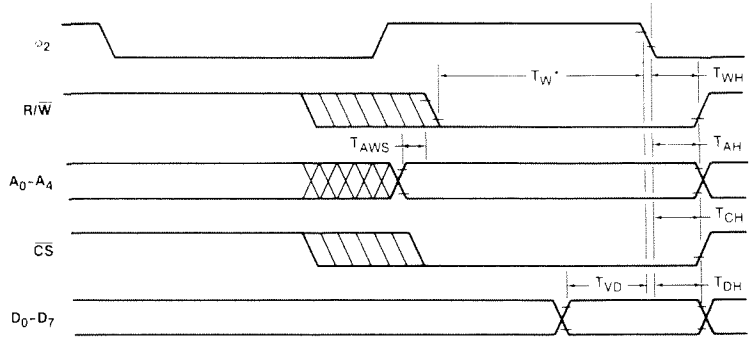
# 6581 SID TIMING




*$T_{ACC}$ is measured from the latest occurring of $o_2$, $\overline{CS}$, $A_0$-$A_4$

| SYMBOL | NAME | MIN | TYP | MAX | UNITS |
|---|---|---|---|---|---|
| $T_{CYC}$ | Clock Cycle Time | 1 | — | 20 | μs |
| $T_C$ | Clock High Pulse Width | 450 | 500 | 10,000 | ns |
| $T_R$,$T_F$ | Clock Rise/Fall Time | — | — | 25 | ns |
| $T_{RS}$ | Read Set-Up Time | 0 | — | — | ns |
| $T_{RH}$ | Read Hold time | 0 | — | — | ns |
| $T_{ACC}$ | Access Time | — | — | 300 | ns |
| $T_{AH}$ | Address Hold Time | 10 | — | — | ns |
| $T_{CH}$ | Chip Select Hold Time | 0 | — | — | ns |
| $T_{DH}$ | Data Hold Time | 20 | — | — | ns |

**Figure 16–15. Read Cycle**




*$T_W$ is measured from the latest occurring of $\phi_2$, $\overline{CS}$, R/$\overline{W}$

| SYMBOL | NAME | MIN | TYP | MAX | UNITS |
|---|---|---|---|---|---|
| $T_W$ | Write Pulse Width | 300 | — | — | ns |
| $T_{WH}$ | Write Hold Time | 0 | — | — | ns |
| $T_{AWS}$ | Address Set-up Time | 0 | — | — | ns |
| $T_{AH}$ | Address Hold Time | 10 | — | — | ns |
| $T_{CH}$ | Chip Select Hold Time | 0 | — | — | ns |
| $T_{VD}$ | Valid Data | 80 | — | — | ns |
| $T_{DH}$ | Data Hold Time | 10 | — | — | ns |

**Figure 16–16. Write Cycle**

# EQUAL-TEMPERED MUSICAL SCALE VALUES

The table in Chapter 11 lists the numerical values which must be stored in the SID Oscillator frequency control registers to produce the notes of the equal-tempered musical scale. The equal-tempered scale consists of an octave containing twelve semitones (notes): C,D,E,F,G,A,B and C#,D#,F#,G#,A#. The frequency of each semitone is exactly the 12th root of 2 ($\sqrt[12]{2}$) times the frequency of the previous semitone. The table is based on a $\phi 2$ clock of 1.02 MHz. Refer to the equation given in the Register Description in Chapter 11 for use of other master clock frequencies. The scale selected is concert pitch, in which A–4 = 440 Hz. Transpositions of this scale and scales other than the equal-tempered scale are also possible.

Although the table in Chapter 11 provides a simple and quick method for generating the equal-tempered scale, it is very memory inefficient as it requires 192 bytes for the table alone. Memory efficiency can be improved by determining the note value algorithmically. Using the fact that each note in an octave is exactly half the frequency of that note in the next octave, the note look-up table can be reduced from ninety-six entries to twelve entries, as there are twelve notes per octave. If the twelve

entries (24 bytes) consist of the 16-bit values for the eighth octave (C-7 through B-7), then notes in lower octaves can be derived by choosing the appropriate note in the eighth octave and dividing the 16-bit value by two for each octave of difference. As division by two is nothing more than a right-shift of the value, the calculation can easily be accomplished by a simple software routine. Although note B-7 is beyond the range of the oscillators, this value should still be included in the table for calculation purposes (the MSB of B-7 would require a special software case, such as generating this bit in the CARRY before shifting). Each note must be specified in a form which indicates which of the twelve semitones is desired, and which of the eight octaves the semitone is in. Since 4 bits are necessary to select one of twelve semitones and 3 bits are necessary to select one of eight octaves, the information can fit in one byte, with the lower nybble selecting the semitone (by addressing the look-up table) and the upper nybble being used by the division routine to determine how many times the table value must be right-shifted.

# SID ENVELOPE GENERATORS

The four-part ADSR (ATTACK, DECAY, SUSTAIN, RELEASE) envelope generator has been proven in electronic music to provide the optimum trade-off between flexibility and ease of amplitude control. Appropriate selection of envelope parameters allows the simulation of a wide range of percussion and sustained instruments. The violin is a good example of a sustained instrument. The violinist controls the volume by bowing the instrument. Typically, the volume builds slowly, reaches a peak, then drops to an intermediate level. The violinist can maintain this level for as long as desired, then the volume is allowed to slowly die away. A "snapshot" of this envelope is shown below:

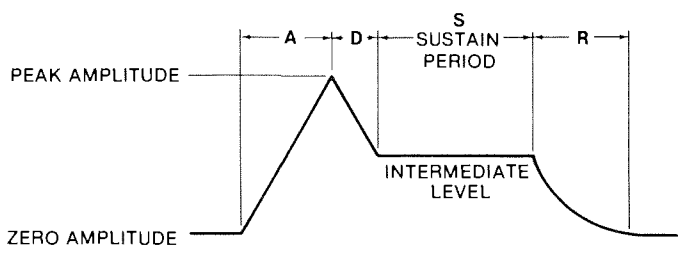
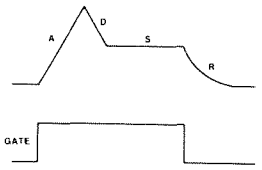



**Figure 16–17. ADSR Envelope**

This volume envelope can be easily reproduced by the ADSR as shown below, with typical envelope rates:

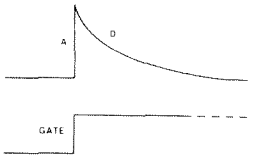
| | | |
|---|---|---|
| **ATTACK:** | **10 ($A)** | **500 ms** |
| **DECAY:** | **8** | **300 ms** |
| **SUSTAIN:** | **10 ($A)** | |
| **RELEASE:** | **9** | **750 ms** |




Note that the tone can be held at the intermediate SUSTAIN level for as long as desired. The tone will not begin to die away until GATE is cleared. With minor alterations, this basic envelope can be used for brass and woodwinds as well as strings.

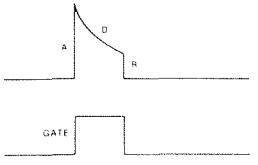
An entirely different form of envelope is produced by percussion instruments such as drums, cymbals and gongs, as well as certain keyboards such as pianos and harpsichords. The percussion envelope is characterized by a nearly instantaneous attack, immediately followed by a decay to zero volume. Percussion instruments cannot be sustained at a constant amplitude. For example, the instant a drum is struck, the sound reaches full volume and decays rapidly regardless of how it was struck. A typical cymbal envelope is shown below:

| | | |
|---|---|---|
| **ATTACK:** | **0** | **2 ms** |
| **DECAY:** | **9** | **750 ms** |
| **SUSTAIN:** | **0** | |
| **RELEASE:** | **9** | **750 ms** |




Note that the tone immediately begins to decay to zero amplitude after the peak is reached, regardless of when GATE is cleared. The amplitude envelope of pianos and harpsichords is somewhat more complicated, but can be generated quite easily with the ADSR. These instruments reach full volume when a key is first struck. The amplitude immediately begins to die away slowly as long as the key remains depressed. If the key is released before the sound has fully died away, the amplitude will immediately drop to zero. This envelope is shown below:

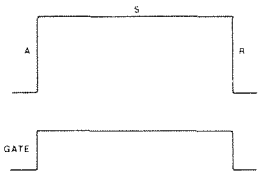
| | | |
|---|---|---|
| **ATTACK:** | **0** | **2 ms** |
| **DECAY:** | **9** | **750 ms** |
| **SUSTAIN:** | **0** | |
| **RELEASE:** | **0** | **6 ms** |




Note that the tone decays slowly until GATE is cleared, at which point the amplitude drops rapidly to zero.

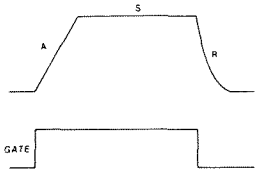
The most simple envelope is that of the organ, When a key is pressed, the tone immediately reaches full volume and remains there. When the key is released, the tone drops immediately to zero volume. This envelope is shown below:

| | | |
|---|---|---|
| **ATTACK:** | **0** | **2 ms** |
| **DECAY:** | **0** | **6 ms** |
| **SUSTAIN:** | **15 ($F)** | |
| **RELEASE:** | **0** | **6 ms** |




The real power of SID lies in the ability to create original sounds rather than simulations of acoustic instruments. The ADSR is capable of creating envelopes which do not correspond to any "real" instruments. A good example would be the "backwards" envelope. This envelope is characterized by a slow attack and rapid decay which sounds very much like an instrument that has been recorded on tape then played backwards. This envelope is shown below:

| | | |
|---|---|---|
| **ATTACK:** | **10 ($A)** | **500 ms** |
| **DECAY:** | **0** | **6 ms** |
| **SUSTAIN:** | **15 ($F)** | |
| **RELEASE:** | **3** | **72 ms** |




Many unique sounds can be created by applying the amplitude envelope of one instrument to the harmonic structure of another. This produces sounds similar to familiar acoustic instruments, yet notably different. In general, sound is quite subjective and experimentation with various envelope rates and harmonic contents will be necessary in order to achieve the desired sound.

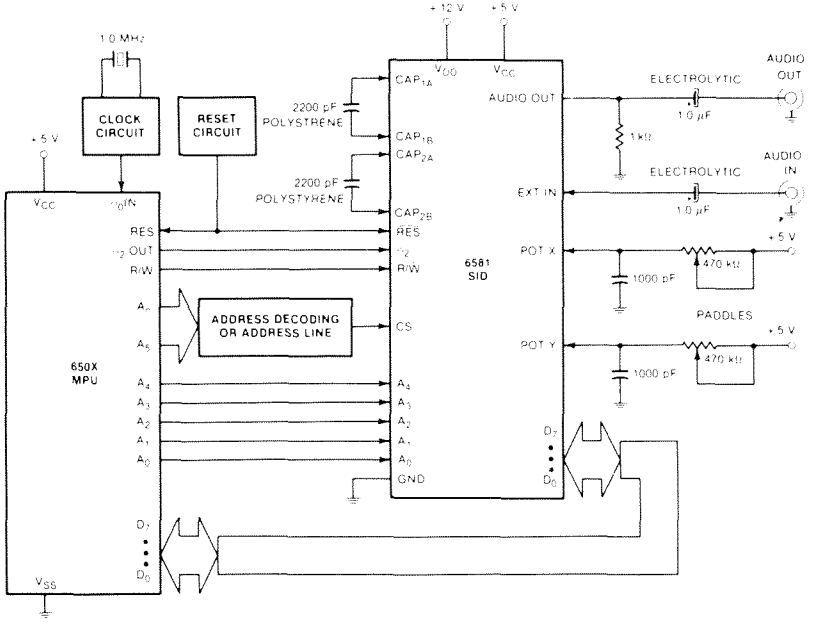



**Figure 16–18. Typical 6581/SID Application**

# 6526 COMPLEX INTERFACE ADAPTER (CIA) CHIP SPECIFICATIONS

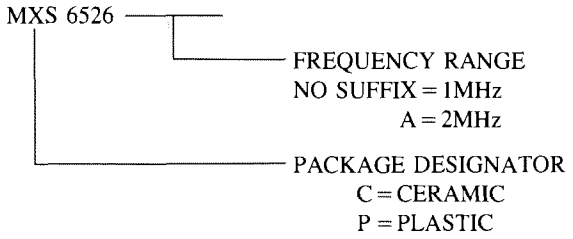
## DESCRIPTION

The 6526 Complex Interface Adapter (CIA) is an 8502 bus compatible peripheral interface device with extremely flexible timing and I/O capabilities. Figure 16–19 shows the 6526 pin configuration. Figure 16–20 shows the 6526 block diagram.

## FEATURES

- 16 Individually programmable I/O lines
- 8 or 16-bit handshaking on read or write
- 2 independent, linkable 16-bit interval timers
- 24-hour (AM/PM) time of day clock with programmable alarm
- 8-bit shift register for serial I/O
- 2TTL load capability
- CMOS compatible I/O lines
- 1 or 2 MHz operation available

## ORDERING INFORMATION



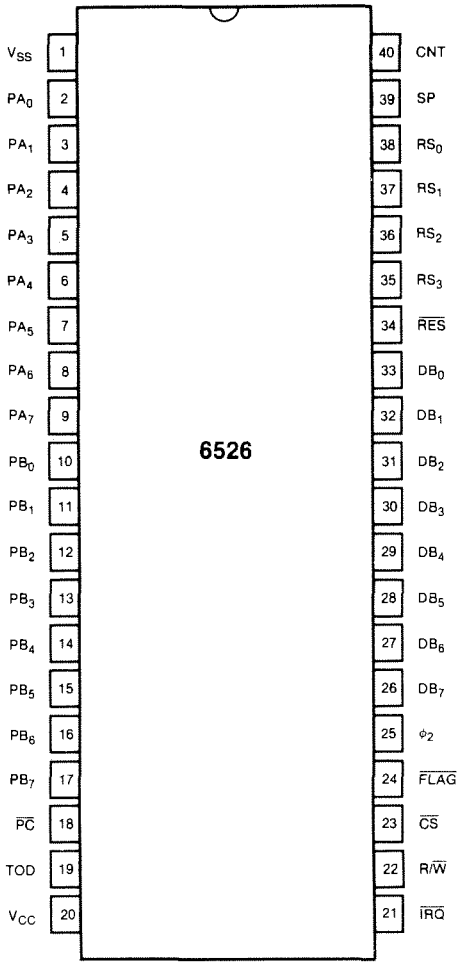



**Figure 16–19. 6526 CIA Pin Configuration**

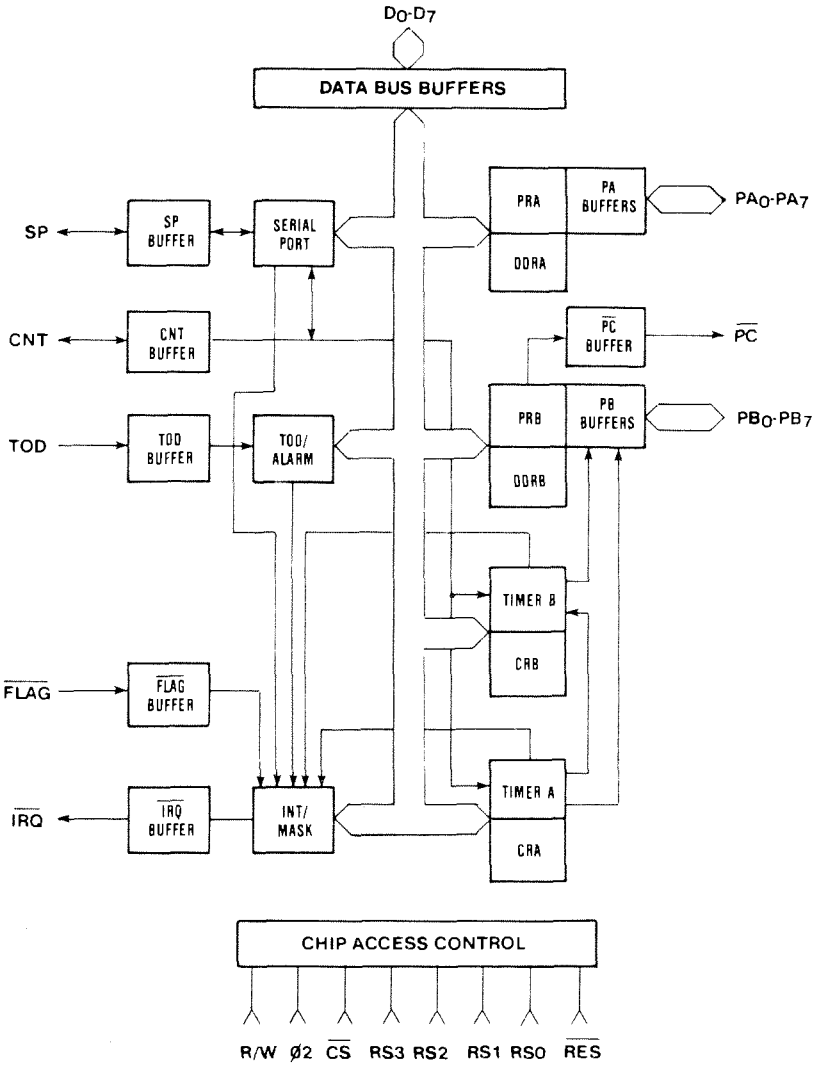



**Figure 16–20. 6526 Block Diagram**

## ABSOLUTE MAXIMUM RATINGS

| | |
|---|---|
| Supply Voltage, $V_{cc}$ | –0.3V to +7.0V |
| Input/Output Voltage, $V_{IN}$ | –0.3V to +7.0V |
| Operating Temperature, $T_{OP}$ | 0° C to 70° C |
| Storage Temperature, $T_{STG}$ | –55° C to 150° C |

All inputs contain protection circuitry to prevent damage due to high static discharges. Care should be exercised to prevent unnecessary application of voltages in excess of the allowable limits.

## COMMENT

Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. These are stress ratings only. Functional operation of this device at these or any other conditions above those indicated in the operational sections of this specification is not implied and exposure to absolute maximum rating conditions for extended periods may affect device reliability.

## ELECTRICAL CHARACTERISTICS ($V_{CC}$ ± 5%, $V_{SS}$ = 0 V, $T_A$ = 0–70°C)

| CHARACTERISTIC | SYMBOL | MIN. | TYP. | MAX. | UNIT |
|---|---|---|---|---|---|
| Input High Voltage | $V_{IH}$ | +2.4 | — | $V_{CC}$ | V |
| Input Low Voltage | $V_{IL}$ | –0.3 | — | — | V |
| Input Leakage Current; $V_{IN} = V_{SS}$ +5V ($\overline{TOD}$, R/W, $\overline{FLAG}$,$\phi$2, $\overline{RES}$, RS0–RS3, $\overline{CS}$) | $I_{IN}$ | — | 1.0 | 2.5 | μA |
| Port Input Pull-up Resistance | $R_{PI}$ | 3.1 | 5.0 | — | KΩ |
| Output Leakage Current for High Impedance State (Three State); $V_{IN}$ = 4V to 2.4V; (DB0–DB7, SP, CNT, $\overline{IRQ}$) | $I_{TSI}$ | — | ±1.0 | ±10.0 | μA |
| Output High Voltage $V_{CC}$ = MIN, $I_{LOAD}$ < –200μA (PA0–PA7, $\overline{PC}$ PB0–PB7, DB0–DB7) | $V_{OH}$ | +2.4 | — | $V_{CC}$ | V |
| Output Low Voltage $V_{CC}$ = MIN, $I_{LOAD}$ < 3.2 mA | $V_{OL}$ | — | — | +0.40 | V |
| Output High Current (Sourcing); $V_{OH}$ > 2.4V (PA0–PA7, PB0–PB7, PC, DB0–DB7 | $I_{OH}$ | –200 | –1000 | — | μA |
| Output Low Current (Sinking); $V_{OL}$ < .4V (PA0–PA7, $\overline{PC}$, PB0–PB7, DB0–DB7) | $I_{OL}$ | 3.2 | — | — | mA |
| Input Capacitance | $C_{IN}$ | — | 7 | 10 | pf |
| Output Capacitance | $C_{OUT}$ | — | 7 | 10 | pf |
| Power Supply Current | $I_{CC}$ | — | 70 | 100 | mA |

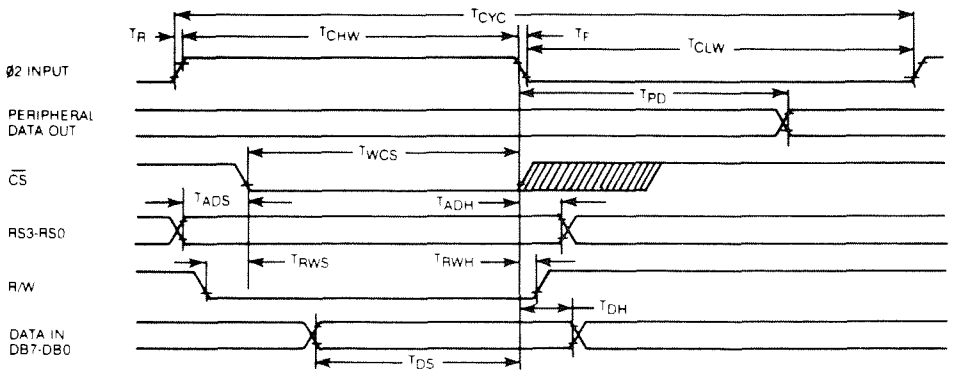



**Figure 16–21. 6526 Write Timing Diagram**

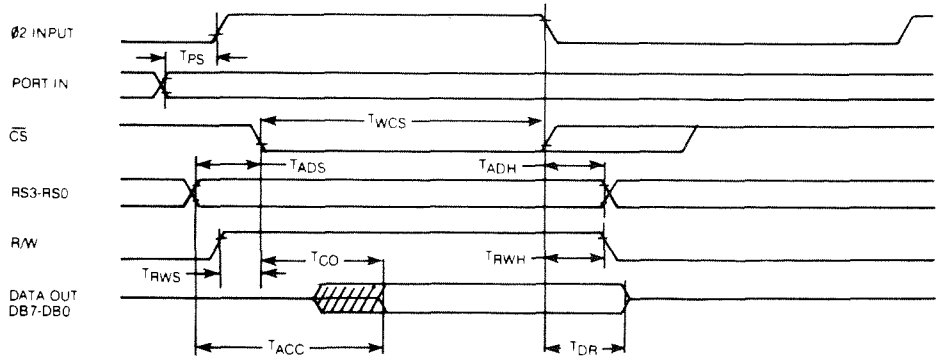



**Figure 16–22. 6526 Read Timing Diagram**

# 6526 INTERFACE SIGNALS

## ϕ2—CLOCK INPUT

The ϕ2 clock is a TTL-compatible input used for internal device operation and as a timing reference for communicating with the system data bus.

## $\overline{\text{CS}}$—CHIP SELECT INPUT

The $\overline{\text{CS}}$ input controls the activity of the 6526. A low level on $\overline{\text{CS}}$ while ϕ2 is high causes the device to respond to signals on the R/W and address (RS) lines. A high on $\overline{\text{CS}}$ prevents these lines from controlling the 6526. The $\overline{\text{CS}}$ line is normally activated (low) at ϕ2 by the appropriate address combination.

## R/W—READ/WRITE INPUT

The R/W signal is normally supplied by the microprocessor and controls the direction of data transfers of the 6526. A high on R/W indicates a read (data transfer out of the 6526), while a low indicates a write (data transfer into the 6526).

## RS3–RS0—ADDRESS INPUTS

The address inputs select the internal registers as described by the Register Map.

## DB7–DB0—DATA BUS INPUTS/OUTPUTS

The eight data bus pins transfer information between the 6526 and the system data bus. These pins are high impedance inputs unless $\overline{CS}$ is low and R/W and $\phi 2$ are high to read the device. During this read, the data bus output buffers are enabled, driving the data from the selected register onto the system data bus.

## $\overline{IRQ}$—INTERRUPT REQUEST OUTPUT

$\overline{IRQ}$ is an open drain output normally connected to the processor interrupt input. An external pullup resistor holds the signal high, allowing multiple $\overline{IRQ}$ outputs to be connected together. The $\overline{IRQ}$ output is normally off (high impedance) and is activated low as indicated in the functional description.

## $\overline{RES}$—RESET INPUT

A low on the $\overline{RES}$ pin resets all internal registers. The port pins are set as inputs and port registers to zero (although a read of the ports will return all highs because of passive pullups). The timer control registers are set to 0 and the timer latches to all ones. All other registers are reset to 0.

**6526 TIMING CHARACTERISTICS**

| | | 1 MHz | | 2MHz | | |
|---|---|---|---|---|---|---|
| SYMBOL | CHARACTERISTIC | MIN | MAX | MIN | MAX | UNIT |
| | *$\phi 2$ Clock* | | | | | |
| $T_{CYC}$ | Cycle Time | 1000 | 20,000 | 500 | 20,000 | ns |
| $T_R$, $T_F$ | Rise and Fall Time | — | 25 | — | 25 | ns |
| $T_{CHW}$ | Clock Pulse Width (High) | 420 | 10,000 | 200 | 10,000 | ns |
| $T_{CLW}$ | Clock Pulse Width (Low) | 420 | 10,000 | 200 | 10,000 | ns |
| | *Write Cycle* | | | | | |
| $T_{PD}$ | Output Delay From $\phi 2$ | — | 1000 | — | 500 | ns |
| $T_{WCS}$ | $\overline{CS}$ low while $\phi 2$ high | 420 | — | 200 | — | ns |
| $T_{ADS}$ | Address Setup Time | 0 | — | 0 | — | ns |
| $T_{ADH}$ | Address Hold Time | 10 | — | 5 | — | ns |
| $T_{RWS}$ | R/W Setup Time | 0 | — | 0 | — | ns |
| $T_{RWH}$ | R/W Hold Time | 0 | — | 0 | — | ns |
| $T_{DS}$ | Data Bus Setup Time | 150 | — | 75 | — | ns |
| $T_{DH}$ | Data Bus Hold Time | 0 | — | 0 | — | ns |

6526 TIMING CHARACTERISTICS (continued)

| | | 1 MHz | | 2MHz | | |
|---|---|---|---|---|---|---|
| SYMBOL | CHARACTERISTIC | MIN | MAX | MIN | MAX | UNIT |
| | *Read Cycle* | | | | | |
| $T_{PS}$ | Port Setup Time | 300 | — | 150 | — | ns |
| $T_{WCS}$(2) | CS low while $\phi$2 high | 420 | — | 20 | — | ns |
| $T_{ADS}$ | Address Setup Time | 0 | — | 0 | — | ns |
| $T_{AOH}$ | Address Hold Time | 10 | — | 5 | — | ns |
| $T_{RWS}$ | R/W Setup Time | 0 | — | 0 | — | ns |
| $T_{RWH}$ | R/W Hold Time | 0 | — | 0 | — | ns |
| $T_{ACC}$ | Data Access from RS3–RS0 | — | 550 | — | 275 | ns |
| $I_{CO}$(3) | Data Access from $\overline{CS}$ | — | 320 | — | 150 | ns |
| $T_{DR}$ | Data Release Time | 50 | — | 25 | — | ns |

1. All timings are referenced from $V_{IL}$ max and $V_{IH}$ min on inputs and $V_{OL}$ max and $V_{DH}$ min on outputs.
2. $T_{WCS}$ is measured from the later of $\phi$2 high or $\overline{CS}$ low. $\overline{CS}$ must be low at least until the end of $\phi$2 high.
3. $T_{CO}$ is measured from the later of $\phi$2 high or $\overline{CS}$ low. Valid data is available only after the later of $T_{ACC}$ or $T_{CO}$.

REGISTER MAP

| RS3 | RS2 | RS1 | RS0 | REG | NAME | DESCRIPTION |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | PRA | PERIPHERAL DATA REG A |
| 0 | 0 | 0 | 1 | 1 | PRB | PERIPHERAL DATA REG B |
| 0 | 0 | 1 | 0 | 2 | DDRA | DATA DIRECTION REG A |
| 0 | 0 | 1 | 1 | 3 | DDRB | DATA DIRECTION REG B |
| 0 | 1 | 0 | 0 | 4 | TA LO | TIMER A LOW REGISTER |
| 0 | 1 | 0 | 1 | 5 | TA HI | TIMER A HIGH REGISTER |
| 0 | 1 | 1 | 0 | 6 | TB LO | TIMER B LOW REGISTER |
| 0 | 1 | 1 | 1 | 7 | TB HI | TIMER B HIGH REGISTER |
| 1 | 0 | 0 | 0 | 8 | TOD 10THS | 10THS OF SECONDS REGISTER |
| 1 | 0 | 0 | 1 | 9 | TOD SEC | SECONDS REGISTER |
| 1 | 0 | 1 | 0 | A | TOD MIN | MINUTES REGISTER |
| 1 | 0 | 1 | 1 | B | TOD HR | HOURS—AM/PM REGISTER |
| 1 | 1 | 0 | 0 | C | SDR | SERIAL DATA REGISTER |
| 1 | 1 | 0 | 1 | D | ICR | INTERRUPT CONTROL REGISTER |
| 1 | 1 | 1 | 0 | E | CRA | CONTROL REG A |
| 1 | 1 | 1 | 1 | F | CRB | CONTROL REG B |

# 6526 FUNCTIONAL DESCRIPTION

## I/O PARTS (PRA, PRB, DDRA, DDRB).

Parts A and B each consist of an 8-bit Peripheral Data Register (PR) and an 8-bit Data Direction Register (DDR). If a *bit in the DDR is set to* one, the corresponding bit in the PR is an *output*; if a DDR bit is set to a 0, the corresponding PR bit is defined *as an input*. On a READ, the PR reflects the information present on the actual port pins (PA0–PA7, PB0–PB7) for both input and output bits. Port A and Port B have passive pull-up devices as well as active pull-ups, providing both CMOS and TTL compatibility. Both parts have two TTL load drive capability. In addition to normal I/O operation, PB6 and PB7 also provide timer output functions.

## HANDSHAKING

Handshaking on data transfers can be accomplished using the $\overline{PC}$ output pin and the $\overline{FLAG}$ input pin. $\overline{PC}$ will go low for one cycle following a read or write of PORT B. This signal can be used to indicate "data ready" at PORT B or "data accepted" from PORT B. Handshaking on 16-bit data transfers (using both PORT A and PORT B) is possible by always reading or writing PORT A first. $\overline{FLAG}$ is a negative edge sensitive input which can be used for receiving the $\overline{PC}$ output from another 6526, or as a general purpose interrupt input. Any negative transition of $\overline{FLAG}$ will set the $\overline{FLAG}$ interrupt bit.

| REG | NAME | $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | PRA | $PA_7$ | $PA_6$ | $PA_5$ | $PA_4$ | $PA_3$ | $PA_2$ | $PA_1$ | $PA_0$ |
| 1 | PRB | $PB_7$ | $PB_6$ | $PB_5$ | $PB_4$ | $PB_3$ | $PB_2$ | $PB_1$ | $PB_0$ |
| 2 | DDRA | $DPA_7$ | $DPA_6$ | $DPA_5$ | $DPA_4$ | $DPA_3$ | $DPA_2$ | $DPA_1$ | $DPA_0$ |
| 3 | DDRB | $DPB_7$ | $DPB_6$ | $DPB_5$ | $DPB_4$ | $DPB_3$ | $DPB_2$ | $DPB_1$ | $DPB_0$ |

## INTERVAL TIMERS (TIMER A, TIMER B)

Each interval timer consists of a 16-bit read-only Timer Counter and a 16-bit write-only Timer Latch. Data written to the timer are latched in the Timer Latch, while data read from the timer are the present contents of the Time Counter. The timers can be used independently or linked for extended operations. The various timer modes allow generation of long time delays, variable width pulses, pulse trains and variable frequency waveforms. Utilizing the CNT input, the timers can count external pulses or measure frequency, pulse width and delay times of external signals. Each timer has an associated control register, providing independent control of the following functions:

## START/STOP

A control bit allows the timer to be started or stopped by the microprocessor at any time.

## PB ON/OFF

A control bit allows the timer output to appear on a PORT B output line (PB6 for TIMER A and PB7 for TIMER B). This function overrides the DDRB control bit and forces the appropriate PB line to an output.

## TOGGLE/PULSE

A control bit selects the output applied to PORT B. On every timer underflow the output can either toggle or generate a single positive pulse of one cycle duration. The toggle output is set high whenever the timer is started and is set low by $\overline{\text{RES}}$.

## ONE-SHOT/CONTINUOUS

A control bit selects either timer mode. In one-shot mode, the timer will count down from the latched value to 0, generate an interrupt, reload the latched value, then stop. In continuous mode, the timer will count from the latched value to 0, generate an interrupt, reload the latched value and repeat the procedure continuously.

## FORCE LOAD

A strobe bit allows the timer latch to be loaded into the timer counter at any time, whether the timer is running or not.

## INPUT MODE

Control bits allow selection of the clock used to decrement the timer. TIMER A can count $\phi 2$ clock pulses or external pulses applied to the CNT pin. TIMER B can count $\phi 2$ pulses, external CNT pulses, TIMER A underflow pulses or TIMER A underflow pulses while the CNT pin is held high.

The timer latch is loaded into the timer on any timer underflow, on a force load or following a write to the high byte of the prescaler while the timer is stopped. If the timer is running, a write to the high byte will load the timer latch, but not reload the counter.

**READ (TIMER)**

| REG | NAME | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 4 | TA LO | $TAL_7$ | $TAL_6$ | $TAL_5$ | $TAL_4$ | $TAL_3$ | $TAL_2$ | $TAL_1$ | $TAL_0$ |
| 5 | TA HI | $TAH_7$ | $TAH_6$ | $TAH_5$ | $TAH_4$ | $TAH_3$ | $TAH_2$ | $TAH_1$ | $TAH_0$ |
| 6 | TB LO | $TBL_7$ | $TBL_6$ | $TBL_5$ | $TBL_4$ | $TBL_3$ | $TBL_2$ | $TBL_1$ | $TBL_0$ |
| 7 | TB HI | $TBH_7$ | $TBH_6$ | $TBH_5$ | $TBH_4$ | $TBH_3$ | $TBH_2$ | $TBH_1$ | $TBH_0$ |

**WRITE (PRESCALER)**

| REG | NAME | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 4 | TA LO | $PAL_7$ | $PAL_6$ | $PAL_5$ | $PAL_4$ | $PAL_3$ | $PAL_2$ | $PAL_1$ | $PAL_0$ |
| 5 | TA HI | $PAH_7$ | $PAH_6$ | $PAH_5$ | $PAH_4$ | $PAH_3$ | $PAH_2$ | $PAH_1$ | $PAH_0$ |
| 6 | TB LO | $PBL_7$ | $PBL_6$ | $PBL_5$ | $PBL_4$ | $PBL_3$ | $PBL_2$ | $PBL_1$ | $PBL_0$ |
| 7 | TB HI | $PBH_7$ | $PBH_6$ | $PBH_5$ | $PBH_4$ | $PBH_3$ | $PBH_2$ | $PBH_1$ | $PBH_0$ |

## TIME OF DAY CLOCK (TOD)

The TOD clock is a special purpose timer for real-time applications. TOD consists of a 24-hour (AM/PM) clock with 1/10th second resolution. It is organized into four registers: 10ths of seconds, Seconds, Minutes and Hours. The AM/PM flag is in the MSB of the Hours register for easy bit testing. Each register reads in BCD format to simplify conversion for driving displays, etc. The clock requires an external 60 Hz or 50 Hz (programmable) TTL level input on the TOD pin for accurate time keeping. In addition to timekeeping, a programmable ALARM is provided for generating an interrupt at a desired time. The ALARM registers are located at the same addresses as the corresponding TOD registers. Access to the ALARM is governed by a Control Register bit. The ALARM is write-only; any read of a TOD address will read time regardless of the state of the ALARM access bit.

A specific sequence of events must be followed for proper setting and reading of TOD. TOD is automatically stopped whenever a write to the Hours register occurs. The clock will not start again until after a write to the 10ths of seconds register. This assures TOD will always start at the desired time. Since a carry from one stage to the next can occur at any time with respect to a read operation, a latching function is included to keep all Time Of Day information constant during a read sequence. All four TOD registers latch on a read of Hours and remain latched until after a read of 10ths of seconds. The TOD clock continues to count when the output registers are latched. If only one register is to be read, there is no carry problem and the register can be read "on the fly," provided that any read of Hours is followed by a read of 10ths of seconds to disable the latching.

**READ**

| REG | NAME | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 8 | TOD 10THS | 0 | 0 | 0 | 0 | $T_8$ | $T_4$ | $T_2$ | $T_1$ |
| 9 | TOD SEC | 0 | $SH_4$ | $SH_2$ | $SH_1$ | $SL_8$ | $SL_4$ | $SL_2$ | $SL_1$ |
| A | TOD MIN | 0 | $MH_4$ | $MH_2$ | $MH_1$ | $ML_8$ | $ML_4$ | $ML_2$ | $ML_1$ |
| B | TOD HR | PM | 0 | 0 | HH | $HL_8$ | $HL_4$ | $HL_2$ | $HL_1$ |

**WRITE**

**$CRB_7 = 0$ TOD**
**$CRB_7 = 1$ ALARM**
**(SAME FORMAT AS READ)**

## SERIAL PORT (SDR)

The serial port is a buffered, 8-bit synchronous shift register system. A control bit selects input or output mode. In input mode, data on the SP pin is shifted into the shift register on the rising edge of the signal applied to the CNT pin. After eight CNT pulses, the data in the shift register is dumped into the Serial Data Register and an interrupt is generated. In the output mode, TIMER A is used for the baud rate generator. Data is

shifted out on the SP pin at one half the underflow rate of TIMER A. The maximum baud rate possible is $\phi 2$ divided by 4, but the maximum useable baud rate will be determined by line loading and the speed at which the receiver responds to input data. Transmission will start following a write to the Serial Data Register (provided TIMER A is running and in continuous mode). The clock signal derived from TIMER A appears as an output on the CNT pin. The data in the Serial Data Register will be loaded into the shift register then shift out to the SP pin when a CNT pulse occurs. Data shifted out becomes valid on the falling edge of CNT and remains valid until the next falling edge. After eight CNT pulses, an interrupt is generated to indicate more data can be sent. If the Serial Data Register was loaded with new information prior to this interrupt, the new data will automatically be loaded into the shift register and transmission will continue. If the microprocessor stays one byte ahead of the shift register, transmission will be continuous. If no further data is to be transmitted, after the 8th CNT pulse, CNT will return high and SP will remain at the level of the last data bit transmitted. SDR data is shifted out MSB first and serial input data should also appear in this format.

The bidirectional capability of the Serial Port and CNT clock allows many 6526 devices to be connected to a common serial communication bus on which one 6526 acts as a master, sourcing data and shift clock, while all other 6526 chips act as slaves. Both CNT and SP outputs are open drain to allow such a common bus. Protocol for master/slave selection can be transmitted over the serial bus, or via dedicated handshaking lines.

| REG | NAME | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| C | SDR | $S_7$ | $S_6$ | $S_5$ | $S_4$ | $S_3$ | $S_2$ | $S_1$ | $S_0$ |

## INTERRUPT CONTROL (ICR)

There are five sources of interrupts on the 6526: underflow from TIMER A, underflow from TIMER B, TOD ALARM, Serial Port full/empty and FLAG. A single register provides masking and interrupt information. The interrupt Control Register consists of a write-only MASK register and a read-only DATA register. Any interrupt will set the corresponding bit in the DATA register. Any interrupt which is enabled by the MASK register will set the IR bit (MSB) of the DATA register and bring the IRQ pin low. In a multi-chip system, the IR bit can be polled to detect which chip has generated an interrupt request. The interrupt DATA register is cleared and the IRQ line returns high following a read of the DATA register. Since each interrupt sets an interrupt bit regardless of the MASK, and each interrupt bit can be selectively masked to prevent the generation of a processor interrupt, it is possible to intermix polled interrupts with true interrupts. However, polling the IR bit will cause the DATA register to clear, therefore, it is up to the user to preserve the information contained in the DATA register if any polled interrupts were present.

The MASK register provides convenient control of individual mask bits. When writing to the MASK register, if bit 7 (SET/CLEAR) of the data written is a ZERO, any

mask bit written with a one will be cleared, while those mask bits written with a 0 will be unaffected. If bit 7 of the data written is a ONE, any mask bit written with a one will be set, while those mask bits written with a 0 will be unaffected. In order for an interrupt flag to set IR and generate an Interrupt Request, the corresponding MASK bit must be set.

**READ (INT DATA)**

| REG | NAME | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **D** | **ICR** | **IR** | **0** | **0** | **FLG** | **SP** | **ALRM** | **TB** | **TA** |

**WRITE (INT MASK)**

| REG | NAME | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **D** | **ICR** | **S/$\overline{C}$** | **X** | **X** | **FLG** | **SP** | **ALRM** | **TB** | **TA** |

## CONTROL REGISTERS

There are two control registers in the 6526, CRA and CRB. CRA is associated with TIMER A and CRB is associated with TIMER B. The register format is as follows:

### CRA:

| BIT | NAME | FUNCTION |
|---|---|---|
| 0 | START | 1 = START TIMER A, 0 = STOP TIMER A, This bit is automatically reset when underflow occurs during one-shot mode. |
| 1 | PBON | 1 = TIMER A output appears on PB6, 0 = PB6 normal operation. |
| 2 | OUTMODE | 1 = TOGGLE, 0 = PULSE |
| 3 | RUNMODE | 1 = ONE-SHOT, 0 = CONTINUOUS |
| 4 | LOAD | 1 = FORCE LOAD (this is a STROBE input, there is no data storage, bit 4 will always read back a 0 and writing a 0 has no effect). |
| 5 | INMODE | 1 = TIMER A counts positive CNT transitions, 0 = TIMER A counts $\phi$2 pulses. |
| 6 | SPMODE | 1 = SERIAL PORT output (CNT sources shift clock), 0 = SERIAL PORT input (external shift clock required). |
| 7 | TODIN | 1 = 50 Hz clock required on TOD pin for accurate time, 0 = 60 Hz clock required on TOD pin for accurate time. |

## CRB:

| BIT | NAME | FUNCTION |
|---|---|---|
| | | (Bits CRB0–CRB4 are identical to CRA0–CRA4 for TIMER B with the exception that bit 1 controls the output of TIMER B on PB7). |
| 5,6 | INMODE | Bits CRB5 and CRB6 select one of four input modes for TIMER B as:<br>CRB6 CRB5<br>0 0 TIMER B counts $\phi 2$ pulses.<br>0 1 TIMER B counts positive CNT transitions.<br>1 0 TIMER B counts TIMER A underflow pulses.<br>1 1 TIMER B counts TIMER A underflow pulses while CNT is high. |
| 7 | ALARM | 1 = writing to TOD registers sets ALARM, 0 = writing to TOD registers sets TOD clock. |

| REG | NAME | TOD IN | SP MODE | IN MODE | LOAD | RUN MODE | OUT MODE | PS ON | START |
|---|---|---|---|---|---|---|---|---|---|
| E | CRA | 0 = 60Hz<br>1 = 50Hz | 0 = INPUT<br>1 = OUTPUT | 0 = $\phi 2$<br>1 = CNT | 1 = FORCE LOAD (STROBE) | 0 = CONT<br>1 = 0.5 | 0 = PULSE<br>1 = TOGGLE | 0 = $PB_6$OFF<br>1 = $PB_6$ ON | 0 = STOP<br>1 = START |

TA

| REG | NAME | ALARM | IN | MODE | LOAD | RUN MODE | OUT MODE | PS ON | START |
|---|---|---|---|---|---|---|---|---|---|
| F | CRS | 0 = TOD<br>1 = ALARM | 0<br>1<br>1<br>1 | 0 = $\phi 2$<br>1 = CNT<br>0 = TA<br>1 = CNT TA | 1 = FORCE LOAD (STROBE) | 0 = CONT.<br>1 = O.S. | 0 = PULSE<br>1 = TOGGLE | 0 = $PS_7$OFF<br>1 = $PS_7$ON | 0 = STOP<br>1 = START |

TB

All unused register bits are unaffected by a write and are forced to zero on a read.

# DYNAMIC RANDOM ACCESS MEMORY

This section discusses the characteristics of the C128 dynamic RAMs, which are currently the 4164 64K-bit RAM. This RAM device is in the 64K by 1-bit configuration. This section also contains information on the 4416 dynamic RAMs used by the 8563 video controller. This type of RAM is a 64K RAM in the 16K by 4 bit configuration.

## SYSTEM RAM DESCRIPTION

The C128 system contains 128K of processor-addressable DRAM in the 64K by 1 configuration, organized into two individual 64K banks. Additionally, the system contains 16K of video display DRAM local to the 8563 video controller, not directly accessable by the processor.

RAM banking, described in detail in the MMU section in Chapter 13, is controlled by several MMU registers: the Configuration Register, the RAM Configuration Register, and the Zero Page and Page One Pointers. Simply put, the Configuration Register controls which 64K bank of RAM is selected; the RAM Configuration Register controls if and how much RAM is kept in common between banks, and the Pointer Registers redirect the zero and one pages to any page in memory, overriding the effect of the two Configuration Registers. In the system, RAM bank select is achieved via gated CAS control.

## PHYSICAL CHARACTERISTICS

This section covers some of the characteristics of the 64K by 1-bit RAM and 16K by 4-bit RAM that are used in the C128 system. A pinout table and a figure are given for both the 4164 and the 4416 packages (See Tables 16–15 and 16–16 and Figures 16–23 and 16–24).

| PIN | NAME | DESCRIPTION |
|---|---|---|
| 1 | NC | No Connection |
| 2 | $D_{in}$ | Data In |
| 3 | /WE | Write Enable (Active Low) |
| 4 | /RAS | Row Address Strobe (Active Low) |
| 5 | $A_0$ | Address Bit 0 |
| 6 | $A_2$ | Address Bit 2 |
| 7 | $A_1$ | Address Bit 1 |
| 8 | $V_{CC}$ | Power Supply +5 Vdc |
| 9 | $A_7$ | Address Bit 7 |
| 10 | $A_5$ | Address Bit 5 |
| 11 | $A_4$ | Address Bit 4 |
| 12 | $A_3$ | Address Bit 3 |
| 13 | $A_6$ | Address Bit 6 |
| 14 | $D_{out}$ | Data Out |
| 15 | /CAS | Column Address Strobe (Active Low) |
| 16 | $V_{ss}$ | Power Supply Ground |

**Table 16–15. 4164 Pinout**

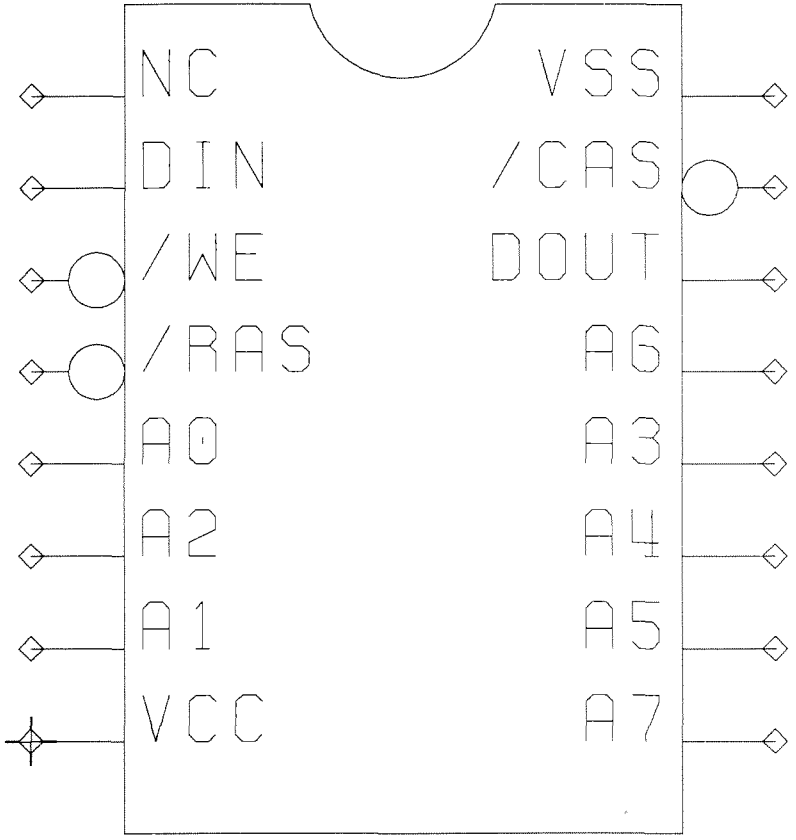



**Figure 16–23. The 4164 Chip**

| PIN | NAME | DESCRIPTION |
|---|---|---|
| 1 | /OE | Output Enable (Active Low) |
| 2 | $D_1$ | Data Bit 1 |
| 3 | $D_2$ | Data Bit 2 |
| 4 | /WE | Write Enable (Active Low) |
| 5 | /RAS | Row Address Strobe (Active Low) |
| 6 | $A_6$ | Address Bit 6 |
| 7 | $A_5$ | Address Bit 5 |
| 8 | $A_4$ | Address Bit 4 |
| 9 | $V_{dd}$ | Power Supply +5 Vdc |
| 10 | $A_7$ | Address Bit 7 |
| 11 | $A_3$ | Address Bit 3 |
| 12 | $A_2$ | Address Bit 2 |
| 13 | $A_1$ | Address Bit 1 |
| 14 | $A_0$ | Address Bit 0 |
| 15 | $D_3$ | Data Bit 3 |
| 16 | /CAS | Column Address Strobe (Active Low) |
| 17 | $D_4$ | Data Bit 4 |
| 18 | Vss | Power Supply Ground |

**Table 16–16. 4416 Pinout**

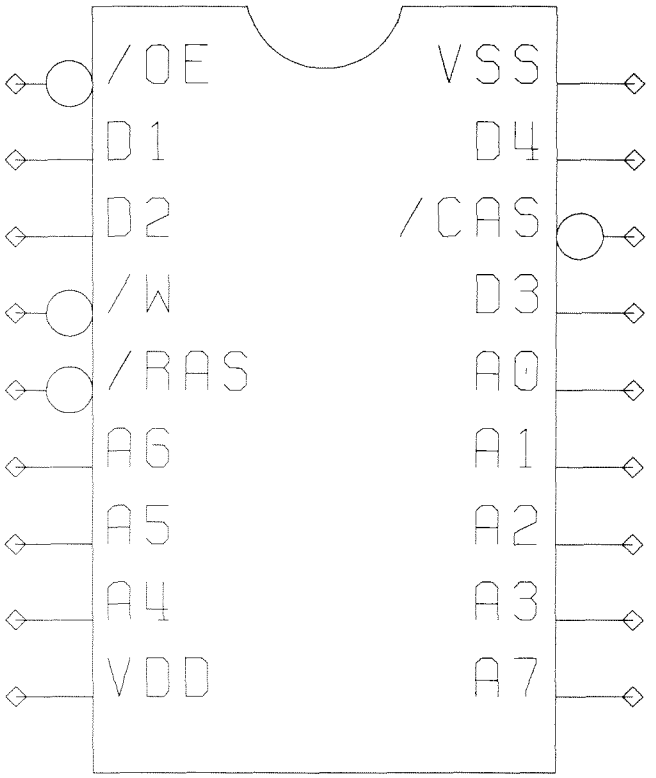



**Figure 16–24. The 4464 Chip**

# READ ONLY MEMORY (ROM)

This section describes the C128 system ROM, both from a logical and a hardware point of view. It mentions aspects of the ROM banking structure, the management of Kernal and BASIC, and explains the physical specifications of the ROM devices themselves.

## SYSTEM ROM DESCRIPTION

In C64 mode, the operating system resides in 16K of ROM, which includes approximately 8K for Kernal and 8K for BASIC. In C128 mode, the operating system resides in 48K of ROM and includes advanced Kernal and BASIC features. The Kernal, by definition, is the general operating system of the computer, with fixed entry points into usable subroutines to facilitate ROM update transparent use by higher-level programs. There is also a character ROM that resides on the Shared Bus, shared by the VIC chip and the processor. This ROM is a 4016 8K by 8, NMOS ROM. The C64 OS ROM is wired so as to appear as two chunks of noncontiguous ROM, copying the actual C64 ROM memory map. Provision is included to handle system ROM as either four 16K × 8 ROMs or as two 32K × 8 ROMs. All internal C128 function ROMs will be the 32K × 8 variety.

## ROM BANKING

Refer back to the MMU register map, Figure 13–4 in Chapter 13. Note that the Configuration Register (CR) controls the type of ROM or RAM seen in a given address location. Dependent on the contents of the CR, ROM may be enabled and disabled to attain the most useful configuration for the application at hand. ROM is enabled in three memory areas in C128 mode, each consisting of 16K of address space. The lower ROM may be defined as RAM or System ROM, the upper two ROMs may be System ROM, Function ROM, Cartridge ROM or RAM. In C64 mode, the C64 memory mapping rules apply, which are primitive compared to those used in C128 mode. C64 ROM is banked as two 8K sections, BASIC and Kernal, according to the page zero port and the cartridge in place at the time. No free banking can occur when a cartridge is in place.

In the C128, if an address falls into the range of an enabled ROM, the MMU will communicate the status of ROM to the PLA decoder via the memory status lines. Essentially, the MMU looks up in the Configuration Register which ROM or RAM is set. See Chapter 13. The way the banking scheme is implemented, it allows up to 32K of internal, bankable ROM for use in such programs as Function Key Applications, and will support 32K of internal bankable ROM. Various combinations of ROM are possible, and can be noted by studying the configurations for the Configuration Register. Type 23128 (16K by 8) and 23256 (32K by 8) ROMs are used by the system.

# TIMING SPECIFICATION

## INTERNAL ROMs

This section specifies timing parameters for both the 23128 and the 27256 Read Only Memories. This timing spec applies to internal ROMs and for external ROMs run at 1 MHz. For external ROMs run at 2 MHz, see Table 16–18.

| PARAMETER | SYMBOL | MIN | MAX | UNIT |
|---|---|---|---|---|
| **Address Valid to Output Delay** | **$T_{ACC}$** | **300** | **—** | **ns** |
| **Chip Enable to Output Delay** | **$T_{CE}$** | **300** | **—** | **ns** |
| **Output Enable to Output Delay** | **$T_{OE}$** | **120** | **—** | **ns** |

**Table 16–17. Internal ROM Timing**

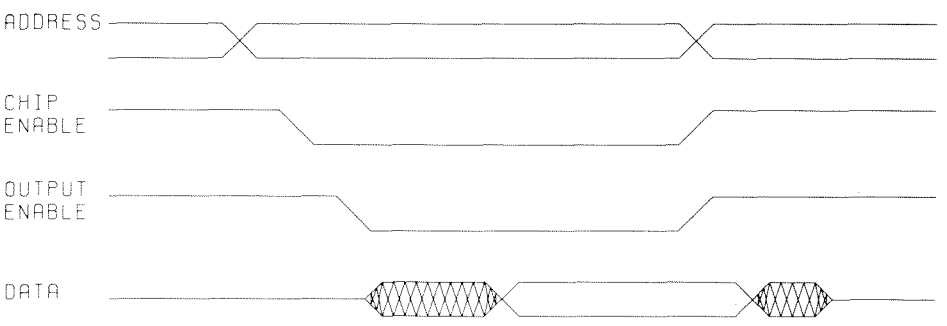



**Figure 16–25. ROM Timing Diagram**

## EXTERNAL ROMS

All C64 mode external ROMs and many C128 mode external ROMs can be of the type mentioned above. Any external ROM that is to run at 2 MHz must be faster, as specified in Table 16–18.

| PARAMETER | SYMBOL | MIN | MAX | UNIT |
|---|---|---|---|---|
| **Address Valid to Output Delay** | **$T_{ACC}$** | **250** | **—** | **ns** |
| **Chip Enable to Output Delay** | **$T_{CE}$** | **200** | **—** | **ns** |
| **Output Enable to Output Delay** | **$T_{0E}$** | **100** | **—** | **ns** |

**Table 16–18. External 2MHz ROM Timing**

# THE 23128 ROM

| PIN | NAME | DESCRIPTION |
|---|---|---|
| 1 | $V_{pp}$ | Programming Voltage |
| 2 | $A_{12}$ | Address Bit 12 ($A_{13}$ on the C64 OS ROM) |
| 3 | $A_7$ | Address Bit 7 |
| 4 | $A_6$ | Address Bit 6 |
| 5 | $A_5$ | Address Bit 5 |
| 6 | $A_4$ | Address Bit 4 |
| 7 | $A_3$ | Address Bit 3 |
| 8 | $A_2$ | Address Bit 2 |
| 9 | $A_1$ | Address Bit 1 |
| 10 | $A_0$ | Address Bit 0 |
| 11 | $D_0$ | Data Bit 0 |
| 12 | $D_1$ | Data Bit 1 |
| 13 | $D_2$ | Data Bit 2 |
| 14 | GND | Power Supply Ground |
| 15 | $D_3$ | Data Bit 3 |
| 16 | $D_4$ | Data Bit 4 |
| 17 | $D_5$ | Data Bit 5 |
| 18 | $D_6$ | Data Bit 6 |
| 19 | $D_7$ | Data Bit 7 |
| 20 | /CE | Chip Enable (Active Low) |
| 21 | $A_{10}$ | Address Bit 10 |
| 22 | /OE | Output Enable (Active Low) |
| 23 | $A_{11}$ | Address Bit 11 |
| 24 | $A_9$ | Address Bit 9 |
| 25 | $A_8$ | Address Bit 8 |
| 26 | $A_{13}$ | Address Bit 13 |
| 27 | /PGM | Program Enable (Active Low) |
| 28 | $V_{cc}$ | Power Supply +5 Vdc |

**Table 16–19. 23128 ROM Pinout**

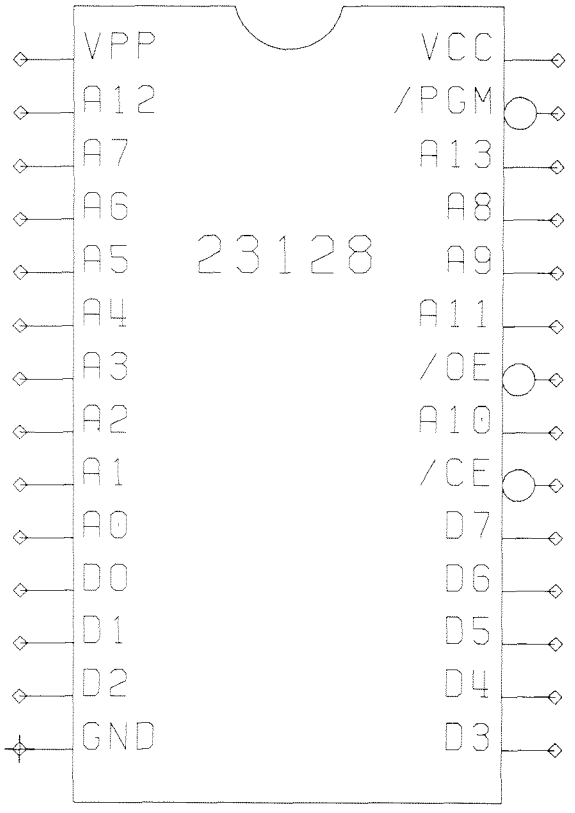



**Figure 16–26. The 23128 ROM Chip (BASIC, Kernal, Editor and External Function ROMs)**

# THE 23256 ROM

| PIN | NAME | DESCRIPTION |
|---|---|---|
| 1 | $V_{pp}$ | Programming Voltage |
| 2 | $A_{12}$ | Address Bit 12 |
| 3 | $A_7$ | Address Bit 7 |
| 4 | $A_6$ | Address Bit 6 |
| 5 | $A_5$ | Address Bit 5 |
| 6 | $A_4$ | Address Bit 4 |
| 7 | $A_3$ | Address Bit 3 |
| 8 | $A_2$ | Address Bit 2 |
| 9 | $A_1$ | Address Bit 1 |
| 10 | $A_0$ | Address Bit 0 |
| 11 | $D_0$ | Data Bit 0 |
| 12 | $D_1$ | Data Bit 1 |
| 13 | $D_2$ | Data Bit 2 |
| 14 | GND | Power Supply Ground |
| 15 | $D_3$ | Data Bit 3 |
| 16 | $D_4$ | Data Bit 4 |
| 17 | $D_5$ | Data Bit 5 |
| 18 | $D_6$ | Data Bit 6 |
| 19 | $D_7$ | Data Bit 7 |
| 20 | /CE–/PGM | Chip Enable–Program Enable (Active Low) |
| 21 | $A_{10}$ | Address Bit 10 |
| 22 | /OE | Output Enable (Active Low) |
| 23 | $A_{11}$ | Address Bit 11 |
| 24 | $A_9$ | Address Bit 9 |
| 25 | $A_8$ | Address Bit 8 |
| 26 | $A_{13}$ | Address Bit 13 |
| 27 | $A_{14}$ | Address Bit 14 |
| 28 | $V_{cc}$ | Power Supply +5 Vdc |

**Table 16–20. 23256 ROM Pinout (Internal or External Function ROMs)**

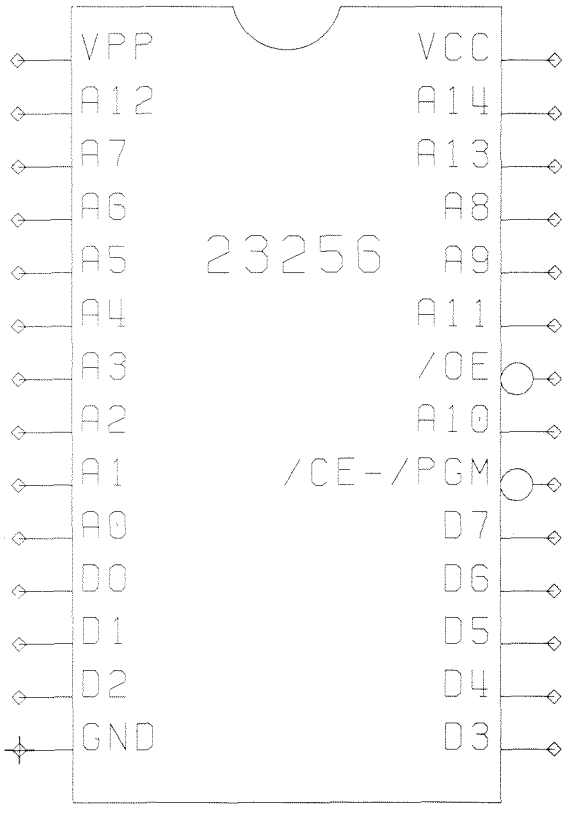



**Figure 16–27. The 23256 ROM Chip**

# THE SERIAL BUS

The C128 Serial Bus is an improved version of the C64 serial bus. This bus uses IEEE-488–type addressing, thus maintaining software compatibility between the consumer market serial devices and the CBM IEEE-488 parallel devices. The C128 improves this bus by allowing communication at much greater speeds with specially designed peripherals, the most important being the disk drive, while still maintaining compatibility with older, slower peripherals used by the C64. This section describes the hardware and some of the software aspects of both the old and the new serial transmission scheme.

## BUS OPERATIONS

There are three basic bus operations that take place on the serial bus, in both fast and slow modes. The first of these is called **Control.** The C128 is the controller in most circumstances. The controller of the bus is always the device that initiates protocol on the bus, requesting peripheral devices to do one of the two other serial operations, either **Talk** or **Listen.**

All serial bus devices can listen. A listening device is a device that has been ordered by the controller to receive data. Some devices, such as disk drives, can talk. A talking device is sending data to the controller. Both hardware and software drive this bus protocol.

## BUS SIGNALS

The Commodore serial bus is composed of the following signals:

- SRQ (pin 1): This signal is called **Service Request.** The slow serial bus does not use this line, the fast bidirectional clock line. (Not used in C64 Mode.)
- GND (pin 2): Chassis ground.
- ATN (pin 3): This signal is called **Attention.** It is used to address a device on the bus. When requesting a device to talk to or listen, the controller brings this signal low, creating some sort of interrupt on all serial bus devices. It then sends out an address that will select one device on the bus. It is the controller's responsibility to time out if a device on the bus does not respond within a reasonable amount of time.
- CLK (pin 4): This is the slow serial clock. It is used by slow serial devices, which are software-clocked, to clock data transmitted on the serial bus.
- DATA (pin 5): This is the serial data line. It is used by both slow and fast serial devices to transmit data in sync with clock signal.
- RESET (pin 6): This is the reset line, used to reset all peripherals when the host resets.

# FAST SERIAL BUS

## FAST PROTOCOL

To function as a fast talker, the system must be addressing a fast device, such as the 1571 disk drive. When addressing any device, the C128 sends a fast byte, toggling the SQR line eight times, with the ATN line low. If the device is a fast device, it will record the fact that a fast controller accessed it and respond with a fast acknowledge. If the device is a slow device, no response is delivered and the C128 then assumes it is talking with a slow device. The status of drive speed is retained until the device is requested to untalk or unlisten, if an error occurs, or if a system reset occurs.

## FAST HARDWARE

The fast serial bus, in order to achieve its speed increase, uses different hardware from that of the slow serial bus. The slow serial bus uses several 6526 port lines to drive ATN, CLK and DATA. Thus, clocking of the data transfers must be software-driven. The fast serial method is to use the serial port line of a 6526 (CIA-1) to actually transfer the serial data. This increases the transfer rate dramatically, to the point where the transfer becomes limited more by software overhead than anything else. The actual speed of transfer is set by the 6526 timer. Current 6526's have a minimum serial timer value of 4, though in actual use this value is closer to 6, owing to loading. Any advances in the 6526 would make a faster data transfer possible.

This scheme could interfere with slow serial transmissions, since the DATA line is shared by both schemes. Thus, circuitry exists that will isolate the fast serial drivers from the slow serial bus. Setting FSDIR to input mode is sufficient to remove any possible fast serial interactions with the slow serial bus, other than the additional device loading, which is not a problem at slow serial bus speeds.

In order to ensure compatibility with the C64, however, the slow serial bus cannot interfere with the fast drivers, since these drivers are shared with the User Port and a user port device could presumably make use of them. Once C64 mode is set, the input direction of the interface circuitry is disabled. Thus, in C64 mode, the FSDIR bit must be set to input to remove fast to slow interference, but slow to fast interference is automatically removed by invoking C64 mode. There is no way to disable slow to fast interference in C128 mode (at least not simultaneously with the elimination of fast to slow interference).

# THE EXPANSION BUS

The C128 Expansion Bus is compatible with the C64 Expansion Bus, while at the same time allowing extended capabilities in C128 mode.

## CARTRIDGE ADDITION

The C128 can use larger and more sophisticated cartridges than the C64. One of the main reasons for this is the new banking scheme used in the C128 for external cartridges. The C64 uses two hardware control lines, /EXROM and /GAME, to control banking out of internal facilities and banking in of cartridge facilities. The C128 uses a software polling method, where upon power-up it polls the cartridge, according to a defined protocol, to determine if such a cartridge exists, and if so, what its software priority is. Since the C128 is always free to bank between cartridges and built-in ROM, an external application can take advantage of internal routines and naturally become an extended part of the C128, as opposed to becoming a replacement application. See Chapter 13 for information on the Auto Start Cartridge ROM sequence.

The elimination of /EXROM and /GAME as hardware control lines for cartridge identification (in C128 mode) has freed up both of these lines for extended functioning. Both of the lines appear as bits in the MMU Mode Configuration Register, and are both input and output ports. Neither has a dedicated function other than general cartridge function expansion, and lend themselves to act as latched banking lines or input sense lines. Of course, neither can be asserted on C128 power-up or C64 mode will automatically be set.

## DMA CAPABILITY

The C128 expansion bus supports DMAs in a fashion similar to that of the C64. A C64 DMA is achieved by pulling the /DMA pin on the expansion bus low. Immediately after this happens, the RDY and AEC lines of the processor are brought low. This can neatly shut down the processor, but it can also cause problems, depending on what the processor is doing at the time. The RDY input of an 8502 series processor, when brought low, will halt the processor of the next Φ1 cycle, leaving the processor's address lines reflecting the current address being fetched. However, if the processor is in a write cycle when RDY is brought low, it will ignore RDY until the next read cycle. Thus, in the C64, a /DMA input occurring during a write cycle will tri-state the processor's address and data bus, but not stop it until up to three cycles later when the next read cycle occurs. The write cycles following the /DMA input do not actually write, causing memory corruption and often processor fatality when the /DMA line is released. Any /DMA input during Φ2 is a potentially fatal DMA.

If a proper /DMA is asserted, the C64 tri-states and shuts down, allowing the DMA source complete access to the processor bus. Such a DMA source must monitor the Φ2 and BA outputs, as it must tri-state when the VIC is on the bus, and it must completely DMA when a VIC DMA is called for. The VIC chip always has the highest

DMA priority. When on the bus, the DMA source has access to RAM, ROM and I/O in the C64 scheme. A proper DMA shutdown is usually achieved via some C64 software handshaking with the DMA source.

The C128 system uses a similar DMA scheme. When the /DMA input goes low, the RDY input to the 8502, the AEC input to the 8502, and the /BUSRQST input to the Z80 immediately go low. Additionally, the gated AEC signal, GAEC, goes low, causing the MMU to go immediately to its VIC CYCLE MODE, and the Z80 Data Out buffer to tri-state. The DMA causes the Address to the Shared Address buffer to reverse direction, and the Translated Address to the Address buffer to be enabled, giving the external DMA source complete access to the processor address bus. The PLA is still looking at ungated AEC and as such will allow access to I/O devices, RAM and ROM. There can be no access to the MMU; thus for C128 memory mapping the memory map must be set up before being DMA'ed. For C64 mode, memory mapping is done by the 8502 processor port lines and by the external /EXROM and /GAME. Since the 8502 ports will be inaccessible by a DMA source, only the C64 map changes based upon /EXROM and /GAME can be made during a DMA. This is the same as in a C64 unit. All DMA sources, as with the C64, must yield to the VIC during Φ0 or BA low. The C128 can perform a destructive DMA as easily as the C64. In order to use DMA's, the DMA source will most likely have to cooperate with a C128 mode program, allowing it to handshake with a DMA source to effect DMA's nondestructively.

# EXPANSION BUS PINOUT

| PIN | NAME | DESCRIPTION |
|---|---|---|
| 1 | GND | System Ground |
| 2 | +5V | System $V_{cc}$ |
| 3 | +5V | System $V_{cc}$ |
| 4 | /IRQ | Interrupt Request |
| 5 | R/W | System Read Write Signal |
| 6 | DCLOCK | 8.18 MHz Video Dot Clock |
| 7 | $I/O_1$ | I/O Chip Select: $DE00–$DEFF, Active Low |
| 8 | /GAME | Sensed for Memory Map Configuration |
| 9 | /EXROM | Sensed for Memory Map Configuration |
| 10 | $I/O_2$ | I/O Chip Select: $DF00–$DFFF, Active Low. |
| 11 | $/ROM_L$ | External ROM Chip Select, $8000–$BFFF in C128 Mode ($8000–$9FFF in C64 mode) |
| 12 | BA | Bus Available Output |
| 13 | /DMA | Direct Memory Access Input (see caution on DMA capability on p. 636) |
| 14 | $D_7$ | Data Bit 7 |
| 15 | $D_6$ | Data Bit 6 |
| 16 | $D_5$ | Data Bit 5 |
| 17 | $D_4$ | Data Bit 4 |
| 18 | $D_3$ | Data Bit 3 |
| 19 | $D_2$ | Data Bit 2 |
| 20 | $D_1$ | Data Bit 1 |
| 21 | $D_0$ | Data Bit 0 |
| 22 | GND | System Ground |
| A | GND | System Ground |
| B | $/ROM_H$ | External ROM Chip Select, $C000–$FFFF in C128 Mode ($C000–$FFFF in C64 mode) |
| C | /RESET | System Reset Signal |
| D | /NMI | Non-Maskable Interrupt Request |
| E | 1MHz | System 1MHz $\phi_0$ Clock |
| F | $TA_{15}$ | Translated Address Bit 15 |
| H | $TA_{14}$ | Translated Address Bit 14 |
| J | $TA_{13}$ | Translated Address Bit 13 |
| K | $TA_{12}$ | Translated Address Bit 12 |
| L | $TA_{11}$ | Translated Address Bit 11 |
| M | $TA_{10}$ | Translated Address Bit 10 |
| N | $TA_9$ | Translated Address Bit 9 |
| P | $TA_8$ | Translated Address Bit 8 |
| R | $SA_7$ | Shared Address Bit 7 |
| S | $SA_6$ | Shared Address Bit 6 |
| T | $SA_5$ | Shared Address Bit 5 |
| U | $SA_4$ | Shared Address Bit 4 |
| V | $SA_3$ | Shared Address Bit 3 |
| W | $SA_2$ | Shared Address Bit 2 |
| X | $SA_1$ | Shared Address Bit 1 |
| Y | $SA_0$ | Shared Address Bit 0 |
| Z | GND | System Ground |

**Table 16–21. Expansion Bus Pinout**

# THE VIDEO INTERFACE

The C128 VIC video interface hardware allows the connection of a standard NTSC or PAL commercial television and/or a color monitor. The monitor can accept either a composite video signal or separate chroma and luminance/sync signals in addition to an audio signal. This output is very similar to the output in later revision C64 units.

The C128 also provides 80-column video interfacing. The available 80-column display is RGBI and monochrome, able to interface to most NTSC- or PAL-compatible RGBI TYPE I monitors and most 80-column–compatible NTSC or PAL monochrome monitors.

# THE VIC VIDEO INTERFACE

The following items specify the VIC video interface for 40-column display in sixteen colors. The VIC signal is available at analog levels at the video connector and at RF levels at the RF output.

## MODULATOR SPECIFICATION

The modulator provides a broadcast-type RF signal carrying the VIC composite video and audio signals. The NTSC modulator is switchable between channels 3 and 4 to help minimize local broadcast interference. The signal generated by the RF modulator complies with the FCC ruling concerning FCC Class B, TV interface devices. The RF output is accessible via a standard RCA-type phono/video jack.

## MONITOR OUTPUT

The VIC video output provides the signals shown in Table 16–22.

| SIGNAL | LEVEL | IMPEDENCE | DC OFFSET |
|---|---|---|---|
| **Luminance/Sync** | **1 V p-p** | **75 Ω** | **0.5 V** |
| **Chroma** | **1 V p-p** | **75 Ω** | **0.5 V** |
| **Composite** | **1 V p-p** | **75 Ω** | **0.5 V** |
| **Audio** | **1 V p-p** | **1K Ω** | |

**Table 16–22. VIC Video Output Signals**

## VIDEO CONNECTOR PINOUT

The VIC video connector exists physically as an eight-pin DIN connector. It provides the signals shown in Table 16–23.

| PIN | SIGNAL |
|---|---|
| 1 | Luminance/Sync |
| 2 | Ground |
| 3 | Audio Out |
| 4 | Composite |
| 5 | Audio In |
| 6 | Chroma |
| 7 | N.C. |
| 8 | N.C. |

**Table 16–23. Video Connector Signals**

# THE 8563 VIDEO INTERFACE

The following items specify the 8563 video interface for 80-column display in sixteen colors. The 8563 signal is available at digital levels for RGBI and at a three-level derived analog for black and white composite video.

## MONITOR OUTPUT

Table 16–24 shows the signals provided by the 8563 output.

| SIGNAL | LEVEL | IMPEDENCE |
|---|---|---|
| Red | TTL | TTL |
| Green | TTL | TTL |
| Blue | TTL | TTL |
| Intensity | TTL | TTL |
| HSync | TTL | TTL |
| VSync | TTL | TTL |
| Composite | | 75 Ω |
| Full Intensity | 2.0 V | |
| Half Intensity | 1.5 V | |
| Sync | 0.5 V | |

**Table 16–24. 8563 Output Signals**

## VIDEO CONNECTOR PINOUT

The 8563 video connector is an IBM-style D9 connector, providing the signals shown in Table 16–25.

| PIN | SIGNAL |
|---|---|
| 1 | Ground |
| 2 | Ground |
| 3 | Red |
| 4 | Green |
| 5 | Blue |
| 6 | Intensity |
| 7 | Monochrome (non-standard) |
| 8 | Horizontal Sync |
| 9 | Vertical Sync |

**Table 16–25. 8563 Video Connector Pinout**

# THE KEYBOARD

The C128 Keyboard is an advance over the standard C64 keyboard, while still maintaining full compatibility. It has several extra keys that are used in C128 mode, but not in C64 mode. It features a numeric keypad, a **HELP** key, extended function keys, a true **CAPS LOCK** key, and a **40/80** column switch key, all of which are strobed by the VIC chip or tied to dedicated 8502 or MMU I/O lines.

## CONNECTOR PINOUT

The C128 keyboard is designed to be connected by one 12- and one 13-pin internal single-in-line connector for the unit with a built-in keyboard. Table 16–26 illustrates both connections.

| D-TYPE | SIGNAL |
|---|---|
| 1 | Ground |
| 2 | Key |
| 3 | Restore |
| 4 | +5V |
| 5 | Row 3 |
| 6 | Row 6 |
| 7 | Row 5 |
| 8 | Row 4 |
| 9 | Row 7 |
| 10 | Row 2 |
| 11 | Row 1 |
| 12 | Row 0 |
| 13 | Column 0 |
| 14 | Column 6 |
| 15 | Column 5 |
| 16 | Column 4 |
| 17 | Column 3 |
| 18 | Column 2 |
| 19 | Column 1 |
| 20 | Column 7 |
| 21 | $K_0$ |
| 22 | $K_1$ |
| 23 | $K_2$ |
| 24 | 40/80 |
| 25 | Alpha Lock |

**Table 16–26. The Keyboard Connector Pinout**

# THE C128 KEYBOARD TABLE

| | C0 PIN13 | C1 PIN19 | C2 PIN18 | C3 PIN17 | C4 PIN16 | C5 PIN15 | C6 PIN14 | C7 PIN20 | K0 PIN21 | K1 PIN22 | K2 PIN23 | GND PIN-1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R0 PIN12 | INS DEL | # 3 | % 5 | ' 7 | ) 9 | + | 1b | £ 1 | HELP | ESC | ALT | |
| R1 PIN11 | RET | W | R | Y | I | P | * | ← | 8 | + | 0 | |
| R2 PIN10 | ↑ ↓ | A | D | G | J | L | ] ; | CTRL | 5 | - | . | |
| R3 PIN5 | F8 F7 | $ 4 | & 6 | ( 8 | 0 | - | CLR HOM | " 2 | TAB | LINE FEED | ↑ | |
| R4 PIN8 | F2 F1 | Z | C | B | M | > . | RIGHT SHIFT | SPACE BAR | 2 | ENTER | ↓ | |
| R5 PIN7 | F4 F3 | S | F | H | K | [ : | = | | 4 | 6 | ← | |
| R6 PIN6 | F6 F5 | E | T | U | O | @ | π ^ | Q | 7 | 9 | → | |
| R7 PIN9 | ← → | LEFT SHIFT | X | V | N | < , | ? / | RUN STOP | 1 | 3 | NO SCRL | |

SHIFT LOCK — / (LOCKING)

NMI PIN3 ← RESTR — /

40/80 PIN24 ← 40/80 DSPLY — / (LOCKING)

P6510 PIN25 ← CAPS LOCK — / (LOCKING)

**NOTE:** Pins R0 through R7 pertain to the keyboard row values for the keyboard SCAN. These pins correspond to bits 0 through 7 of location 56321 ($DC01).

Pins C0 through C7 are the keyboard's column values, which correspond to bits 0 through 7 of location 56320 ($DC00).

Pins K0 through K2 pertain to the C128 keyboard control register bit, 0 through 2 of location 53295 ($D02F).

# APPENDIXES


Appendix A – BASIC Language Error Messages 644

Appendix B – DOS Error Messages 648

Appendix C – Connectors/Ports for Peripheral Equipment 652

Appendix D – Screen Display Codes 658

Appendix E – ASCII and CHR$ Codes 660

Appendix F – Screen and Color Memory Maps 663

Appendix G – Derived Trigonometric Functions 665

Appendix H – Control and Escape Codes 666

Appendix I – BASIC 7.0 Abbreviations 670

Appendix J – Disk Command Summary 674

Appendix K –

Part I–Commodore 128 CP/M 676

Part II–Calling CP/M BIOS, 8502 BIOS, and CP/M User Functions in Z80 Machine Language 702

Part III–The CP/M System Memory Map 709

Appendix L – Commodore 128 System Schematics 721

# APPENDIX A

## BASIC LANGUAGE ERROR MESSAGES

The following error messages are displayed by BASIC. Error messages can also be displayed with the use of the ERR$() function. The error numbers below refer only to the number assigned to the error for use with the ERR$() function.

| ERROR # | ERROR NAME | DESCRIPTION |
|---|---|---|
| 1 | TOO MANY FILES | There is a limit of ten files OPEN at one time. |
| 2 | FILE OPEN | An attempt was made to open a file using the number of an already open file. |
| 3 | FILE NOT OPEN | The file number specified in an I/O statement must be opened before use. |
| 4 | FILE NOT FOUND | Either no file with that name exists (disk) or an end-of-tape marker was read (tape). |
| 5 | DEVICE NOT PRESENT | The required I/O device is not available or buffers deallocated (cassette). Check to make sure the device is connected and turned on. |
| 6 | NOT INPUT FILE | An attempt was made to GET or INPUT data from a file that was specified as output only. |
| 7 | NOT OUTPUT FILE | An attempt was made to send data to a file that was specified as input only. |
| 8 | MISSING FILE NAME | File name missing in command. |
| 9 | ILLEGAL DEVICE NUMBER | An attempt was made to use a device improperly (SAVE to the screen, etc.). |
| 10 | NEXT WITHOUT FOR | Either loops are nested incorrectly, or there is a variable name in a NEXT statement that doesn't correspond with one in FOR. |
| 11 | SYNTAX | A statement not recognized by BASIC. This could be because of a missing or extra parenthesis, a misspelled key word, etc. |

| ERROR # | ERROR NAME | DESCRIPTION |
|---|---|---|
| 12 | RETURN WITHOUT GOSUB | A RETURN statement was encountered when no GOSUB statement was active. |
| 13 | OUT OF DATA | A READ statement is encountered without any data left to READ. |
| 14 | ILLEGAL QUANTITY | A number used as the argument of a function or statement is outside the allowable range. |
| 15 | OVERFLOW | The result of a computation is larger than the largest number allowed (1.701411834E + 38). |
| 16 | OUT OF MEMORY | Either there is no more room for program code and/or program variables, or there are too many nested DO, FOR or GOSUB statements in effect. |
| 17 | UNDEF'D STATEMENT | A referenced line number doesn't exist in the program. |
| 18 | BAD SUBSCRIPT | The program tried to reference an element of an array out of the range specified by the DIM statement. |
| 19 | REDIM'D ARRAY | An array can only be DIMensioned once. |
| 20 | DIVISION BY ZERO | Division by zero is not allowed. |
| 21 | ILLEGAL DIRECT | INPUT, GET, INPUT#, GET# and GETKEY statements are allowed only within a program. |
| 22 | TYPE MISMATCH | This error occurs when a numeric value is assigned to a string variable or vice versa. |
| 23 | STRING TOO LONG | A string can contain up to 255 characters. |
| 24 | FILE DATA | Bad data read from a tape or disk file. |
| 25 | FORMULA TOO COMPLEX | The computer was unable to evaluate this expression. Simplify the expression (break into two parts or use fewer parentheses). |
| 26 | CAN'T CONTINUE | The CONT command does not work if the program was not RUN, if there was an error, or if a line has been edited. |
| 27 | UNDEF'D FUNCTION | A user-defined function that was never defined was referenced. |

| ERROR # | ERROR NAME | DESCRIPTION |
|---|---|---|
| 28 | VERIFY | The program on tape or disk does not match the program in memory. |
| 29 | LOAD | There was a problem loading. Try again. |
| 30 | BREAK | The STOP command was issued in a program or the **STOP** key was pressed to halt program execution. |
| 31 | CAN'T RESUME | A RESUME statement was encountered without a TRAP statement in effect. |
| 32 | LOOP NOT FOUND | The program has encountered a DO statement and cannot find the corresponding LOOP. |
| 33 | LOOP WITHOUT DO | LOOP was encountered without a DO statement active. |
| 34 | DIRECT MODE ONLY | This command is allowed only in direct mode, not from a program. |
| 35 | NO GRAPHICS AREA | A command (DRAW, BOX, etc.) to create graphics was encountered before the GRAPHIC command was executed. |
| 36 | BAD DISK | An attempt failed to HEADER a diskette, because the quick header method (no ID) was attempted on an unformatted diskette or the diskette is bad. |
| 37 | BEND NOT FOUND | The program encountered an "IF . . . THEN BEGIN" or "IF . . . THEN . . . ELSE BEGIN" construct, and could not find a BEND keyword to match the BEGIN. |
| 38 | LINE NUMBER TOO LARGE | An error has occurred in renumbering a BASIC program. The given parameters result in a line number greater than 63999 being generated; therefore, the renumbering was not performed. |
| 39 | UNRESOLVED REFERENCE | An error has occurred in renumbering a BASIC program. A line number referred to be a command (e.g., GOTO 999) does not exist. Therefore, the renumbering was not performed. |
| 40 | UNIMPLEMENTED COMMAND | A command not supported by BASIC 7.0 was encountered. |

| ERROR # | ERROR NAME | DESCRIPTION |
|---|---|---|
| 41 | FILE READ | An error condition was encountered while loading or reading a program or file from the disk drive (e.g., opening the disk drive door while a program was loading). |

# APPENDIX B

## DOS ERROR MESSAGES

The following DOS error messages are returned through the DS and DS$ variables. The DS variable contains just the error number, and the DS$ variable contains the error number, the error message, and any corresponding track and sector number. NOTE: Error message numbers less than 20 should be ignored with the exception of 01, which gives information about the number of files scratched with the SCRATCH command.

| ERROR NUMBER | ERROR MESSAGE AND DESCRIPTION |
|---|---|
| 20 | READ ERROR (block header not found)<br>The disk controller is unable to locate the header of the requested data block. Causes: an illegal sector number, or the header has been destroyed. |
| 21 | READ ERROR (no sync character)<br>The disk controller is unable to detect a sync mark on the desired track. Causes: misalignment of the read/write head, no diskette, or an unformatted or improperly seated diskette. Can also indicate a hardware failure. |
| 22 | READ ERROR (data block not present)<br>The disk controller has been requested to read or verify a data block that was not properly written. This error occurs in conjunction with the BLOCK commands and can indicate an illegal track and/or sector request. |
| 23 | READ ERROR (checksum error in data block)<br>This error message indicates there is an error in one or more of the data bytes. The data has been read into the DOS memory, but the checksum over the data is in error. This message may also indicate hardware grounding problems. |
| 24 | READ ERROR (byte decoding error)<br>The data or header has been read into the DOS memory but a hardware error has been created owing to an invalid bit pattern in the data byte. This message may also indicate hardware grounding problems. |
| 25 | WRITE ERROR (write-verify error)<br>This message is generated if the controller detects a mismatch between the written data and the data in the DOS memory. |
| 26 | WRITE PROTECT ON<br>This message is generated when the controller has been requested to write a data block while the write protect switch is depressed. This is caused by using a diskette with a write protect tab over the notch or a notchless diskette. |

| ERROR NUMBER | ERROR MESSAGE AND DESCRIPTION |
|---|---|
| 27 | READ ERROR<br>This message is generated when a checksum error has been detected in the header of the requested data block. The block has not been read into DOS memory. |
| 28 | WRITE ERROR<br>This error message is generated when a data block is too long and overwrites the sync mark of the next header. |
| 29 | DISK ID MISMATCH<br>This message is generated when the controller has been requested to access a diskette that has not been initialized or improperly formatted. The message can also occur if a diskette has a bad header. |
| 30 | SYNTAX ERROR (general syntax)<br>The DOS cannot interpret the command sent to the command channel. Typically, this is caused by an illegal number of file names, or patterns that are illegally used. For example, two file names appear on the left side of the COPY command. |
| 31 | SYNTAX ERROR (invalid command)<br>The DOS does not recognize the command. The command must start in the first position. |
| 32 | SYNTAX ERROR (invalid command)<br>The command sent is longer than 58 characters. Use abbreviated disk commands. |
| 33 | SYNTAX ERROR (invalid file name)<br>Pattern matching is invalidly used in the OPEN or SAVE command. Spell out the file name. |
| 34 | SYNTAX ERROR (no file given)<br>The file name was left out of the command or the DOS does not recognize it as such. Typically, a colon(:) has been left out of the command. |
| 39 | SYNTAX ERROR (invalid command)<br>This error may result if the command sent to the command channel (secondary address 15) is unrecognized by the DOS. |
| 50 | RECORD NOT PRESENT<br>Result of disk reading past the last record through INPUT# or GET# commands. This message will also appear after positioning to a record beyond end-of-file in a relative file. If the intent is to expand the file by adding the new record (with a PRINT# command), the error message may be ignored. INPUT# and GET# should not be attempted after this error is detected without first repositioning. |

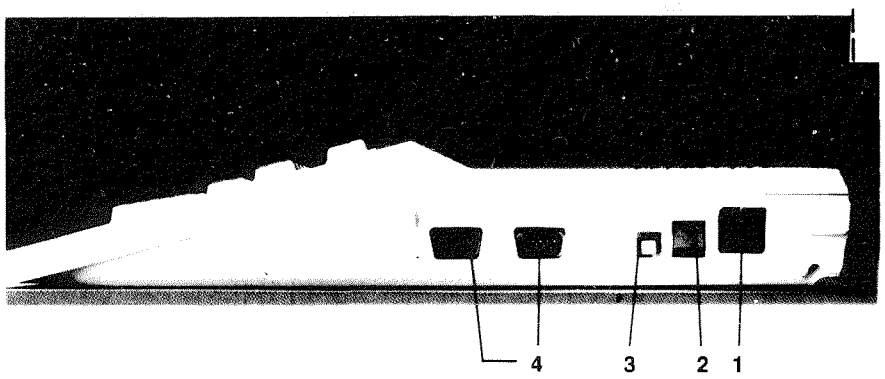
| ERROR NUMBER | ERROR MESSAGE AND DESCRIPTION |
|---|---|
| 51 | OVERFLOW IN RECORD<br>PRINT# statement exceeds record boundary. Information is truncated. Since the carriage return that is sent as a record terminator is counted in the record size, this message will occur if the total characters in the record (including the final carriage return) exceed the defined size of the record. |
| 52 | FILE TOO LARGE<br>The record position within a relative file indicates that disk overflow will result. |
| 60 | WRITE FILE OPEN<br>This message is generated when a write file that has not been closed is being opened for reading. |
| 61 | FILE NOT OPEN<br>A file that has not been opened in the DOS is being accessed. Sometimes in this situation, a message is not generated; the request is simply ignored. |
| 62 | FILE NOT FOUND<br>The requested file does not exist on the indicated drive. |
| 63 | FILE EXISTS<br>The file name of the file being created already exists on the diskette. |
| 64 | FILE TYPE MISMATCH<br>The requested file access is not possible using files of the type named. Reread the chapter covering that file type. |
| 65 | NO BLOCK<br>Occurs in conjunction with block allocation. The sector you tried to allocate is already allocated. The track and sector numbers returned are the next higher track and sector available. If the track number returned is 0, all remaining higher sectors are full. If the diskette is not full yet, try a lower track and sector. |
| 66 | ILLEGAL TRACK AND SECTOR<br>The DOS has attempted to access a track or a block that does not exist in the format being used. This may indicate a problem reading the pointer to the next block. |
| 67 | ILLEGAL SYSTEM T OR S<br>This special error message indicates an illegal system track or sector. |
| 70 | NO CHANNEL (available)<br>The requested channel is not available, or all channels are in use. A maximum of five buffers are available for use. A sequential file requires two buffers; a relative file requires three buffers; and the error/command channel requires one buffer. You may use any combination of those as long as the combination does not exceed five buffers. |

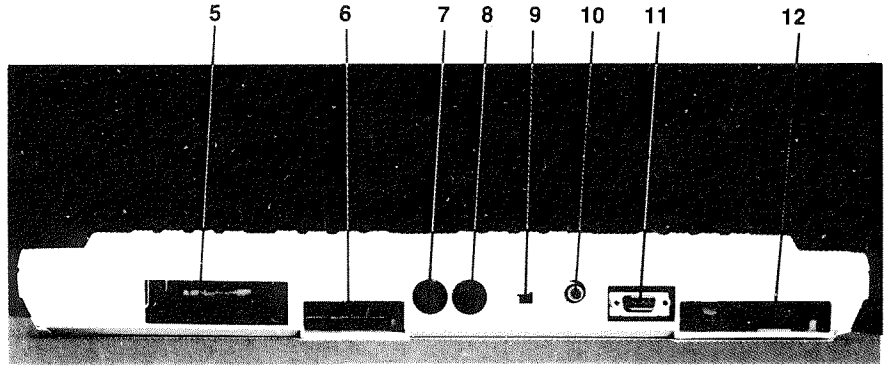
| ERROR NUMBER | ERROR MESSAGE AND DESCRIPTION |
|---|---|
| 71 | DIRECTORY ERROR<br>The BAM (Block Availability Map) on the diskette does not match the copy on disk memory. To correct this, initialize the disk drive. |
| 72 | DISK FULL<br>Either the blocks on the diskette are used, or the directory is at its entry limit. DISK FULL is sent when two blocks are still available on the diskette, in order to allow the current file to be closed. |
| 73 | DOS VERSION NUMBER (73, CBM DOS V30 1571, 00, 00)<br>DOS 1 and 2 are read compatible but not write compatible. Disks may be interchangeably read with either DOS, but a disk formatted on one version cannot be written upon with the other version because the format is different. This error is displayed whenever an attempt is made to write upon a disk that has been formatted in a noncompatible format. This message will also appear after power-up or reset and is not an error in this case. |
| 74 | DRIVE NOT READY<br>An attempt has been made to access the disk drive without a diskette inserted; or the drive lever or door is open. |

# APPENDIX C

## CONNECTORS/PORTS FOR PERIPHERAL EQUIPMENT

1. Power Socket
2. Power Switch
3. Reset Button
4. Controller Ports
5. Expansion Port
6. Cassette Port
7. Serial Port
8. Composite Video Connector
9. Channel Selector
10. RF Connector
11. RGBI Connector
12. User Port



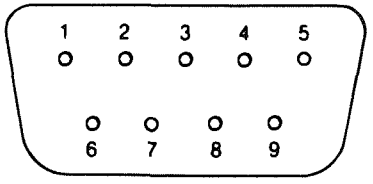



## SIDE PANEL CONNECTIONS

1. Power Socket—The free end of the cable from the power supply is attached here.
2. Power Switch—Turns on power from the transformer.
3. Reset Button—Resets computer (warm start).
4. Controller Ports—There are two Controller ports, numbered 1 and 2. Each Controller port can accept a joystick mouse or a game controller paddle. A light pen can be plugged only into port 1, the port closest to the front of the computer. Use the ports as instructed with the software.

### CONTROLLER PORT 1

| PIN | TYPE | NOTE |
|---|---|---|
| 1 | JOYA0 | |
| 2 | JOYA1 | |
| 3 | JOYA2 | |
| 4 | JOYA3 | |
| 5 | POT AY | |
| 6 | BUTTON A/LP | |
| 7 | +5V | MAX. 50mA |
| 8 | GND | |
| 9 | POT AX | |




### CONTROLLER PORT 2

| PIN | TYPE | NOTE |
|---|---|---|
| 1 | JOYB0 | |
| 2 | JOYB1 | |
| 3 | JOYB2 | |
| 4 | JOYB3 | |
| 5 | POT BY | |
| 6 | BUTTON B | |
| 7 | +5V | MAX. 50mA |
| 8 | GND | |
| 9 | POT BX | |

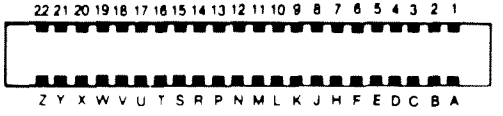
## REAR CONNECTIONS

5. Expansion Port—This rectangular slot is a parallel port that accepts program or game cartridges as well as special interfaces.

## CARTRIDGE EXPANSION PORT

| PIN | TYPE | PIN | TYPE |
|---|---|---|---|
| 12 | BA | 1 | GND |
| 13 | DMA | 2 | +5V |
| 14 | D7 | 3 | +5V |
| 15 | D6 | 4 | IRQ |
| 16 | D5 | 5 | R/W |
| 17 | D4 | 6 | Dot Clock |
| 18 | D3 | 7 | I/O 1 |
| 19 | D2 | 8 | GAME |
| 20 | D1 | 9 | EXROM |
| 21 | D0 | 10 | I/O 2 |
| 22 | GND | 11 | ROML |

| PIN | TYPE | PIN | TYPE |
|---|---|---|---|
| N | A9 | A | GND |
| P | A8 | B | ROMH |
| R | A7 | C | RESET |
| S | A6 | D | NMI |
| T | A5 | E | S 02 |
| U | A4 | F | A15 |
| V | A3 | H | A14 |
| W | A2 | J | A13 |
| X | A1 | K | A12 |
| Y | A0 | L | A11 |
| Z | GND | M | A10 |

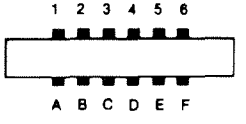



(view of port from the back of the C128)

6. Cassette Port—A 1530 Datassette recorder can be attached here to store programs and information.

## CASSETTE PORT

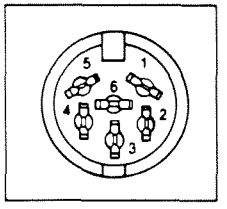
| PIN | TYPE |
|---|---|
| A-1 | GND |
| B-2 | +5V |
| C-3 | CASSETTE MOTOR |
| D-4 | CASSETTE READ |
| E-5 | CASSETTE WRITE |
| F-6 | CASSETTE SENSE |



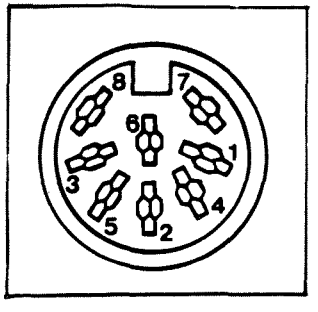
7. Serial Port—A Commodore serial printer or disk drive can be attached directly to the Commodore 128 through this port.

**SERIAL I/O PORT**

| PIN | TYPE |
|---|---|
| 1 | SERIAL $\overline{\text{SRQIN}}$ |
| 2 | GND |
| 3 | SERIAL ATN IN/OUT |
| 4 | SERIAL CLK IN/OUT |
| 5 | SERIAL DATA IN/OUT |
| 6 | $\overline{\text{RESET}}$ |




8. Composite Video Connector—This DIN connector supplies direct audio and composite video signals. These can be connected to the Commodore monitor or used with separate components. This is the 40-column output connector.

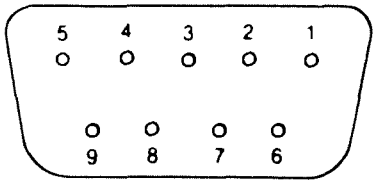



(view of port while facing the rear of the C128)

**COMPOSITE VIDEO CONNECTOR**

| PIN | TYPE | NOTE |
|---|---|---|
| 1 | LUM/SYNC | Luminance/SYNC output |
| 2 | GND | |
| 3 | AUDIO OUT | |
| 4 | VIDEO OUT | Composite signal output |
| 5 | AUDIO IN | |
| 6 | COLOR OUT | Chroma signal output |
| 7 | NC | No connection |
| 8 | NC | No connection |

9. Channel Selector—Use this switch to select which TV channel (L = channel 3, H = channel 4) the computer's picture will be displayed on when using a television instead of a monitor.
10. RF Connector—This connector supplies both picture and sound to your television set. (A television can display only a 40-column picture.)
11. RGBI Connector—This 9-pin connector supplies direct audio and an RGBI (Red/ Green/ Blue/Intensity) signal. This is the 80-column output.




**RGBI CONNECTOR**

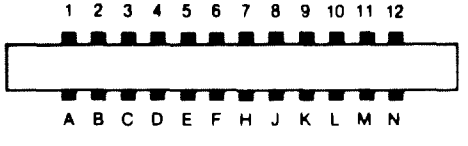
| PIN | SIGNAL |
|---|---|
| **1** | **Ground** |
| **2** | **Ground** |
| **3** | **Red** |
| **4** | **Green** |
| **5** | **Blue** |
| **6** | **Intensity** |
| **7** | **Monochrome** |
| **8** | **Horizontal Sync** |
| **9** | **Vertical Sync** |

12. User Port—Various interface devices can be attached here, including a Commodore modem.

## USER I/O PORT

| PIN | TYPE | NOTE |
|---|---|---|
| 1 | GND | |
| 2 | +5V | MAX. 100mA |
| 3 | RESET | |
| 4 | CNT1 | |
| 5 | SP1 | |
| 6 | CNT2 | |
| 7 | SP2 | |
| 8 | PC2 | |
| 9 | SER. ATN IN | |
| 10 | 9 VAC | MAX. 100mA |
| 11 | 9 VAC | MAX. 100mA |
| 12 | GND | |

| PIN | TYPE | NOTE |
|---|---|---|
| A | GND | |
| B | FLAG2 | |
| C | PB0 | |
| D | PB1 | |
| E | PB2 | |
| F | PB3 | |
| H | PB4 | |
| J | PB5 | |
| K | PB6 | |
| L | PB7 | |
| M | PA2 | |
| N | GND | |



# APPENDIX D

## SCREEN DISPLAY CODES

### SCREEN DISPLAY CODES 40 COLUMNS

The chart below lists all the characters built into the Commodore screen character sets. It shows which numbers should be POKEd into the VIC chip (40-column) screen memory (location 1024 to 2023) to get a desired character on the 40-column screen. (Remember, to set color memory, use locations 55296 to 56295.) Also shown is which character corresponds to a number PEEKed from the screen.

Two character sets are available. Both are available simultaneously in 80-column mode, but only one is available at a time in 40-column mode. The sets are switched by holding down the **SHIFT** and C= (Commodore) keys simultaneously. The entire screen of characters changes to the selected character set.

From BASIC, PRINT CHR$(142) will switch to upper case/graphics mode and PRINT CHR$(14) will switch to upper/lower case mode.

Any number on the chart may also be displayed in reverse. The reverse character code can be obtained by adding 128 to the values shown.

| SET 1 | SET 2 | POKE | SET 1 | SET 2 | POKE | SET 1 | SET 2 | POKE |
|---|---|---|---|---|---|---|---|---|
| @ | | 0 | N | n | 14 | £ | | 28 |
| A | a | 1 | O | o | 15 | ] | | 29 |
| B | b | 2 | P | p | 16 | ↑ | | 30 |
| C | c | 3 | Q | q | 17 | ← | | 31 |
| D | d | 4 | R | r | 18 | SPACE | | 32 |
| E | e | 5 | S | s | 19 | ! | | 33 |
| F | f | 6 | T | t | 20 | " | | 34 |
| G | g | 7 | U | u | 21 | # | | 35 |
| H | h | 8 | V | v | 22 | $ | | 36 |
| I | i | 9 | W | w | 23 | % | | 37 |
| J | j | 10 | X | x | 24 | & | | 38 |
| K | k | 11 | Y | y | 25 | ' | | 39 |
| L | l | 12 | Z | z | 26 | ( | | 40 |
| M | m | 13 | [ | | 27 | ) | | 41 |

| SET 1 | SET 2 | POKE | SET 1 | SET 2 | POKE | SET 1 | SET 2 | POKE |
|---|---|---|---|---|---|---|---|---|
| * | | 42 | | G | 71 | | | 100 |
| + | | 43 | | H | 72 | | | 101 |
| , | | 44 | | I | 73 | | | 102 |
| – | | 45 | | J | 74 | | | 103 |
| . | | 46 | | K | 75 | | | 104 |
| / | | 47 | | L | 76 | | | 105 |
| 0 | | 48 | | M | 77 | | | 106 |
| 1 | | 49 | | N | 78 | | | 107 |
| 2 | | 50 | | O | 79 | | | 108 |
| 3 | | 51 | | P | 80 | | | 109 |
| 4 | | 52 | | Q | 81 | | | 110 |
| 5 | | 53 | | R | 82 | | | 111 |
| 6 | | 54 | | S | 83 | | | 112 |
| 7 | | 55 | | T | 84 | | | 113 |
| 8 | | 56 | | U | 85 | | | 114 |
| 9 | | 57 | | V | 86 | | | 115 |
| : | | 58 | | W | 87 | | | 116 |
| ; | | 59 | | X | 88 | | | 117 |
| < | | 60 | | Y | 89 | | | 118 |
| = | | 61 | | Z | 90 | | | 119 |
| > | | 62 | | | 91 | | | 120 |
| ? | | 63 | | | 92 | | | 121 |
| | | 64 | | | 93 | | | 122 |
| | A | 65 | | | 94 | | | 123 |
| | B | 66 | | | 95 | | | 124 |
| | C | 67 | SPACE | | 96 | | | 125 |
| | D | 68 | | | 97 | | | 126 |
| | E | 69 | | | 98 | | | 127 |
| | F | 70 | | | 99 | | | |

**Codes from 128-255 are reversed images of codes 0-127.**

# APPENDIX E

## ASCII AND CHR$ CODES

This appendix shows you what characters will appear if you PRINT CHR$(X), for all possible values of X. It also shows the values obtained by typing PRINT ASC (''x''), where x is any character that can be displayed. This is useful in evaluating the character received in a GET statement, converting upper to lower case and printing character-based commands (such as switch to upper/lower case) that could not be enclosed in quotes.

| PRINTS | CHR$ | PRINTS | CHR$ | PRINTS | CHR$ | PRINTS | CHR$ |
|---|---|---|---|---|---|---|---|
| | 0 | | 23 | . | 46 | E | 69 |
| | 1 | Tab set/cl | 24 | / | 47 | F | 70 |
| Underline(80) | 2 | | 25 | 0 | 48 | G | 71 |
| | 3 | | 26 | 1 | 49 | H | 72 |
| | 4 | ESC CHAR | 27 | 2 | 50 | I | 73 |
| WHT | 5 | RED | 28 | 3 | 51 | J | 74 |
| | 6 | CRSR → | 29 | 4 | 52 | K | 75 |
| Bell tone | 7 | GRN | 30 | 5 | 53 | L | 76 |
| DISABLES SHIFT C= | 8 | RED | 31 | 6 | 54 | M | 77 |
| ENABLES SHIFT C= | 9 | SPACE | 32 | 7 | 55 | N | 78 |
| Line feed | 10 | ! | 33 | 8 | 56 | O | 79 |
| Used | 11 | " | 34 | 9 | 57 | P | 80 |
| Used | 12 | # | 35 | : | 58 | Q | 81 |
| RETURN | 13 | $ | 36 | ; | 59 | R | 82 |
| SWITCH TO LOWER CASE | 14 | % | 37 | < | 60 | S | 83 |
| Flash on | 15 | & | 38 | = | 61 | T | 84 |
| | 16 | ' | 39 | > | 62 | U | 85 |
| CRSR ↑ | 17 | ( | 40 | ? | 63 | V | 86 |
| RVS OFF | 18 | ) | 41 | @ | 64 | W | 87 |
| CLR HOME | 19 | * | 42 | A | 65 | X | 88 |
| INST DEL | 20 | + | 43 | B | 66 | Y | 89 |
| | 21 | , | 44 | C | 67 | Z | 90 |
| | 22 | - | 45 | D | 68 | [ | 91 |

| PRINTS | CHR$ | PRINTS | CHR$ | PRINTS | CHR$ | PRINTS | CHR$ |
|---|---|---|---|---|---|---|---|
| £ | 92 | ♥ | 115 | f4 | 138 | ▌ | 161 |
| ] | 93 | | 116 | f6 | 139 | ▄ | 162 |
| ↑ | 94 | | 117 | f8 | 140 | | 163 |
| ← | 95 | ╳ | 118 | SHIFT RETURN | 141 | | 164 |
| | 96 | ○ | 119 | SWITCH TO UPPER CASE | 142 | | 165 |
| ♠ | 97 | ♣ | 120 | | 143 | ▒ | 166 |
| | 98 | | 121 | BLK | 144 | | 167 |
| | 99 | ♦ | 122 | ↓ CRSR | 145 | ▒ | 168 |
| | 100 | ┼ | 123 | RVS OFF | 146 | ◢ | 169 |
| | 101 | ▒ | 124 | CLR HOME | 147 | ▐ | 170 |
| | 102 | ┬ | 125 | INST DEL | 148 | ┼ | 171 |
| | 103 | ┬ | 126 | Brown | 149 | ▀ | 172 |
| | 104 | ◥ | 127 | Lt. Red | 150 | ┌ | 173 |
| | 105 | | 128 | Dk. Gray | 151 | ┐ | 174 |
| | 106 | Orange | 129 | Gray | 152 | ▄ | 175 |
| | 107 | | 130 | Lt. Green | 153 | └ | 176 |
| | 108 | | 131 | Lt. Blue | 154 | ┴ | 177 |
| | 109 | | 132 | Lt. Gray | 155 | ┬ | 178 |
| | 110 | f1 | 133 | RUN | 156 | ┼ | 179 |
| | 111 | f3 | 134 | ← CRSR | 157 | ▐ | 180 |
| | 112 | f5 | 135 | YEL | 158 | ▌ | 181 |
| ● | 113 | f7 | 136 | CYN | 159 | ▄ | 182 |
| | 114 | f2 | 137 | SPACE | 160 | ▀ | 183 |

| PRINTS | CHR$ | PRINTS | CHR$ | PRINTS | CHR$ | PRINTS | CHR$ |
|---|---|---|---|---|---|---|---|
| ▀ | 184 | ▗ | 186 | ▙ | 188 | ▙ | 190 |
| ▄ | 185 | ▙ | 187 | ▙ | 189 | ▚ | 191 |

**CODES 192-223 SAME AS 96-127**
**CODES 224-254 SAME AS 160-190**
**CODE 255 SAME AS 126**

**NOTE:** The 80-column (RGBI) output has three colors that are different from the 40-column (composite video) color output. This means that the character string codes that represent color codes for these three colors are used differently depending on which video output is used. The following character string codes represent these colors in each video output.

| CHR$ | 40-COLUMN (VIC COMPOSITE) | 80-COLUMN (8563 RGBI) |
|---|---|---|
| 129 | Orange | Dark Purple |
| 149 | Brown | Dark Yellow |
| 151 | Dark Gray | Dark Cyan |

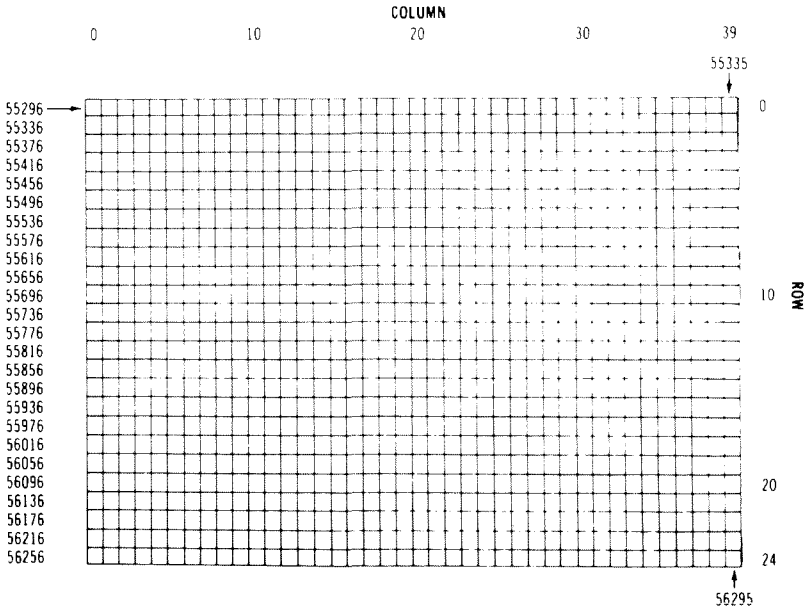
# APPENDIX F

## SCREEN AND COLOR MEMORY MAPS— C128 MODE, 40-COLUMN AND C64 MODE

The maps below display the memory locations used in 40-column mode (C128 and C64) for identifying the characters on the screen as well as their color. Each map is separately controlled and consists of 1000 positions.

The character displayed on the maps can be controlled directly with the POKE command.

### VIC CHIP (40-COLUMN) SCREEN MEMORY MAP

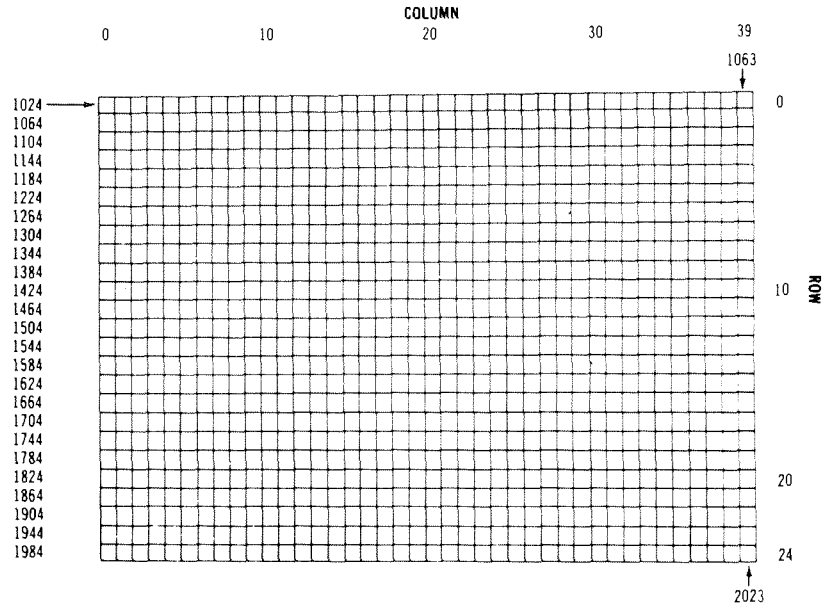



The Screen Map is POKEd with a Screen Display Code value (see Appendix D). For example:

**POKE 1024, 13** (from BANK 0 or 15)

will display the letter M in the upper-left corner of the screen.

## VIC CHIP (40-COLUMN) COLOR MEMORY MAP




If the color map is POKEd with a color value, this changes the character color. For example:

**POKE 55296,1** (from BANK 15)

will change the letter M inserted above from light green to white.

---

COLOR CODES—40 COLUMNS

| | |
|---|---|
| **0 Black** | **8 Orange** |
| **1 White** | **9 Brown** |
| **2 Red** | **10 Light Red** |
| **3 Cyan** | **11 Dark Gray** |
| **4 Purple** | **12 Medium Gray** |
| **5 Green** | **13 Light Green** |
| **6 Blue** | **14 Light Blue** |
| **7 Yellow** | **15 Light Gray** |

**Border Control Memory 53280**
**Background Control Memory 53281**

---

# APPENDIX G

## DERIVED TRIGONOMETRIC FUNCTIONS

| FUNCTION | 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)) + π/2 |
| INVERSE SECANT | ARCSEC(X) = ATN(X/SQR(X*X–1)) |
| INVERSE COSECANT | ARCCSC(X) = ATN(X/SQR(X*X–1)) + (SGN(X)–1)*π/2 |
| INVERSE COTANGENT | ARCCOT(X) = –ATN(X) + π/2 |
| HYPERBOLIC SINE | SINH(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 H

## CONTROL AND ESCAPE CODES

### CONTROL CODES

The table below lists the control codes used by the Commodore 128. The print codes in the first column are used in PRINT statements. The key codes in the second column are the sequence of keys pressed to perform specific controls. Hold down the **CONTROL** key (or the key specified on the left in the key code column) and strike the key specified to the right in the key code column.

| PRINT CODES (CHR$) | KEY CODES KEY SEQUENCE | FUNCTION | EFFECTIVE IN MODE: C64 | C128 |
|---|---|---|---|---|
| CHR$(2) | CTRL B | Underline (80) | | √ |
| CHR$(5) | CTRL 2 or CTRL E | Set character color to white | √ | √ |
| CHR$(7) | CTRL G | Produce bell tone | | √ |
| CHR$(8) | CTRL H | Disable character set change | √ | |
| CHR$(9) | CTRL I | Enable character set change | √ | |
| | | Move cursor to next set tab position | | √ |
| CHR$(10) | CTRL J | Line feed | | √ |
| CHR$(11) | CTRL K | Enable character set change | | √ |
| CHR$(12) | CTRL L | Disable character mode change | | √ |
| CHR$(13) | CTRL M | Send a carriage return and line feed to the computer and enter a line of BASIC | √ | √ |
| CHR$(14) | CTRL N | Set character set to upper/lower case | √ | √ |
| CHR$(15) | CTRL O | Turn flash on (80) | | √ |
| CHR$(17) | CRSR DOWN/CTRL Q | Move the cursor down one row | √ | √ |
| CHR$(18) | CTRL 9 or CTRL R | Cause characters to be printed in reverse field | √ | √ |
| CHR$(19) | HOME | Move the cursor to the home position (top left) of the display (the current window) | √ | √ |

**NOTE:** (40) = 40-column screen only.
(80) = 80-column screen only.

## PRINT CODES KEY CODES

| (CHR$) | KEY SEQUENCE | FUNCTION | EFFECTIVE IN MODE: C64 | C128 |
|---|---|---|---|---|
| CHR$(20) | DEL or CTRL T | Delete last character typed and move all characters to the right of the deleted character one space to the left | √ | √ |
| CHR$(24) | CTRL X, CTRL TAB or C= TAB | Tab set/clear | | |
| CHR$(27) | ESC or CTRL[ | Send an ESC character | | √ |
| CHR$(28) | CTRL 3 or CTRL £ | Set character color to red (40) and (80) | √ | √ |
| CHR$(29) | CRSR or CTRL] | Move cursor one column to the right | √ | √ |
| CHR$(30) | CTRL 6 or CTRL | Set character color to green (40) and (80) | √ | √ |
| CHR$(31) | CTRL 7 or CTRL = | Set character color to blue (40) and (80) | | |
| CHR$(34) | | Print a double quote on screen and place editor in quote mode | √ | √ |
| CHR$(129) | C= 1 | Set character color to orange (40); dark purple (80) | √ | √ |
| CHR$(130) | | Underline off (80) | | √ |
| CHR$(131) | | Run a program. This CHR$ code does not work in PRINT CHR$ (131), but works from keyboard buffer | √ | √ |
| CHR$(133) | F1 | Reserved CHR$ code for F1 key | √ | √ |
| CHR$(134) | F3 | Reserved CHR$ code for F3 key | √ | √ |
| CHR$(135) | F5 | Reserved CHR$ code for F5 key | √ | √ |
| CHR$(136) | F7 | Reserved CHR$ code for F7 key | √ | √ |
| CHR$(137) | F2 | Reserved CHR$ code for F2 key | √ | √ |
| CHR$(138) | F4 | Reserved CHR$ code for F4 key | √ | √ |
| CHR$(139) | F6 | Reserved CHR$ code for F6 key | √ | √ |
| CHR$(140) | F8 | Reserved CHR$ code for F8 key | √ | √ |
| CHR$(141) | SHIFT RETURN, CTRL ENTER, C= ENTER or C= RETURN | Send a carriage return and line feed without entering a BASIC line | √ | √ |
| CHR$(142) | | Set the character set to upper case/graphic | √ | √ |
| CHR$(143) | | Turn flash off (80) | | √ |

**NOTE:** (40) = 40-column screen only.
(80) = 80-column screen only.

## PRINT CODES KEY CODES

| (CHR$) | KEY SEQUENCE | FUNCTION | EFFECTIVE IN MODE: C64 | C128 |
|---|---|---|---|---|
| CHR$(144) | CTRL 1 | Set character color to black (40) and (80) | √ | √ |
| CHR$(145) | CRSR UP | Move cursor or printing position up one row | √ | √ |
| CHR$(146) | CTRL 0 | Terminate reverse field display | √ | √ |
| CHR$(147) | CLEAR HOME | Clear the window screen and move the cursor to the top-left position | √ | √ |
| CHR$(148) | INST | Move character from cursor position end of line right one column | √ | √ |
| CHR$(149) | C= 2 | Set character color to brown (40); dark yellow (80) | √ | √ |
| CHR$(150) | C= 3 | Set character color to light red (40) and (80) | √ | √ |
| CHR$(151) | C= 4 | Set character color to dark gray (40); dark cyan (80) | √ | √ |
| CHR$(152) | C= 5 | Set character color to medium gray (40) and (80) | √ | √ |
| CHR$(153) | C= 6 | Set character color to light green (40) and (80) | √ | √ |
| CHR$(154) | C= 7 | Set character color to light blue (40) and (80) | √ | √ |
| CHR$(155) | C= 8 | Set character color to light gray (40) and (80) | √ | √ |
| CHR$(156) | CTRL 5 | Set character color to purple (40) and (80) | √ | √ |
| CHR$(157) | CRSR LEFT | Move cursor left by one column | √ | √ |
| CHR$(158) | CTRL 8 | Set character color to yellow (40) and (80) | | |
| CHR$(159) | or CTRL 4 | Set character color to cyan (40); light cyan (80) | √ | √ |

**NOTE:** (40) = 40-column screen only.
(80) = 80-column screen only.

## ESCAPE CODES

This table lists the key sequences for the ESCape functions available on the Commodore 128. ESCape sequences are entered by pressing and releasing the **ESC** key, followed by pressing the key listed in the right column.

| ESCAPE FUNCTION | SEQUENCE KEY |
|---|---|
| **Cancel quote, reverse, flash** | **ESC O** |
| **Erase to end of current line** | **ESC Q** |
| **Erase to start of current line** | **ESC P** |
| **Clear to end of screen** | **ESC @** |
| **Move to start of current line** | **ESC J** |
| **Move to end of current line** | **ESC K** |
| **Enable auto-insert mode** | **ESC A** |
| **Disable auto-insert mode** | **ESC C** |
| **Delete current line** | **ESC D** |
| **Insert line** | **ESC I** |
| **Set default tab stop (8 spaces)** | **ESC Y** |
| **Clear all tab stops** | **ESC Z** |
| **Enable scrolling** | **ESC L** |
| **Disable scrolling** | **ESC M** |
| **Scroll up** | **ESC V** |
| **Scroll down** | **ESC W** |
| **Enable bell (by control-G)** | **ESC G** |
| **Disable bell** | **ESC H** |
| **Set cursor to nonflashing mode** | **ESC E** |
| **Set cursor to flashing mode** | **ESC F** |
| **Set bottom right corner of screen window at cursor position** | **ESC B** |
| **Set top left corner of screen window at cursor position** | **ESC T** |
| **Swap 40/80 column display output device** | **ESC X** |

The following ESCape sequences have an effect on an 80-column screen only.

| ESCAPE FUNCTION | SEQUENCE KEY |
|---|---|
| **Change to underlined cursor (80)** | **ESC U** |
| **Change to block cursor (80)** | **ESC S** |
| **Set screen to reverse video (80)** | **ESC R** |
| **Return screen to normal (non reverse video) state (80)** | **ESC N** |

# APPENDIX I

## BASIC 7.0 ABBREVIATIONS

**NOTE:** The abbreviations below operate in upper case/graphics mode. Press the letter key(s) indicated, then hold down the **SHIFT** key and press the letter key following the word SHIFT.

| KEYWORD | ABBREVIATION |
|---|---|
| ABS | A SHIFT B |
| APPEND | A SHIFT P |
| ASC | A SHIFT S |
| ATN | A SHIFT T |
| AUTO | A SHIFT U |
| BACKUP | BA SHIFT C |
| BANK | B SHIFT A |
| BEGIN | B SHIFT E |
| BEND | BE SHIFT N |
| BLOAD | B SHIFT L |
| BOOT | B SHIFT O |
| BOX | none |
| BSAVE | B SHIFT S |
| BUMP | B SHIFT U |
| CATALOG | C SHIFT A |
| CHAR | CH SHIFT A |
| CHR$ | C SHIFT H |
| CIRCLE | C SHIFT I |
| CLOSE | CL SHIFT O |
| CLR | C SHIFT L |
| CMD | C SHIFT M |
| COLLECT | COLL SHIFT E |
| COLLISION | COL SHIFT L |
| COLOR | COL SHIFT O |
| CONCAT | C SHIFT O |
| CONT | none |
| COPY | CO SHIFT P |
| COS | none |
| DATA | D SHIFT A |
| DEC | none |
| DCLEAR | DCL SHIFT E |
| DCLOSE | D SHIFT C |
| DEF FN | none |
| DELETE | DE SHIFT L |
| DIM | D SHIFT I |

| KEYWORD | ABBREVIATION |
|---|---|
| DIRECTORY | DI SHIFT R |
| DLOAD | D SHIFT L |
| DO | none |
| DOPEN | D SHIFT O |
| DRAW | D SHIFT R |
| DS | none |
| DS$ | none |
| DSAVE | D SHIFT S |
| DVERIFY | D SHIFT V |
| EL | none |
| END | none |
| ENVELOPE | E SHIFT N |
| ER | none |
| ERR$ | E SHIFT R |
| EXIT | EX SHIFT I |
| EXP | E SHIFT X |
| FAST | none |
| FETCH | F SHIFT E |
| FILTER | F SHIFT I |
| FOR | F SHIFT O |
| FRE | F SHIFT R |
| FNXX | none |
| GET | G SHIFT E |
| GETKEY | GETK SHIFT E |
| GET# | none |
| GOSUB | GO SHIFT S |
| GO64 | none |
| GOTO | G SHIFT O |
| GRAPHIC | G SHIFT R |
| GSHAPE | G SHIFT S |
| HEADER | HE SHIFT A |
| HELP | X SHIFT X |
| HEX$ | H SHIFT E |
| IF . . . GOTO | none |
| IF . . . THEN . . . ELSE | none |
| INPUT | none |
| INPUT# | I SHIFT N |
| INSTR | IN SHIFT S |
| INT | none |
| JOY | J SHIFT O |
| KEY | K SHIFT E |
| LEFT$ | LE SHIFT F |
| LEN | none |
| LET | L SHIFT E |
| LIST | L SHIFT I |
| LOAD | L SHIFT O |
| LOCATE | LO SHIFT C |
| LOG | none |

| KEYWORD | ABBREVIATION |
|---|---|
| LOOP | LO SHIFT O |
| MID$ | M SHIFT I |
| MONITOR | MO SHIFT N |
| MOVSPR | M SHIFT O |
| NEW | none |
| NEXT | N SHIFT E |
| ON GOSUB | ON GO SHIFT S |
| ON GOTO | ON G SHIFT O |
| OPEN | O SHIFT P |
| PAINT | P SHIFT A |
| PEEK | PE SHIFT E |
| PEN | P SHIFT E |
| PI | none |
| PLAY | P SHIFT L |
| POINTER | PO SHIFT I |
| POKE | PO SHIFT K |
| POS | none |
| POT | P SHIFT O |
| PRINT | none |
| PRINT# | P SHIFT R |
| PRINT USING | US SHIFT I |
| PUDEF | P SHIFT U |
| RCLR | R SHIFT C |
| RDOT | R SHIFT D |
| READ | RE SHIFT A |
| RECORD | R SHIFT E |
| REM | none |
| RENAME | RE SHIFT N |
| RENUMBER | REN SHIFT U |
| RESTORE | RE SHIFT S |
| RESUME | RES SHIFT U |
| RETURN | RE SHIFT T |
| RGR | R SHIFT G |
| RIGHT$ | R SHIFT I |
| RND | R SHIFT N |
| RREG | R SHIFT R |
| RSPCOLOR | RSP SHIFT C |
| RSPPOS | R SHIFT S |
| RSPR | none |
| RSPRITE | RSP SHIFT R |
| RUN | R SHIFT U |
| RWINDOW | R SHIFT W |
| SAVE | S SHIFT A |
| SCALE | SC SHIFT A |
| SCNCLR | S SHIFT C |
| SCRATCH | SC SHIFT R |
| SGN | S SHIFT G |

| KEYWORD | ABBREVIATION |
|---|---|
| SIN | S SHIFT I |
| SLEEP | S SHIFT L |
| SLOW | none |
| SOUND | S SHIFT O |
| SPC | none |
| SPRCOLOR | SPR SHIFT C |
| SPRDEF | SPR SHIFT D |
| SPRITE | S SHIFT P |
| SPRSAV | SPR SHIFT S |
| SQR | S SHIFT Q |
| SSHAPE | S SHIFT S |
| STASH | S SHIFT T |
| ST | none |
| STEP | ST SHIFT E |
| STOP | ST SHIFT O |
| STR$ | ST SHIFT R |
| SWAP | S SHIFT W |
| SYS | none |
| TAB( | T SHIFT A |
| TAN | none |
| TEMPO | T SHIFT E |
| TI | none |
| TI$ | none |
| TO | none |
| TRAP | T SHIFT R |
| TROFF | TRO SHIFT F |
| TRON | TR SHIFT O |
| UNTIL | U SHIFT N |
| USR | U SHIFT S |
| VAL | none |
| VERIFY | V SHIFT E |
| VOL | V SHIFT O |
| WAIT | W SHIFT A |
| WHILE | W SHIFT H |
| WIDTH | WI SHIFT D |
| WINDOW | W SHIFT I |
| XOR | X SHIFT O |

# APPENDIX J

## DISK COMMAND SUMMARY

This appendix lists the commands used for disk operation in C128 and C64 modes on the Commodore 128. For detailed information on any of these commands, see Chapter 2. Your disk drive manual also has information on disk commands.

The *new* **BASIC 7.0** commands can be used *only* in C128 mode. *All* BASIC 2.0 commands can be used in both C128 and C64 modes.

| COMMAND | USE | BASIC 2.0 | BASIC 7.0 |
|---|---|---|---|
| APPEND | Append data to file | | √ |
| BLOAD | Load a binary file starting at the specified memory location | | √ |
| BOOT | Load and execute a bootable program | | √ |
| BSAVE | Save a binary file from the specified memory location | | √ |
| CATALOG | Display directory contents of disk on screen* | | √ |
| CLOSE | Close logical disk file | √ | |
| CMD | Redirect screen output to a peripheral device | √ | |
| COLLECT | Free inaccessible disk space* | | √ |
| CONCAT | Concatenate two data files* | | √ |
| COPY | Copy files between devices* | | √ |
| DCLEAR | Clear all open channels on disk drives | | √ |
| DCLOSE | Close logical disk file | | √ |
| DIRECTORY | Display directory of contents of disk on screen* | | √ |
| DLOAD | Load a BASIC program from disk | | √ |
| DOPEN | Open a disk file for a read and/or write operation | | √ |
| DSAVE | Save a BASIC program to disk | | √ |
| DVERIFY | Verify program in memory against program on disk | | √ |
| GET# | Receive input from open disk file | √ | |
| HEADER | Format a disk* | | √ |
| LOAD | Load a file from disk | √ | |
| OPEN | Open a file for input or output | √ | |
| PRINT# | Output a data to file | √ | |
| RECORD | Position relative file pointers* | | √ |
| RENAME | Change name of a file on disk* | | √ |

| COMMAND | USE | BASIC 2.0 | BASIC 7.0 |
|---|---|---|---|
| **RUN filename** | **Execute BASIC program from disk** | | √ |
| **SAVE** | **Store program in memory to disk** | √ | |
| **VERIFY** | **Verify program in memory against program on disk** | √ | |

*Although there is no single equivalent command in BASIC 2.0, there is an equivalent multi-command instruction. See your disk drive manual for these BASIC 2.0 conventions.

# APPENDIX K

## PART I—COMMODORE 128 CP/M

This appendix explains each CP/M BIOS, 8502 BIOS and User Function routine and how to call each in Z80 assembly language. This section assumes you already have some knowledge about Z80 machine language and the basic operations of the CP/M system. If you need more information about Z80 or CP/M, your local bookstore probably has several good reference books about these widely covered subjects. To fully cover these topics is beyond the scope of this reference guide. See ''Suggestions for Reading'' at the end of this guide.

Part I of this appendix first lists each CP/M BIOS, 8502 BIOS and User Function routine by number. Part II explains how and provides examples to call these routines. Part III lists the Z80 memory map.

The format used to describe these routines are as follows:

a) Function Name
b) Input Parameters
c) Output Parameters
d) Brief Description
e) Other required preparatory/post routines (or additional information)

The 8502 BIOS and User Function routines require certain values to be placed into the Z80 microprocessor registers. In Chapter 5 you learned about the 8502 microprocessor registers: A, X, Y, Status (PSW), Stack Pointer (S) and Program Counter (PC). The Z80 also has applicable registers. The Z80 registers are named as follows:

| | |
|---|---|
| A | (Accumulator) |
| BC | |
| DE | |
| HL | |
| PSW | (Status Word) |
| IX | (X register) |
| IY | (Y index register) |
| PC | (program counter)' |
| SP | (stack pointer) |

Certain registers can be used as a register pair, to represent a 16 bit address, or as single 8-bit register. The Z80 has a duplicate set of registers for interrupt processing.

For more detailed information on the Z80 microprocessor consult the ''Suggestions for Further Reading.'' For more detailed information on the CP/M system, refer to the *Commodore 128 System Guide* to receive the full set of CP/M Plus documentation, written by Digital Research Inc., through Commodore.

# COMMODORE 128 CP/M BIOS ROUTINES

The Commodore 128 CP/M system has a set of routines called the CP/M BIOS, which handle the low level input/output operations of the system. Each of these routines can be accessed via the CP/M BIOS jump table below. The jump vector numbers 0 through 29 are the CP/M BIOS jump vectors. The 30th jump vector is for system dependent User Functions. These are discussed following the CP/M BIOS routines.

The 8502 BIOS routines are a subset of the system dependent User Functions (specifically, User Function 4) contained in the Commodore 128 CP/M system. Many of the User Functions have subfunctions which require certain parameters to be passed through the Z80 registers. This is discussed in detail in the User Function section in this appendix. Examples of calling the CP/M BIOS and User Function routines are provided in Part II of this appendix.

| NO. | INSTRUCTION | DESCRIPTION |
|---|---|---|
| 0 | JMP BOOT | Perform cold start initialization |
| 1 | JMP WBOOT | Perform warm start initialization |
| 2 | JMP CONST | Check for console input character ready |
| 3 | JMP CONIN | Read Console Character in |
| 4 | JMP CONOUT | Write Console Character out |
| 5 | JMP LIST | Write List Character out |
| 6 | JMP AUXOUT | Write Auxiliary Output Character |
| 7 | JMP AUXIN | Read Auxiliary Input Character |
| 8 | JMP HOME | Move to Track 00 on Selected Disk |
| 9 | JMP SELDSK | Select Disk Drive |
| 10 | JMP SETTRK | Set Track Number |
| 11 | JMP SETSEC | Set Sector Number |
| 12 | JMP SETDMA | Set DMA Address |
| 13 | JMP READ | Read Specified Sector |
| 14 | JMP WRITE | Write Specified Sector |
| 15 | JMP LISTST | Return List Status |
| 16 | JMP SECTRN | Translate Logical to Physical Sector |
| 17 | JMP CONOST | Return Output Status of Console |
| 18 | JMP AUXIST | Return Input Status of Aux. Port |
| 19 | JMP AUXOST | Return Output Status of Aux. Port |
| 20 | JMP DEVTBL | Return Address of Char. I/O Table |
| 21 | JMP DEVINI | Initialize Char. I/O Devices |
| 22 | JMP DRVTBL | Return Address of Disk Drive Table |
| 23 | JMP MULTIO | Set Number of Logically Consecutive sectors to be read or written |
| 24 | JMP FLUSH | Force Physical Buffer Flushing for user-supported deblocking |
| 25 | JMP MOVE | Memory to Memory Move |
| 26 | JMP TIME | Time Set/Get signal |
| 27 | JMP SELMEN | Select Bank of Memory |
| 28 | JMP SETBNK | Specify Bank for DMA Operation |
| 29 | JMP XMOVE | Set Bank When a Buffer is in a Bank other than 0 or 1 |
| 30 | JMP USERF | Reserved for System Implementor |
| 31 | JMP RESERV1 | Reserved for Future Use |
| 32 | JMP RESERV2 | Reserved for Future Use |

CP/M 3 BIOS Jump Vector Table

0 BOOT

Bank: 0
Input: None
Output: None
Function: This code does all of the hardware initialization, sets up zero page, prints any sign-on message and loads the CCP and then transfers control to the CCP.

1 WBOOT

Bank: 0 or 1
Input: None
Output: None
Function: This code sets up page zero, reloads the CCP and then executes the CCP.

2 CONST

Bank: 0 or 1
Input: None
Output: A = 0FFH if console character
A = 00H if no console character
Function: Checks the console input status of the current console devices. If any of the devices have a character available, FFH is returned, otherwise 00H is returned.

3 CONIN

Bank: 0 or 1
Input: None
Output: A = ASCII console character
Function: Reads a character from any ONE of the assigned console input devices. A scan of each assigned device is done until an input character is found. The character is returned in the A register.

4 CONOUT

Bank: 0 or 1
Input: C = ASCII character to display
Output: None
Function: Sends the character in C to ALL devices that are currently assigned to the console. It waits for all assigned devices to accept a character before exiting.

5 LIST

Bank: 0 or 1
Input: C = ASCII character to print
Output: None
Function: Sends the character in C to ALL devices that are currently assigned to the LIST device. It waits for all assigned devices to accept a character before exiting.

6 AUXOUT

Bank: 0 or 1
Input: C = ASCII Character to send to AUX device
Output: None
Function: Sends the character in C to ALL devices that are currently assigned to the AUXOUT device. It waits for all assigned devices to accept a character before exiting.

7 AUXIN

Bank: 0 or 1
Input: None
Output: A = ASCII character from AUX device
Function: Reads a character from any ONE of the assigned AUXIN devices. A scan of each assigned device is done until an input character is found. The character is returned in the A register.

8 HOME

Bank: 0
Input: None
Output: None
Function: Homes the head on the currently selected disk drive. This function sets the current track to 0 and does not move the head of the disk.

9 SELDSK

Bank: 0
Input: C = Disk Drive (0–15) (A = 0)
E = Initial Select Flag (LSB)
Output: HL = Address of Disk Parameter Header (DPH) if drive exists.
HL = 000H if drive does not exist.
Function: Selects the disk drive whose address is in C as the current drive for all further disk operations. If the LSB of the E register is a zero, then this is the first logging of this disk. The disk type (C64 CP/M, MFM or C128 CP/M) is checked and the DPB parameters adjusted for the diskette currently in the drive.

10 SETTRK

Bank: 0
Input: BC = Track number
Output: None
Function: Register pair BC contains the track number to be used in the subsequent disk access. This value is saved.

11 SETSEC

Bank: 0
Input: BC = Sector number
Output: None
Function: Register pair BC contains the sector number to be used in the subsequent disk access. This value is saved. The value in BC is the value returned by the sector translation routine (in HL).

12 SETDMA

Bank: 0
Input: BC = Direct memory access address
Output: None
Function: The value in BC is saved as the current DMA address. This is the address where ALL disk reads or writes occur. The DMA address that is set is used until it is changed by a future call to this routine to change it.

13 READ

Bank: 0
Input: None
Output: A = 000H if no errors
A = 001H if nonrecoverable error
A = 0FFH if media has changed
Function: Reads the sector addressed by the current disk, track and sector to the current DMA address. If the data is read with no errors then A = 0 on return. If an error occurs, the operation is tried several more times, and if a successful read does not occur then A is set to 001H. A test for media change is performed each time this routine is called and A is set to −1 if the media has been changed.

14 WRITE

Bank: 0
Input: C = Deblocking code (not used)
Output: A = 000H if no errors
A = 001H if nonrecoverable error
A = 002H if disk is read only
A = 0FFH if media has changed
Function: Writes the sector addressed by the current disk track and sector from the current DMA address. If the data is written with no errors, then A is set to 0 on return. If an error occurs, the operation is tried several more times, and if a successful write does not occur, then A is set to 001H. A test for media change is performed each time this routine is called and A is set to −1 if the media has been changed. Also, if an attempt is made to write to a read-only disk, then the A register is set to 002H.

15 LISTST

Bank: 0 or 1
Input: None
Output: A = 00H if list device is not ready to accept a character.
A = 0FFH if list device is ready to accept a character.
Function: This routine scans the currently assigned list devices and returns with A set to 0FFH if ALL assigned devices are ready to accept a character. If any assigned device is not ready then A is set to 00H.

16 SECTRN

Bank: 0
Input: BC = Logical sector number (0-n)
DE = Translation table address (from DPB)
Output: HL = Physical sector number
Function: This routine converts the physical sector number to a logical sector number. If no translation is needed then it moves the BC register to HL and returns.

17 CONOST

Bank: 0 or 1
Input: None
Output: A = 0FFH if Ready
A = 000H if not Ready
Function: This routine scans the currently assigned console devices and returns with A set to 0FFH if ALL assigned devices are ready to accept a character. If any assigned device is not ready then A is set to 000H.

18 AUXIST

Bank: 0 or 1
Input: None
Output: A = 0FFH if console character present
A = 000H if no console character
Function: Checks the status of the current AUXIN device. If any of the devices have a character available, 0FFH is returned, otherwise 000H is returned.

19 AUXOST

Bank: 0 or 1
Input: None
Output: A = 0FFH if ready
A = 000H if not ready
Function: This routine scans the currently assigned AUXOUT devices and returns with A set to 0FFH if ALL devices are ready to accept a character. If any assigned device is not ready then A is set to 00H.

20 DEVTBL

Bank: 0 or 1
Input: None
Output: HL = address of character I/O table
Function: This routine returns the address of the Character I/O table. This table is used to name each of the driver modules and set/control the baud rate and XON/XOFF logic for each driver. Note: the device drive mechanism is used to replace the IOBYTE used with CP/M 2.2.

21 DEVINI

Bank: 0 or 1
Input: C = device number
Output: None
Function: Initializes the physical character device specified in the C register to the BAUD rate in the DEVTBL.

22 DRVTBL

Bank: 0
Input: None
Output: HL = address of the drive table
Function: Returns the address of the drive table in HL (NOTE: first instruction MUST be LXI H, DRVTBL). The drive table is a list of 16 word pointers that point to the DPH for that drive. If a drive is not present in the system, then the pointer for that drive is set to zero.

23 MULTIO

Bank: 0
Input: C = multisector count
Output: None
Function: The multisector count is set before the track, sector, and DMA address and the read/write of the sectors occur. A maximum of 16K can be transferred by each multisector count.

24 FLUSH

Bank: 0
Input: None
Output: A = 000H if no errors
A = 001H if nonrecoverable error
A = 002H if disk is read-only
A = 0FFH if media has changed
Function: This routine is used only if blocking/deblocking is done in the BIOS. This code ALWAYS returns with A = 000H.

25 MOVE

Bank: 0 or 1
Input: HL = destination address
DE = source address
BC = count
Output: HL = HL(in) + BC(in)
DE = DE(in) + BC(in)
Function: Moves a block of data. Data to be moved is to/from the current memory bank (or common) unless the XMOVE routine is called first, then the move is an interbank data movement.

26 TIME

Bank: 0 or 1
Input: C = 000H (Time Get) / 0FFH (Set Time)
Output: None

Function: This function is called with C = 000H if the system time in the SCB needs to be updated by the clock. If C = 0FFH, then the time in the SCB has just been updated and the clock is set to the SCB time. NOTE: HL and DE MUST be preserved.

27 SELMEM

Bank: 0 or 1
Input: A = memory bank
Output: None
Function: Changes the current memory bank. This code MUST be in common memory.

| **NOTE:** ONLY A can be changed. |
|---|

28 SETBNK

Bank: 0
Input: A = DMA memory bank
Output: None
Function: Sets the DMA bank for the next READ/WRITE operation.

29 XMOVE

Bank: 0
Input: B = destination bank
C = source bank
Output: None
Function: Provides the system with the ability to perform memory-to-memory DMA through the entire system space.

30 USERF

Banks 0 or 1
Input: A = function number, L = subfunction number
Output: Outputs depend on the called function or subfunction.
Function: This calls the user functions and 8502 BIOS routines. which are defined in the Commodore 128 System Dependent User Function section.

31 RESERV1

Bank: N/A
Input: N/A
Output: N/A
Function: Not available to the user.

32 RESERV2

Bank: N/A
Input: N/A
Output: N/A
Function: Not available to the user.

# DATA STRUCTURES

## SYSTEM CONTROL BLOCK—(SCB)

The System Control Block is a 100-byte data structure. The data structure is used as the basic communication between the various modules that make up the CP/M Plus system. The contents of the data structure are system parameters and variables.

## DRIVE TABLE (DRVTBL)

### PH0 through DPH15

A list of 16 word pointers (reverse-byte format). The first pointer (DPH0) is drive A and the last pointer (DPH15) is drive P. The pointers point to the XDPH for that disk drive. Any drive that is not in the system has its pointers set to zero.

## EXTENDED DISK PARAMETER HEADER (XDPH) (NORMAL DPH WITH A HEADER)

| WRT | READ | LOGIN | INIT | TYPE | UNIT | XLT | -0- | MF | DPB | CSV | ALV | DIRBCB | DTABCB | HASH | HBANK |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 16b | 16b | 16b | 16b | 8b | 8b | 16b | 72b | 8b | 16b | 16b | 16b | 16b | 16b | 16b | 8b |
| –10 | –8 | –6 | –4 | –2 | –1 | 0 | 2 | 11 | 12 | 14 | 16 | 18 | 20 | 22 | 24 |

| | |
|---|---|
| WRT | Contains the address of the sector write routine for this drive. |
| READ | Contains the address of the sector read routine for this drive. |
| LOGIN | Contains the address of the login routine for this drive. |
| INIT | Contains the address of the first time initialization routine for this drive. |
| TYPE | This byte is used by the BIOS to keep track of density and media type. |
| UNIT | Contains the drive number relative to the disk controller. |
| XLT | Contains the address of the sector translation table or zero if none. |
| -0- | BDOS scratch area (9 bytes). |
| MF | Media flag cleared to zero if disk logged in. BIOS sets to 0FFH if media has changed. |
| DPB | Contains a pointer to the current DPB that describes the current media type. |
| CSV | Contains a pointer to directory checksum area (one per disk drive). |
| ALV | Contains a pointer to allocation vector area (one per disk drive). |
| DIRBCB | Contains a pointer to a single directory Buffer Control Block (BCB). |
| DTSBCB | Contains a pointer to a single data Buffer Control Block (BCB). |
| HASH | Contains a pointer to an optional directory hashing table (FFFFH is not used). |
| HBANK | Contains a bank number of the directory hashing table. |

## DISK PARAMETER BLOCK—DPB

| SPT | BSH | BLM | EXM | DSM | DRM | AL0 | AL1 | CKS | OFF | PSH | PHM |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 16b | 8b | 8b | 8b | 16b | 16b | 8b | 8b | 16b | 16b | 8b | 8b |

| | |
|---|---|
| SPT | Number of 128 records per track |
| BSH | Data allocation block shift factor |
| BLM | Block mask |
| EXM | Extent mask |
| DSM | Number of allocation block on disk minus one |
| DRM | Number of directory entries minus one |
| AL0 | First byte: directory block allocation vector. Filled from MSB to LSB |
| AL1 | Second byte: (Up to 16 allocation blocks can be used for the directory) |
| CKS | Size of the directory check vector, (DRM + 1)/4 |
| OFF | Number of reserved tracks at the beginning of the disk |
| PSH | Physical record shift factor |
| PHM | Physical record mask |

## BUFFER CONTROL BLOCK (LRU CONTROL BLOCK—BCB)

| DRV | REC# | WFLG | 00 | TRACK | SECTOR | BUFFAD | BANK | LINK |
|---|---|---|---|---|---|---|---|---|
| 8b | 24b | 8b | 8b | 16b | 16b | 16b | 8b | 16b |

| | |
|---|---|
| DRV | Drive associated with this record. Set to 0FFH when not used. |
| REC# | Contains the absolute sector number of the buffer. |
| WFLG | Set to 0FFH when buffer contains data that must be written to disk. |
| 00 | Scratch byte used by BDOS |
| TRACK | Physical track address of buffer. |
| SECTOR | Physical sector address of buffer. |
| BUFFAD | Address of the buffer associated with this BCB. |
| BANK | Bank number of buffer associated with this BCB. |
| LINK | Contains the address of the next BCB in this linked list. Set to zero if last. |

# COMMODORE 128 SYSTEM DEPENDENT USER FUNCTIONS

The Z80 machine language routines discussed in this section pertain explicitly to the Commodore 128 Personal Computer. These routines are not part of the standard CP/M system that is transportable from one manufacturer's computer to another. They are written to run only on the Commodore 128.

Most of these customized C128 Z80 routines require certain parameters to be passed from the user into the appropriate Z80 registers. All these routines, listed by

function number, require that the function number be placed in the Z80 A register (accumulator). Many of these routines have subfunctions which require the user to place a value in the eight bit L register, which is used to call the appropriate subfunction. All other additional required parameters are noted if they are necessary. This section follows the same format as the preceding CP/M BIOS section:

a) Function Name
b) Input Parameters
c) Output Parameters
d) Brief Description
e) Additional Information

0 a) User Function 0: Read any Byte in RAM Bank 00
b) A = 0 (function number)
DE = 16 bit address to be read
c) C = Value read from address in RAM Bank 0
A = 0 on RETurn from routine
d) This reads a value from RAM BANK 0 ($0–$0FFF, $1000–$FFFF RAM) and places it in register C.

I. a) User Function 1: Write Function in RAM Bank 0
b) A = 1 (function number)
DE = 16-bit address where write operation occurs
C = Value to be written to the address in Bank 0
c) A = 0 on RETurn from routine if write is successful
A = –1 ($FF) on RETurn from routine if write is unsuccessful
d) This function writes the value in register C to RAM in bank 0 specified by the address in register pair DE. An error is flagged if a write to ROM ($0–$DFFFH) or the top page of memory ($FF00–$FFFF) is attempted.

II. a) User Function 2: Keyscan Function
b) A = 2 (function number)
c) B = –1 ($FF) if no key is pressed
B = Matrix position in the keyboard matrix table (if pressed)
A = Contents (character value) of matrix position whether A is lower- or uppercase or containing the **CONTROL** key.
HL = Address (pointer) where A (contents of matrix position) is found in memory
Address = Tablestart + (4 * B) + C (bits 1 and 0)
C = Returns control code information, where each bit has a specific meaning as follows:

Bits 1 & 0

| | | |
|---|---|---|
| 0 | 0 | = lower case |
| 0 | 1 | = upper case |
| 1 | 0 | = shifted |
| 1 | 1 | = control key |

C = Bit 2 1 = control key down, 0 = up
Bit 3 Not implemented
Bit 4 1 = Right shift key down, 0 = up
Bit 5 1 = C key is active (not necessarily down)
Bit 6 Not implemented
Bit 7 1 = left shift key down, 0 = up

d) The keyscan function allows the user to bypass the normal I/O BIOS keyboard processing and check for a particular key or key sequence being pressed.
e) Additional Information—Important Addresses:
$FD09 = Pointer to Tablestart (low byte first)

Each ASCII character has four coded definitions. Each key has a defined code for the following:

a) lower case
b) upper case
c) shifted key and character
d) control key and character

These four definitions are labeled in columns in the ASCII table.

```
                    ASCII$TBL:    A    B    C    D

1292  7F7F7F16      DB   7FH,7FH,7FH,16H     ; INS DEL   matrix 0 position
1296  0D0D0D0D      DB   0DH,0DH,0DH,0DH     ; RETURN
129A  06060101      DB   06H,06H,01H,01H     ; LF RT
129E  86868787      DB   86H,86H,87H,87H     ; F7 F8
12A2  80808181      DB   80H,80H,81H,81H     ; F1 F2
12A6  82828383      DB   82H,82H,83H,83H     ; F3 F4
12AA  84848585      DB   84H,84H,85H,85H     ; F5 F6
12AE  1717171A      DB   17H,17H,17H,1AH     ; UP DOWN

12B2  333323A2      DB   33H,33H,23H,0A2H    ; 3 #
12B6  77575717      DB   77H,57H,57H,17H     ; W
12BA  61414101      DB   61H,41H,41H,01H     ; A
12BE  343424A3      DB   34H,34H,24H,0A3H    ; 4 $
12C2  7A5A5A1A      DB   7AH,5AH,5AH,1AH     ; Z
12C6  73535313      DB   73H,53H,53H,13H     ; S
12CA  65454505      DB   65H,45H,45H,05H     ; E
12CE  00000000      DB   00H,00H,00H,00H     ; (LF SHIFT)

12D2  353525A4      DB   35H,35H,25H,0A4H    ; 5 %
12D6  72525212      DB   72H,52H,52H,12H     ; R
12DA  64444404      DB   64H,44H,44H,04H     ; D
12DE  363626A5      DB   36H,36H,26H,0A5H    ; 6 &
12E2  63434303      DB   63H,43H,43H,03H     ; C
12E6  66464606      DB   66H,46H,46H,06H     ; F
```

```
                ASCII$TBL:     A    B    C    D

12EA  74545414        DB   74H,54H,54H,14H     ; T
12EE  78585818        DB   78H,58H,58H,18H     ; X

12F2  373727A6        DB   37H,37H,27H,0A6H    ; 7 '
12F6  79595919        DB   79H,59H,59H,19H     ; Y
12FA  67474707        DB   67H,47H,47H,07H     ; G
12FE  383828A7        DB   38H,38H,28H,0A7H    ; 8 (
1302  62424202        DB   62H,42H,42H,02H     ; B
1306  68484808        DB   68H,48H,48H,08H     ; H
130A  75555515        DB   75H,55H,55H,15H     ; U
130E  76565616        DB   76H,56H,56H,16H     ; V

1312  39392900        DB   39H,39H,29H,00H     ; 9 )
1316  69494909        DB   69H,49H,49H,09H     ; I
131A  6A4A4A0A        DB   6AH,4AH,4AH,0AH     ; J
131E  30303000        DB   30H,30H,30H,00H     ; 0

1322  6D4D4D0D        DB   6DH,4DH,4DH,0DH     ; M
1326  6B4B4B0B        DB   6BH,4BH,4BH,0BH     ; K
132A  6F4F4F0F        DB   6FH,4FH,4FH,0FH     ; O
132E  6E4E4E0E        DB   6EH,4EH,4EH,0EH     ; N

1332  2B2B2B00        DB   2BH,2BH,2BH,00H     ; +
1336  70505010        DB   70H,50H,50H,10H     ; P
133A  6C4C4C0C        DB   6CH,4CH,4CH,0CH     ; L
133E  2D2D2D00        DB   2DH,2DH,2DH,00H     ; -
1342  2E2E3E00        DB   2EH,2EH,3EH,00H     ; , <
1346  3A3A5B7B        DB   3AH,3AH,5BH,7BH     ; : [ {
134A  40404000        DB   40H,40H,40H,00H     ; @
134E  2C2C3C00        DB   2CH,2CH,3CH,00H     ;,<

1352  23232360        DB   23H,23H,23H,60H     ; POUND
1356  2A2A2A00        DB   2AH,2AH,2AH,00H     ; *
135A  3B3B5070        DB   3BH,3BH,5DH,7DH     ; ; ] }
135E  000000F5        DB   00H,00H,00H,0F5H    ; CLEAR/HOME
1362  00000000        DB   00H,00H,00H,00H     ; (RT SHIFT)
1366  3D3D3D7E        DB   3DH,3DH,3DH,7EH     ; = ~
136A  5E5E7C7C        DB   5EH,5EH,7CH,7CH     ; PI
136E  2F2F3F5C        DB   2FH,2FH,3FH,5CH     ; / ? \

1372  313121A0        DB   31H,31H,21H,0A0H    ; 1
1376  5F5F5F7F        DB   5FH,5FH,5FH,7FH     ; <-
137A  09153000        DB   09H,15H,30H,00H     ; (CONTROL) SOUND1
                                               SOUND2
137E  323222A1        DB   32H,32H,22H,0A1H    ; 2 "
1382  20202000        DB   20H,20H,20H,00H     ; SPACE
1386  21200000        DB   21H,20H,00H,00H     ; (COMMODORE) SOUND3
138A  71515111        DB   71H,51H,51H,11H     ; Q
138E  000000F0        DB   00H,00H,00H,0F0H    ; RUN STOP
```

```
                    ASCII$TBL:      A     B     C     D

1392  9F9F9F9F          DB    9FH,9FH,9FH,9FH     ; /HELP/
1396  383838B7          DB    38H,38H,38H,0B7H    ; /8/
139A  353535B4          DB    35H,35H,35H,0B4H    ; /5/
139E  09090900          DB    09H,09H,09H,00H     ; /TAB/
13A2  323232B1          DB    32H,32H,32H,0B1H    ; /2/
13A6  343434B3          DB    34H,34H,34H,0B3H    ; /4/
13AA  373737B6          DB    37H,37H,37H,0B6H    ; /7/
13AE  313131B0          DB    31H,31H,31H,0B0H    ; /1/

13B2  1B1B1B00          DB    1BH,1BH,1BH,00H     ; /ESC/
13B6  2B2B2B00          DB    2BH,2BH,2BH,00H     ; /+/
13BA  2D2D2D00          DB    2DH,2DH,2DH,00H     ; /-/
13BE  0A0A0A0A          DB    0AH,0AH,0AH,0AH     ; /LINE FEED/
13C2  0D0D0DFF          DB    0DH,0DH,0DH,0FFH    ; /ENTR/
13C6  363636B5          DB    36H,36H,36H,0B5H    ; /6/
13CA  39393900          DB    39H,39H,39H,00H     ; /9/
13CE  333333B2          DB    33H,33H,33H,0B2H    ; /3/
13D2  00000000          DB    00H,00H,00H,00H     ; /ALT/
13D6  30303000          DB    30H,30H,30H,00H     ; /0/
13DA  2E2E2E00          DB    2EH,2EH,2EH,00H     ; /./
13DE  05050512          DB    05H,05H,05H,12H     ; /UP/
13E2  18181803          DB    18H,18H,18H,03H     ; /DN/
13E6  1313138D          DB    13H,13H,13H,08DH    ; /LF/
13EA  0404048E          DB    04H,04H,04H,08EH    ; /RT/
13EE  F1F1F1F2          DB    0F1H,0F1H,0F1H,0F2H; /NO SCROLL/
                ;
                ;     LOGICAL COLOR TABLE (USED WITH ESC ESC ESC CHAR)
                ;                          (WHERE CHAR IS 50H TO 7FH)
                ;
13F2  00112233          DB    000H,011H,022H,033H
13F6  44556677          DB    044H,055H,066H,077H
13FA  8899AABB          DB    088H,099H,0AAH,0BBH
13FE  CCDDEEFF          DB    0CCH,0DDH,0EEH,0FFH
```

III. a) User Function 3: Execute a Z80 ROM function

b) A = 3
L = subfunction number
Additional input parameters may be necessary

c) Most of the ROM functions do not output values; they instead perform an action, which in this user function is assumed to be the output.

d) This function executes a Z80 ROM function. These ROM routines primarily perform screen manipulation routines.
All subfunction numbers are even numbers. The first 80 subfunctions are screen manipulating routines for the 40- and 80-column display. Only the 80-column subfunction is listed here. Add 2 to the 80-column subfunction number to get the corresponding 40-column subfunction number.

III. 0a) Subfunction 0: Write Character
b) wr$char
c) Register D = character to write auto advance cursor to next position
d) ——

III. 4a) Subfunction 4: Cursor Position
b) cursor$pos
c) Register D = row value
Register E = column value
d) This subfunction sets the current position of the cursor on the 80-column screen.

III. 8a) Subfunction 8: Cursor Up one position
b) cursor$up
c) no values returned
d) This subfunction moves the cursor up one row on the current screen (40 or 80), but not past the top of the screen.

III. 12a) Subfunction 12: Cursor down one row
b) cursor$down
c) no values returned
d) This subfunction moves the cursor down one row on the current screen (40 or 80). The screen scrolls down if on bottom line.

III. 14a) Subfunction 14: Not Implemented
b) ——
c) ——

III. 16a) Subfunction 16: Cursor left one column
b) cursor$left
c) no values returned
d) This subfunction moves the cursor left one column, but not past the left margin.

III. 20a) Subfunction 20: Cursor right one column
b) cursor$rt
c) no values returned.
d) This subfunction moves the cursor right one column, but not past the right edge of the screen.

III. 24a) Subfunction 24: Execute a carriage return
b) do$cr
c) No values returned.
d) This subfunction executes a carriage return and places the cursor at the left margin.

III. 28a) Subfunction 28: Clear to end of line
b) CEL
c) No values returned.
d) This subfunction clears the cursor row starting where the cursor is currently located and ending at the end of the line.

III. 32a) Subfunction 32: Clear to end of screen
b) CES
c) No values returned.
d) This subfunction clears the screen starting where the cursor is currently located through the end of screen.

III. 36a) Subfunction 36: Character insert
b) char$ins
c) No values returned.
d) This subfunction inserts a character at the current cursor position, the last character on the line is lost.

III. 40a) Subfunction 40: character delete
b) char$del
c) No values returned.
d) This subfunction deletes a character at the current cursor position, and places a space in the last position in the line.

III. 44a) Subfunction 44: Line insert
b) line$ins
c) No values returned.
d) This subfunction inserts a line of spaces on the current cursor row, the current cursor row is moved down one and the last line (24) is lost.

III. 48a) Subfunction 48: Line delete
b) line$del
c) No values returned.
d) This subfunction deletes a line at the character row marked by the current cursor position. The bottom line (24) is filled with spaces.

III. 50a) Subfunction 50: Not Implemented
b) ——
c) ——
d) ——

III. 52a) Subfunction 52: Set Cursor Color
b) color
B register returns color code value
c) No values returned.
d) This function sets the current cursor color code with the value in color.

III. 56a) Subfunction 56: Set 80-column Attributes
b) attr
B = bit to set/clear
C = bit value
c) No values returned.
d) This function enables/disables the 8563 attributes for the 80-column screen. Attributes are extra screen features such as foreground R,G,B and I, character blinking, underlining, reverse video and the alternate character set. The 40-column function (subfunction 58) controls only reverse video.

III. 60a) Subfunction 60: Read Character Attribute (80)
b) rd$chr$atr
D = 8563 character row
E = 8563 character column
c) D = 8563 character row
L = 8563 character column
B = character value
C = attribute bit pattern of selected character location
d) This function reads the attribute byte of the selected character row and column on the 8563 screen, and returns the true RGBI color. The corresponding 40-column subfunction (62) controls only reverse video.

III. 64a) Subfunction 64: Write Character Attribute (80)
b) wr$chr$atr
B = character value
C = attribute
c) No values returned
d) This function writes an attribute byte of the selected character value on the 8563 screen.

III. 68a) Subfunction 68: Read Color
b) rd$color
c) A = character color
B = background color
C = border color (40-column only)
D = 8563 attribute
d) This function reads the current color of the 8563 color sources. Subfunction 70 returns the VIC color sources.

III. 80a) Subfunction 80: Convert record (GCR only)
b) @trk
c) VIC$trk, VIC$SEC
d) This function returns the physical track and sector from the disk drive, including skew, given the logical track.

III. 82a) Subfunction 82: Check CBM code from disk (GCR only)
b) check$CBM
c) zero flag = 1 if CBM (C128 mode disk) is present in drive
zero flag = 0 if CP/M 2.2 (C64 CP/M disk) is present in drive
A = 0 if single sided
A = $FF if double sided
d) This function detects which type of GCR disk is in the drive. Reads data from t=1, s=0 into buffer at $FE00.

III. 84a) Subfunction 84: Bell Function
b) Sound 1, Sound 2, Sound 3 ($FD10, $FD12, $FD14, respectively)
c) Outputs a bell sound
d) This function outputs a bell sound from the SID chip.
Subfunctions 86–95 are not defined

III. 96a) Subfunction 96: Track 40
b) trk$40
c) @ off 40
d) This function computes the logical (offset) position of the cursor on the physical 80-column screen. The variable @ off40 is computed on 8-character boundaries, and is used in the next subfunction.

III. 98a) Subfunction 98: Set cursor position
b) set$cur$40
c) No value returned.
d) This function calls Track 40 to check the offset between the logical (window) and physical (40-column) screen. It keeps the cursor within the 40-column logical window on the scrolled 80-column virtual screen for VIC screen output.

III. 100a) Subfunction 100: Line Paint
b) line$paint
c) No value returned.
d) This function updates the current character line only if @ off 40 and old $offset are the same. If old $offset =−1, TRACK 40 is called. If @off40 and old $offset are not equal, Screen Paint is called to scroll the 40-column logical screen (window) over the virtual 80-column screen.

III. 102a) Subfunction 102: Screen paint
b) screen$paint
c) No value returned.
d) This function updates (scrolls) the 40-column logical (window) screen across the VIC 80-column virtual screen based on the value of @off40. This and the three previous subfunctions (96, 98, 100) are intertwined to make the scrolled 40-column logical screen display as fast as possible. Normally you would only call Set Cursor Position or Line Paint, since they call the other related routines, if necessary. See file CXROM.ASM listing on the DRI disks that come with the documentation for the source listings.

III. 104a) Subfunction 104: Print Message (BOTH)
b) prt$msg$both
c) No value returned.
d) This function prints simultaneous output to both screens, displaying the string pointed to by the top value on the stack. Place the address of the string on the stack, and terminate the string with a zero. Execution resumes with the byte following the zero terminator. This works like an "in-line Print"

III. 106a) Subfunction 106: (Print Message (BOTH)
b) prt$de$both
c) No value returned.

d) This function works like the previous one, except the start address of the string is taken from DE, and execution resumes with the return address from the stack.

III. 108a) Subfunction 108: Incripted Messages
b) update$it
c) No value returned
d) This function displays incripted messages.
Subfunction 110 is not implemented.

III. 112a) Subfunction 112: ASCII to PET ASCII Conversion
b) ASCII$pet, B register = ASCII character ($20–$7F)
c) A register = converted PET ASCII character.
d) This function performs a standard ASCII to Pet ASCII conversion on the characters printed to the screen (from any input device)
Control codes are not translated.

III. 114a) Subfunction 114: Place 40-column cursor at specified address
b) cur$adr$40
c) HL = address of cursor on screen
DE = cursor line (row) start address
BC = # of characters to end of line (<80, not counting cursor)
d) This function places the cursor at the address specified in HL (in RAM bank 0). This address is of the logical screen, not the virtual one.

III. 116a) Subfunction 116: Place 80-column cursor at specified address
b) cur$adr$80
c) HL = address of cursor on screen
DE = cursor line (row) start address
BC = # of characters to end of line (<80, not counting cursor)
d) This function places the cursor at the address specified in HL (in 8563 RAM).

III. 118a) Subfunction 118: Look up color
b) look$color, B = color code ($30–$3F), C = max. Value of color code.
c) HL = pointer to logical color table (lower nybble = 80, high nybble = 40-
B = $0 (Character color)
$10 (Background color)
$20 (Border color)
d) This function sets the 8563 screen colors to the VIC screen colors.
Subfunction 120 is not defined.

III. 122a) Subfunction 122: Block Fill
b) blk$fill, put start address on stack (8563)
BC = # of bytes to fill
D = fill character
E = attribute
c) No value returned.
d) This function fills a 256-byte block with data specified in the D register.

III. 124a) Subfunction 124: Block move
b) blk$move, place source address (in 8563 RAM) on the stack
DE = destination address (in 8563 RAM)
BC = count
c) No value returned.
d) This function moves 256-byte blocks from one memory block to another in 8563 RAM.

**NOTE:** Subfunctions 122 and 124 and 126 must be direct Z80 calls. The example routines provided in this section for subfunctions 122, 124 and 126 will not work. They are usually not user-accessible, and must run in BANK 0.

III. 126a) Subfunction 126: Character Install
b) chr$inst, stack = 8563 RAM address to install character definition
DE = address of system memory (C128 RAM) bank 0) of new character definition (8 bytes per character).
B = number of consecutive characters
c) No value
d) This function installs a user-defined character in 8563 RAM.

IV. a) User Function 4: 8502 BIOS Functions
b) A = 4 (function number)
L = Subfunction number (-1 through 11)
Additional input parameters may be required
c) Outputs depend on each subfunction
d) User function 4 allows you to call 8502 input/output functions that are performed by the 8502 processor. These functions are not part of the standard CP/M system and are completely hardware-dependent. This function enables you to go back and forth between the Z80 and 8502 processors.
e) Each subfunction is discussed in detail using the defined conventions.

IV. –1.a) Subfunction –1: Reboot C128 hardware
b) A = 4
L = –1
c) none
d) This subfunction reboots the Commodore 128 hardware when the value in the L register is equal to –1 ($FF). This subfunction is not normally used. It performs the same actions as pressing reset.

IV. 0.a) Subfunction 0: Initialize 8502 BIOS
b) A = 4
L = 0
c1) Sets up interrupt vector $0314–$0319 to vector to the 8502 interrupt handler
c2) Copies ROM interrupt vector to RAM

c3) Sets up PAL and NTSC variables (SYSFREQ)
c4) Closes all open channels
d) This subfunction intializes 8502 system variables, interrupt processing and system frequencies so that the Z80 and 8502 may communicate back and forth. Processor control is given to either the Z80 or the 8502 at one particular time. The processors cannot run simultaneously.

IV. 1.a) Subfunction 1: 1541 Read (GCR format only)
b) A = 4
L = 1
VICTRACK = (1–35)—Variable for track number on disk
VICSECTOR = (0–21)—Variable for sector number on disk
VICDRV = —Variable for disk drive device number where:

| LOWER NYBBLE OF VICTORY BIT VALUE | | DEVICE NUMBER | DRIVE | EQUIVALENT OPEN STATEMENT IN BASIC |
|---|---|---|---|---|
| **0001** | = | **8** | **0** | **OPEN 8,11,15** |
| **0010** | = | **9** | **0** | **OPEN 9,12,16** |
| **0100** | = | **10** | **0** | **OPEN 10,13,17** |
| **1000** | = | **11** | **0** | **OPEN 11,14,18** |

**Values of the Lower Nybble of VICDRV**

c) VICDATA = 11 ($0B) if disk in drive has been changed
VICDATA = 13 ($0D) if read/write or channel error occurs
VICDATA = 0 if read is successful
VICDATA = 15 ($0F) if device is not present
d) This function reads a particular track and sector on the disk in the drive as specified by VICTRACK and VICSECTOR and VICDRV respectively. Data is read into the buffer at $FE00H. The value returned in the variable VICDATA depends on the conditions described in c.
e) Additional Information: This subfunction assumes that both the data and command channels have been opened previously. An error will occur if this routine is called to read from a 1571.

IV. 2.a) Subfunction 2: 1541 Write (GCR format only)
b) A = 4
L = 2
VICTRACK = Same as for 1541 Read
VICSECTOR = Same as 1541 Read
VICDRV = Same as 1541 Read
c) VICDATA = Same as 1541 Read
d) This subfunction writes data to the specified track and sector on the disk in the drive as specified by VICTRACK, VICSECTOR and VICDRV respectively. The value returned in VICDATA depends on the conditions described for the outputs in c. See subfunction 1 for details.

e) Additional Information: This subfunction assumes that both the data and command channels have been opened previously. Data is written from the buffer at $FE00H.

IV. 3.a) Subfunction 3: 1571 Read Set Up (MFM or GCR formats)
b) A = 4
L = 3
* VICTRACK = (1–35)—Variable for track number on disk
* VICSECTOR = (0–21)—Variable for sector number on disk
* VICDRV = —Variable for disk drive device number (See VICDRV table above.)
* = Ranges apply to GCR format only. The ranges are different for MFM disks depending on the manufacturer.
VIC$COUNT = Number of sectors to read (on the track)
c) VICDATA = 11 ($0B) if disk in drive has been changed
VICDATA = 12 ($0C) if drive is not a fast (1571) disk drive
VICDATA = 13 ($0D) if channel error occurs
VICDATA = 15 ($0F) if device is not present
If FAST ANDed with VICDRV = 0 meaning drive is a 1541
If FAST ANDed with VICDRV = 1 meaning drive is a 1571
d) This subfunction sets up the 1571 disk drive for a read operation. However, the data transfer is not performed by the 8502 BIOS. The data is transferred by the Z80.
e) Additional Information: To access the back side of an MFM disk set bit 7 ($80) of VICSECTOR. For MFM formats, a dash between the track and sector on the display window means that the drive accesses the back side of the disk. This is usually performed by the BIOS.

IV. 4.a) Subfunction 4: 1571 Write Set Up (MFM or GCR formats)
b) A = 4
L = 4
* VICTRACK = (1–35)—Variable for track number on disk
* VICSECTOR = (0–21)—Variable for sector number on disk
VICDRV = —Variable for disk drive device number. See VICDRV table on previous page.
VIC$COUNT = Number of sectors to read
* = Ranges apply to GCR format only. The ranges are different for MFM disks depending on the manufacturer.
c) VICDATA = 11 ($0B) if disk in drive has been changed
VICDATA = 12 ($0C) if drive is not a fast (1571) disk drive
VICDATA = 13 ($0D) if channel error occurs
VICDATA = 15 ($0F) if device is not present
If FAST ANDed with VICDRV = 0 meaning drive is a 1541
If FAST ANDed with VICDRV = 1 meaning drive is a 1571
d) This subfunction sets up the 1571 disk drive for a write operation. However, the data is not performed by the 8502 BIOS. The data is transferred by the Z80.

e) Additional Information. This is how the user should select between a 1541 and 1571 Drive. To access the back side of an MFM disk set bit 7 of VICSECTOR. To perform a write operation, the user will have to do so in their application.

IV. 5.a) Subfunction 5: Interrogate 1541 or 1571 Disk Drive

b) A = 4
L = 5
VICDRV = (8–11)—Disk drive device number variable

c) VICDATA = lower four bits returns status for FAST read/write (same as previous VICDATA Disk error codes, which are listed in the Disk Error Status table on the next page).
= upper four bits return sector size for MFM disks

d) This subfunction interrogates the disk drive for the disk sector size (MFM or GCR) and the drive status. In addition, this subfunction initializes the FAST variable, closes and reopens the channel for the corresponding drive, and clears the drive status.

IV. 6.a) Subfunction 6: Query to Disk

b) A = 4
L = 6
VICTRACK = (1–35)—Variable for track number on disk
VICSECTOR = (0–21)—Variable for sector number on disk
VICDRV = —Variable for disk drive device number

c) If error-free
VICDATA = lower four bits returns status for FAST read/write
= upper four bits return sector size for MFM disks and the subfunction inputs 6 bytes into a memory buffer starting at location $FE00 (and ending at $FEFF). These 6 bytes are defined as :

1) $FE00-TRACK STATUS (on track below)
2) $FE01-Number of Sectors
3) $FE02-Logical Track
4) $FE03-Minimum Sector Number (on this track)
5) $FE04-Maximum Sector Number (on this track)
6) $FE05-Physical Interleave

If an error occurs:

VICDATA = 11 ($0B) if disk in drive has been changed
VICDATA = 12 ($0C) if drive is not a fast (1571) disk drive
VICDATA = 13 ($0D) if channel error occurs
VICDATA = 15 ($0F) if device is not present

d) This subfunction queries the disk, and returns the disk status and sector size (if MFM format). In addition, the buffer located between $FE00 and $FEFF receives 6 data items as described above.

## Disk Error Status Table

MD DN I1 I2 D1 D2 D3 D4

MD ..... Mode: 1 = MFM, 0 = GCR.
DN ..... Drive Number.
I1, I2 ..... Sector Size.
a) 00 = 128 bytes
b) 01 = 256 bytes
c) 10 = 512 bytes
d) 11 = 1024 bytes
D1–D4 ..... Controller Status

GCR

000x = Ok.
0010 = Sector not found.
0011 = No Sync.
0100 = Data block not found.
0101 = Data block checksum.
0110 = Format error.
0111 = Verify error.
1000 = Write protect error.
1001 = Header block checksum.
1010 = Data extends into next block.
1011 = Disk ID mismatch/Disk change.
1100 = Drive is not fast (1571).
1101 = Channel Error.
1110 = Syntax.
1111 = No Drive present.

MFM

000x = Ok.
0010 = Sector not found.
0011 = No address mark.
0100 = Unused.
0101 = Data CRC error.
0110 = Format error.
0111 = Verify error.
1000 = Write protect error.
1001 = Header CRC error.
1010 = Unused.
1011 = Disk change.
1100 = Drive is not fast (1571).
1101 = Channel Error.
1110 = Syntax.
1111 = No Drive present.

## Disk Error Status Table

IV. 7.a) Subfunction 7: Print characters to a serial bus printer

b) A = 4
L = 7
VICDRV = Printer number (either 4 or 5)
VICTRACK = Secondary address in which device is opened as
VICDATA = Character to be printed to the serial bus printer (if VICCOUNT = 0)

c) VICDATA = –1 ($FF) if device is not present

d) This subfunction outputs characters to the previously opened serial bus printer.

e) Additional Information: If the secondary address (which is normally 7) is changed, the device is closed and reopened with the new secondary address. If a serial bus error besides device not present occurs, the channel is closed, reopened and the original operation is executed again.
If VICCOUNT is not equal to zero, the data is printed from the buffer pointed to by $FE00. The number of bytes printed is supplied in VICCOUNT.

IV. 8.a) Subfunction 8: Format a 1541 or 1571 Diskette

b) A = 4
L = 8
DRIVE#
FAST

c) VICDATA

d) This subfunction formats a 1541 or 1571 diskette in the appropriate drive. If FAST is enabled, (FAST ANDed with DRIVE# (not equal to 0)), the length of a disk command is fetched from memory buffer location $FE00. The command starting at location $FE01 and ending at the location specified by the length of the command in $FE00 is sent to the drive. For example, if $FE00 = $06, the command in the memory buffer between $FE01 and $FE06 is sent to the drive. All commands have a "U0" preceding them, so only the command from the memory buffer must be supplied.

IV. 9.a) Subfunction 9: User Call to 8502 Code Routine

b) A = 4
L = 9
VICOUNT = ($FD05) low byte address of 8502 routine (pointer to the start of execution of the user routine)
VICDATA = ($FD06) high-byte address of 8502 routine

c) User defined outputs only

d) This is the routine that allows you to call an 8502 machine language subroutine from Z80 mode. The 8502 coded routine is usually user defined. It must execute in RAM bank 0 with the input/output registers enabled. The MMU value = $3E. Control is transferred to the 8502 processor with the KERNAL disabled. If you want to call a C128

KERNAL routine, you must enable the KERNAL after you have transferred control to the 8502. Before you return to the Z80, you must disable the KERNAL again.

When control passes from the Z80 to the 8502 processor, the Z80 is idle. To return control to the Z80 processor, place the customary 8502 RTS instruction at the end of your 8502 coded routine and control is passed back to the Z80.

e) Additional Information: Once control is passed from the Z80 to the 8502, the 8502 is running at the speed of 1 Mhz. You can increase the speed to 2 Mhz to speed up processing on the 8502 side of the computer. However, YOU MUST RETURN TO 1 Mhz SPEED BEFORE RETURNING TO THE Z80 OR A SYSTEM CRASH WILL OCCUR. The nature of the timing of the two processors dictates this. If you don't return to 1 Mhz, the clock cycle timing is thrown off and the system crashes.

IV. 10.a) Subfunction 10: RAM Disk Read
b) A = 4
L = 10
c) Data is transferred to expansion RAM from RAM (BANK 0)
d) All expansion RAM registers must be set up prior to calling this routine.

IV. 11.a) Subfunction 11: RAM Disk Write
b) A = 4
L = 11
c) Data is transferred to expansion RAM from RAM (BANK 0)
d) All expansion RAM registers must be set up prior to calling this routine.

V. a) User Function 5: Read 40/80 Column Key
b) A = 5 (function number)
c) A = Value stored in $D505 (C128 Mode Configuration register)
If bit 7 is high (1), 40/80 key is up, otherwise 40/80 is down.
d) This function returns the value of location $D505, the mode configuration register. Only bit 7 is significant as noted above. The 40/80 key is not in the keyboard matrix table, so this function is dedicated to reading its position.

VI. a) Functions 6 through 254 are not implemented.
b) none
c) HL = number of days (in binary) since 1/1/78

VII. a) User Function 255: System Date
b) A = –1 ($FF)
c) HL = number of days (in binary) since 1/1/78
d) By specifying A = –1, the system date is returned.

# APPENDIX K

## PART II

# CALLING CP/M BIOS, 8502 BIOS AND CP/M USER FUNCTIONS IN Z80 MACHINE LANGUAGE

The Commodore 128 CP/M system allows you to call the CP/M BIOS, 8502 BIOS and CP/M user functions in your own Z80 assembly language programs. However, in order to program in Z80 assembly language, you need either an assembler or machine language monitor. Many Z80 assemblers and monitors are available on the market; however, the full featured Digital Research CP/M Plus (3.0) system now available on the Commodore 128 comes with two Z80 assemblers, MAC and RMAC and CP/M plus documentation. These programs or documentation are not included in the Commodore 128 Personal Computer package; refer to the *Commodore 128 System Guide* for information on obtaining them.

Assuming you have the MAC and RMAC assemblers, you can now enter and assemble Z80 assembly language programs. At this point, this reference guide must make a substantially large assumption about its readers and their knowledge of Z80 assembly language programming. As you probably agree, this reference guide could not possibly introduce Z80 machine language and thoroughly cover it. That is simply beyond the scope of this book, considering how voluminous it is already. Your trusty local bookstore undoubtedly offers several excellent books on Z80 programming. See "Suggestions for Further Reading" for a few Z80 book titles.

Now that all the assumptions are out of the way, here's how to call a Z80 user or BIOS function. First, the user function call.

## CALLING A CP/M SYSTEM USER FUNCTION

As you saw in Part I of this appendix, certain user functions had subfunctions, others did not. User function 4 is the 8502 BIOS call function. The 8502 BIOS functions have 13 (–1–11) subfunctions which perform the machine level 8502 input and output routines. Do not confuse the 8502 BIOS with the CP/M BIOS. The CP/M BIOS comes standard with every CP/M Plus system regardless of the hardware running it. Remember that CP/M was designed to be transportable from one microcomputer to another. Each different microcomputer has its own machine level input and output which it must perform. The 8502 BIOS is completely hardware-dependent and will only run on the Commodore 128's 8502 microprocessor.

Another way of understanding the difference between the two BIOS types is

recalling the CP/M jump vector on page 677 of Appendix K. The first 30 jumps (0–29) are direct calls to CP/M BIOS routines. Jump number 30 is the call to the user functions, of which user function 4 is the 8502 BIOS. The 8502 BIOS is a subset of the CP/M user functions. The user function call is 1 of 31 system routine calls within the CP/M BIOS jump table.

The following example calls user function 2 the keyscan function. The calling routine starts at "waitkey" and the subroutine starts at "user$fun".

```
            MVI useroffset,30
waitkey:
            MVI A,2              ; load A reg. with function no. 2
            CALL user$fun        ;calls subroutine user$fun
            INR B                ;increment B—test for –1 if no key is pressed
            JZ waitkey           ;jump to wait key if none
            DCR B                ;decrement B if B not equal to 0
                                 ;this restores original matrix value
              .
              .
              .
            Rest of Program
              .
              .
user$fun
            PUSH H               ;place HL reg. pair on stack
            LHLD 1               ;load address of jump vector 1 (WBOOT) into HL
            MVI L, useroffset * 3 ;get offset of 90 (30*3) to L
            XTHL                 ;exchange HL with top of stack
            RET                  ;jump to new HL pointer location of routine
```

First, the main program loads the user function number (2) into the A register. To call a user function, place the required input parameter, the user function number, in the A register. If a subfunction is going to be called, like in user function 4 (8502 BIOS), the input parameter for the subfunction number must be placed in L.

The second instruction in the main routine is a CALL to the subroutine user$fun. However, in this example of the keyscan user function, the returned value B is –1 if no key is pressed. If this is the case, B is incremented to zero and the main program jumps to waitkey and scans the keyboard again. Otherwise, the rest of the program continues processing.

When the subroutine is called, the first instruction saves the HL register pair, the address pointer, on top of the stack. The next instruction loads memory location 1 (low) and 2 (high) into register pair HL. The high byte points to the page number that the jump vectors are on and the low byte is always 3 (Jump #1 the warm boot vector). Next the jump number times 3 is loaded into the low byte (L) of register pair HL. This adds the offset of 90 memory locations to base address of the CP/M BIOS jump table, which now points to jump number 30, USERF.

The XTHL instruction exchanges the HL register pair with the top of the stack. This places the computed address on the top of the stack and the entry values of HL back in HL. When the RETurn instruction is reached by the Z80, program control is therefore passed to the USERF vector, entry 30 in the CP/M BIOS jump table. When the function has completed, control returns to the instruction immediately following the CALL User$Fun instruction (INR B).

In order for the routines to be called successfully, the proper required input parameters must be placed in the appropriate registers. The user function number must be placed in the A register, the subfunction number is placed in L, if any. Additional inputs must be placed in the correct register or variable prior to calling the user function.

The above example calls user function 2, the keyscan function. To call any other user function, load the subfunction number into the A register. To call a subfunction, load the L register with a subfunction number on the beginning of the main (calling) program as follows:

```
MVI L, subfun
```

The way this program is written, the value in L, which you will load at the beginning of the program with the above instruction, is placed back in L from the stack when the XTHL is reached at the end of the subroutine. Use this example as a template when calling other subfunctions. Make sure the proper inputs are present in the correct locations prior to calling the user function.

## CALLING A CP/M BIOS ROUTINE

Making a direct BIOS call is different from calling a user function. User function calls always enter jump number 30 in the CP/M BIOS jump table. A direct BIOS call enters any of the first 30 (0–29) jump vectors. These are the input and output routines that are a part of any CP/M system on any microcomputer. The standard method of calling a CP/M BIOS routine is via BIOS function 50. This function handles the banking of the two 64K RAM banks in the C128.

You could call a CP/M BIOS routine similar to the first example, but there is a limitation. With the user function call method, you are only able to call BIOS routines that reside in RAM bank 1. The TPA resides in bank 1, while the Z80 function code is stored in RAM bank 0. If you try to call a direct BIOS routine in bank 0, the system will crash because bank 1 is in context. This is why it is important to make direct CP/M BIOS calls through user function 50. Refer to the Digital Research CP/M Plus documentation for more details on user function 50.

Here's a program example that illustrates how to call a CP/M BIOS function.

```
main
        MVI C,50        ;store function no. 50 in reg. C
        LXI D,bios$pb   ;load immediate a 16 bit pointer in DE
        CALL 5          ;standard BIOS call
          .
          .
          .
```

```
        Rest of Program
          .
          .
          .
        RET
bios$pb:                ;BIOS variable table
        db FUNNUM       ;BIOS function no. variable
        db AREG         ;temporary A reg. storage
        dw BCREG        ;temp BC reg. pair storage
        dw DEREG        ;temp DE reg. pair storage
        dw HLREG        ;temp HL reg. pair storage
```

The first instruction in the main routine stores the function number 50 in the C register. The system expects the input parameter C to be the BIOS function number. The next instruction loads the address of the BIOS variable table bios$pb into register pair DE. The third instruction calls BDOS function 50 through the call BDOS vector, the standard BIOS vector through which all direct BIOS functions are called. BDOS function 50 manipulates the banking in and out of RAM banks 0 and 1. This is the recommended way of directly calling a CP/M BIOS function over that of the first example which is designed primarily for calling user functions.

The variable table "bios$pb" contains the necessary input parameters required for BDOS function 50. The "db" 's stand for a byte of storage (like .byte in 8502) while the "dw" 's stand for a 16 bit word (like .word in 8502).

This appendix is the only section of the book that crosses the Z80 programming barrier. It at least points you in the right direction and gives you the "hooks" into the machine level routines of the CP/M system. For more detailed Z80 programming information see the "Suggestions for Further Reading" at the back of the book. For more detailed CP/M information, refer to the Digital Research CP/M Plus documentation.

## MFM DISK FORMATS

The abbreviation MFM stands for Modified Frequency Modulation. This type of disk format is variable and programmable according to the specifications of the particular computer manufacturer.

The Commodore 128 CP/M system supports four standard MFM disk formats. These four formats constitute the majority of CP/M software formats available on the market. This does not mean that the Commodore 128 CP/M system can read every single CP/M disk format in the universe; however, the majority of available CP/M software can run on the Commodore 128, if the particular application is *not* hardware dependant.

The four major formats that are supported are as follows:

a) Epson QX10 (double sided)
a1) Epson QX10 (double sided)
b) IBM-8 (single sided)
b1) IBM-8 (double sided)
c) KayPro IV (double sided)
d) KayPro II (single sided)
e) Osborne (single sided)

Each of these formats is compatible with the Commodore 128 CP/M system. At the present time, the system cannot format these disk types and successfully use them on the host system, but they can be used on the C128 system. This portion of the system is still in development.

The following table lists the parameters that these disk formats are looking for when reading third party CP/M software.

| | Manufacturers | | | | | | |
|---|---|---|---|---|---|---|---|
| | a | b | c | d | e | a1 | b1 |
| **Disk Type** | **DSDD** | **SSDD** | **DSDD** | **SSDD** | **SSDD** | **DSDD** | **DSDD** |
| **Starting pos. (t/s)** | **2/1** | **1/1** | **0/10** | **1/0** | **3/1** | **2/1** | **1/1** |
| **Sector size (bytes)** | **256** | **512** | **512** | **512** | **1024** | **512** | **512** |
| **Number of sec/trk** | **32** | **8** | **10** | **10** | **5** | **20** | **8** |
| **Number of tracks** | **40** | **40** | **80** | **40** | **40** | **40** | **80** |
| **Block alloc size** | **2048** | **1024** | **2048** | **1024** | **1024** | **2048** | **2048** |
| **# of dir entries** | **128** | **64** | **128** | **64** | **64** | **128** | **64** |
| **# of resvd tracks** | **2** | **1** | **1** | **1** | **3** | **2** | **1** |

SS = single sided
DS = double sided
DD = double density

### MFM Disk Format Table

**NOTE:** Epson (a) labels sector numbers 1 through 16. The other Epson QX10 format (a1) labels sectors from 1 to 10. Both the top and bottom of the disk are labeled the same way.

IBM (b) labels sector numbers 1 through 8. Both the top and bottom of the disk are labeled the same way.

KayPro IV and KayPro II (c and d) label sector numbers 0 through 9 on top and 10 through 19 on the bottom.

These values are taken from the MFM table. The vector at $FD46 holds the pointer to the start of the table labeled MFM$table. In the current system, these are the formats that are read and write compatible on the Commodore 128 CP/M Plus system.

The following is a listing of the MFM format table:

```
db      S256*2+(16*2-8)+1           ; 256 byte sect, 16 sect/trk
db      MFM+S256+Type0+C0+S1        ;  DSDD
dw      0                           ;start on track 2 sect 1 (2 alc)
dpb     256,32,40,2048,128,2        ; sect# 1 to 16

db      16                          ; (top and bottom numbered the same)
db      'Epson QX10'                ;1 Epson QX10

db      80h+S512*2+(10*2-8)+1       ;512 byte sect, 10 sect/trk
db      S256*2                      ;track 0 is 256 bytes/sector
db      MFM+S512+Type0+C0+S1        ;  DSDD
dw      0                           ;start on track 2 sect 1 (2 alc)
dpb     512,20,40,2048,128,2        ;sect# 1 to 10

db      10                          ;(top and bottom numbered the same)
db      'Epson QX10'                ;2

db      S512*2+(8*2-8)+1            ;512 byte sect 8 sect/trk
db      MFM+S512+Type2+C0+S1        ;  SSDD
dw      0                           ;start on track 1 sector 1 (2 alc)
dpb     512,8,40,1024,64,1          ;sect# 1 to 8

db      8                           ;
db      ' IBM-8 SS'                 ;3

db      S512*2+(8*2-8)+1            ;512 byte sect 8 sect/trk
db      MFM+S512+Type2+C0+S1        ;  DSDD
dw      0                           ; start on track 1 sector 1 (1 alc)
dpb     512,8,80,2048,64,1          ; sect# 1 to 8

db      8                           ; (top and bottom numbered the same)
db      ' IBM-8 DS'                 ;4

db      S512*2+(10*2-8)+0           ; 512 byte sector, 10 sect/trk
db      MFM+S512+Type1+C1+S0        ;  DSDD
dw      0                           ; start on track 0 sector 10 (2 alc)
dpb     512,10,80,2048,128,1        ; sect# 0 to 9 on top (even tracks)

db      10                          ; sect# 10 to 19 on bottom (odd tracks)
db      'KayPro IV'                 ;5

db      S512*2+(10*2-8)+0           ; 512 byte sect, 10 sect/trk
db      MFM+S512+Type0+C1+S0        ;  SSDD
dw      0                           ; start on track 1 sector 0 (4 alc)
dpb     512,10,40,1024,64,1         ; sect# 0 to 9

db      10                          ;
db      'KayPro II'                 ;6
```

MFM Format Table

```
db      S1024*2+(5*2-8)+1           ; 1024 byte sect, 5 sect/trk
db      MFM+S1024+Type0+C0+S1       ;   SSDD
dw      0                           ; start on track 3 sector 1 (2 alc)
dpb     1024,5,40,1024,64,3         ; sect# 1 to 5

db      5                           ;
db      'Osborne DD'                ;7
```

**MFM Format Table (continued)**

# APPENDIX K

## PART III
## THE CP/M SYSTEM MEMORY MAP

The following pages contain the Z80 CP/M memory map for the Commodore 128. The memory map includes all the key CP/M locations, vectors and variable tables. Use it as a guide through the Z80 CP/M system within the Commodore 128.

The CP/M memory map is available as a disk file "CXEQU.LIB" on the disk that comes with the computer.

```
$*MACRO

false           equ     0
true            equ     not false
banked          equ     true
EXTSYS          equ     false   ; use external system as disk and char I/O
pre$release     equ     false

; start at Jan 1,1978    78  79  80  81  82  83  84
dt$hx$yr        equ     365+365+366+365+365+365+366
;       1985                        1  2  3  4  5  6  7  8  9 10 11 12
date$hex        equ     dt$hx$yr+31+28+31+30+31+30+31+31+30+31+30+6

date    macro
        db      '6 Dec 85'
        endm
;
;
;
;
; boot memory map (bank 0 only)
;
bios02          equ     3000h   ;
block$buffer    equ     3400h   ; uses 2K
boot$parm       equ     3C00h   ; uses about 256 bytes
;
; bank 0 low memory map
;
ROM             equ     0000h
VIC$color       equ     1000h   ; I/O page only (IO$0 selected)
SYS$key$area    equ     1000h   ; 3 256 byte blocks (allow 4)
screen$40       equ     1400h   ; 2 X 80 X 25 = 4000
BANK$parm$blk   equ     2400h   ; allow 0.5K of parameters
BIOS8502        equ     2600h   ; 1.5K
VIC$screen      equ     2C00h   ; 1K
ccp$buffer      equ     3000h   ; 0c80h (allow 4K)
bank0$free      equ     4000h   ; start of free area in bank 0
;
; mapped I/O locations
;
VIC             equ     0D000h          ;
SID             equ     0D400h          ;
MMU             equ     0D500h          ;
DS8563          equ     0D600h          ; 8563
VIC$C$H         equ     0D800h          ; (memory mapped only in IO$0)
VIC$C$L         equ     01000h          ; (memory and i/o mapped in IO$0)
CIA1            equ     0DC00h          ; 6526
CIA2            equ     0DD00h          ; 6526
USART           equ     0DE00h          ; 6551 (extrn card)
RAM$dsk$base    equ     0DF00h          ; 8726

;
; Common memory allocation
;
int$block       equ     0FC00h  ; mode 2 interrupt pointers (to FDFDh)
parm$block      equ     0FD00h  ; system parameters
@buffer         equ     0FE00h  ; disk buffer (256 bytes)
;                       0FF00h  ; to 0FFFFh used by 8502
;
; the following are C128 system equates
;
enable$z80      equ     0FFD0h          ; 8502 code
return$z80      equ     0FFDCh
enable$6502     equ     0FFE0h          ; Z80 code
return$6502     equ     0FFEEh
                                        ; 1st byte used as Intterrupt pointer
vic$cmd         equ     parm$block+1    ;; bios8502 command byte
vic$drv         equ     vic$cmd+1       ;  bios8502 drive (bit 0 set, drv 0)
vic$trk         equ     vic$drv+1       ;; bios8502 track #
```

```
vic$sect        equ     vic$trk+1       ;; bios8502 sector #
vic$count       equ     vic$sect+1      ;  bios8502 sector count
vic$data        equ     vic$count+1     ;; bios8502 data byte to/from
cur$drv         equ     vic$data+1      ;  current disk installed to Vir. drive
fast            equ     cur$drv+1       ;  bit 0 set, drv 0 is fast. ect.

key$tbl         equ     fast+1          ;; pointer to keyboard table
fun$tbl         equ     key$tbl+2       ;; pointer to function table
color$tbl$ptr   equ     fun$tbl+2       ;; pointer to logical color table
fun$offset      equ     color$tbl$ptr+2 ;; function # to be prepormed
sound$1         equ     fun$offset+1    ;; unused
sound$2         equ     sound$1+2       ;;
sound$3         equ     sound$2+2       ;;

@trk            equ     sound$3+2       ;; current track number
@dma            equ     @trk+2          ;; current DMA address
;
; below here not used by ROM
;
@sect           equ     @dma+2          ; current sector number
@cnt            equ     @sect+2         ; record count for multisector transfer
@cbnk           equ     @cnt+1          ; bank for processor operations
@dbnk           equ     @cbnk+1         ; bank for DMA operations
@adrv           equ     @dbnk+1         ; currently selected disk drive
@rdrv           equ     @adrv+1         ; controller relative disk drive
ccp$count       equ     @rdrv+1         ; number of records in the CCP
stat$enable     equ     ccp$count+1     ; status line enable
                                        ;  7 kybrd, key codes(1), functions(0)
                                        ;  6 40 column tracking on(0), off(1)
                                        ;  0 disk stat, enable(1), disable(0)
emulation$adr   equ     stat$enable+1   ; address of current emulation
usart$adr       equ     emulation$adr+2 ; pointer to USART (6551) register
; CXIO equates
int$hl          equ     usart$adr+2     ; interrupt HL hold location
int$stack       equ     int$hl+2+20     ; currently only 5*2 used
user$hl$temp    equ     int$stack       ; user function HL hold location
hl$temp         equ     user$hl$temp+2  ; misc temp storage (used by VECTOR)
de$temp         equ     hl$temp+2       ; misc temp storage (used by VECTOR)
a$temp          equ     de$temp+2       ; misc temp storage (used by VECTOR)
source$bnk      equ     a$temp+1        ; inter bank move source bank #
dest$bnk        equ     source$bnk+1    ; inter bank move dest bank #
MFM$tbl$ptr     equ     dest$bnk+1      ; pointer to MFM table
; lst release end
prt$conv$1      equ     MFM$tbl$ptr+2
prt$conv$2      equ     prt$conv$1+2
key$FX$function equ     prt$conv$2+2
XxD$config      equ     key$FX$function+2
                                        ; bit 7 0 = no parity   1 = parity
                                        ; bit 6 0 = mark/space  1 = odd/even
                                        ; bit 5 0 = space/even  1 = mark/odd

                                        ; bit 1 0 = 1 stop bit  1 = 2 stop bits
                                        ; bit 0 0 = 7 data bits 1 = 8 data bits

RS232$status    equ     XxD$config+1    ; bit 7, 1=send data, 0=no data
                                        ; bit 6, 1=sending data
                                        ; bit 5, 1=recv que active
                                        ; bit 4, 1=parity error
                                        ; bit 3, 1=framing error
                                        ; bit 2, 1=recv over run (no used)
                                        ; bit 1, 1=receiving data
                                        ; bit 0, 1=Data byte ready

xmit$data       equ     RS232$status+1  ; data byte to send
recv$data       equ     xmit$data+1     ; received data byte

;
; The following equates are used by the interrupt driven keyboard handler
;
```

```
int$rate        equ     recv$data+1
;
;       1st byte is a pointer into table, 2nd to 12th byte represent
;       the keyboards current state (active low), NOTE: only
;       current if key$buffer is not full
;
key$scan$tbl    equ     int$rate+1
;
;       keyboard roll over buffer
;
key$buf$size    equ     8*2             ; must be an even number of bytes
key$get$ptr     equ     key$scan$tbl+12
key$put$ptr     equ     key$get$ptr+2
key$buffer      equ     key$put$ptr+2
;
;       software UART recv buffer
;
RxD$buf$size    equ     64
RxD$buf$count   equ     key$buffer+key$buf$size
RxD$buf$put     equ     RxD$buf$count+1
RxD$buf$get     equ     RxD$buf$put+1
RxD$buffer      equ     RxD$buf$get+1
tick$vol        equ     RxD$buffer+RxD$buf$size
;               equ     tick$vol+1

INT$vector      equ     0FDFDh          ;; contains a JMP int$handler
                                        ; (in common)
;===> 40 column misc parm
temp$1          equ     BANK$parm$blk   ;;
@off40          equ     temp$1+2        ;;
cur$offset      equ     @off40          ;;
old$offset      equ     @off40+2        ;;
prt$flg         equ     old$offset+1    ;;
flash$pos       equ     prt$flg+1       ;;
;
;===> 40 column position and color storage
paint$size      equ     flash$pos+2     ;;
char$adr$40     equ     paint$size+1    ;;
char$col$40     equ     char$adr$40+2   ;;
char$row$40     equ     char$col$40+1   ;;
attr$40         equ     char$row$40+1   ;;
bg$color$40     equ     attr$40+1       ;;
bd$color$40     equ     bg$color$40+1   ;;
rev$40          equ     bd$color$40+1   ;;
;
;===> 80 column position and color storage
char$adr        equ     rev$40+1        ;;
char$col        equ     char$adr+2      ;;
char$row        equ     char$col+1      ;;
current$atr     equ     char$row+1      ;;
bg$color$80     equ     current$atr+1   ;;
char$color$80   equ     bg$color$80+1   ;;
;       ROM uses localtions above this point
;
;===> Emulation parameters
parm$base       equ     char$color$80+1
parm$area$80    equ     parm$base+2
        ;       ds      2               ; 80 column exec$adr
        ;       ds      1               ; 80 column row #
parm$area$40    equ     parm$area$80+3
        ;       ds      2               ; 40 column exec$adr
        ;       ds      1               ; 40 column row #
buffer$80$col   equ     parm$area$40+3
;
;===> CXIO parameters
;       int$count not used by releases past 10 Oct 85
int$count       qu      buffer$80$col+40*2      ; one added every 1/60th sec
key$buf         e       int$count+1
;
```

```
;===> CXKEYS parameters
key$down$tbl    equ     key$buf+1               ; not used any more (int code)
;;;;; free space above, new interrupt driven code does not require this space
; control$keys  equ     key$down$tbl+11*2 ; byte, not used any more (int code)

commodore$mode  equ     key$down$tbl+11*2
msgptr          equ     commodore$mode+1
offset          equ     msgptr+2
cur$pos         equ     offset+1
string$index    equ     cur$pos+1
; 1st release end (3 June 85)
sys$freq        equ     string$index+1          ; -1=50Hz, 0=60Hz
; 2nd release end (1 Aug 85)
;               equ     sys$freq+1

;===> temp ROM boot data storage
blk$ptr$cnt     equ     32
load$count      equ     boot$parm       ; number of 128 byte blocks to load
ld$blk$ptr      equ     load$count+2    ; current sector dma pointer
blk$unld$ptr    equ     ld$blk$ptr+2    ; read memory block (1k,2K) pointer
block$size      equ     blk$unld$ptr+2  ; block size (1K=32 or 2K=64)
block$end       equ     block$size+1    ; allow 48K cpm.sys to load
block$ptrs      equ     block$end+2     ; end of block load buffer (+1K or +2K)
info$buffer     equ     block$ptrs+blk$ptr$cnt
                                        ; CPM3.sys load adr's and counts
ext$num         equ     info$buffer+12  ; CPM3.SYS current ext #
retry           equ     ext$num+1
boot$stack      equ     retry+1+64      ; allow 64 bytes of stack
;               equ     boot$stack

;===> special equates used by CXKEY
special         equ     00010111b
SF$exit         equ     001h            ; RETURN KEY
SF$insert       equ     028h            ; PLUS KEY
SF$delete       equ     02Bh            ; MINUS KEY
alpha$toggle    equ     03Dh            ; commodore key
alt$key         equ     050h            ; alterant key
SF$left         equ     055h            ; left arrow key
lf$arrow        equ     055h            ; left arrow key
SF$right        equ     056h            ; right arrow key
rt$arrow        equ     056h            ; right arrow key


buff$large      equ     25
buff$small      equ     7
buff$pos        equ     7

;===> External RS232 interface controls
rxd$6551        equ     USART+0         ; read
txd$6551        equ     USART+0         ; write
status$6551     equ     USART+1         ; read
reset$6551      equ     USART+1         ; write
command$6551    equ     USART+2         ; read/write
control$6551    equ     USART+3         ; read/write
txrdy           equ     10h
rxrdy           equ     08h
cmd$init        equ     0bh     ; no parity, enable txd + rxd, interrupts off
cntr$init$19200 equ     1Fh     ; 1 stop, 8 bits, 19200 baud
cntr$init$9600  equ     1Eh     ; 1 stop, 8 bits, 9600 baud (internal)
cntr$init$600   equ     017h    ; 600 baud

;===> memory management loactions
mmu$start       equ     MMU
conf$reg        equ     MMU             ; 3eh
conf$reg$1      equ     MMU+1           ; 3fh
conf$reg$2      equ     MMU+2           ; 7fh
conf$reg$3      equ     MMU+3           ; 3eh
conf$reg$4      equ     MMU+4           ; 7eh
mode$reg        equ     MMU+5           ; b1h
```

```
ram$reg         equ     MMU+6           ; 0bh 16K top Common
page$0$l        equ     MMU+7           ; 00h
page$0$h        equ     MMU+8           ; 01h
page$1$l        equ     MMU+9           ; 01h
page$1$h        equ     MMU+10          ; 01h
mmu$version     equ     MMU+11          ; ??h

enable$C64      equ     11110001b       ; FS=0
z80$off         equ     10110001b       ; value to be write to enable 8502
z80$on          equ     10110000b
fast$rd$en      equ     Z80$on+0        ; fast serial read
fast$wr$en      equ     Z80$on+8        ; fast serial write
common$4K       equ     09h             ; top  4K common
common$8K       equ     0ah             ; top  8K common
common$16K      equ     0bh             ; top 16K common

;===> preconfiguration maps
force$map       equ     0ff00h
bank$0          equ     0ff01h          ; 3fh
bank$1          equ     0ff02h          ; 7fh
io              equ     0ff03h          ; 3eh
io$0            equ     0ff03h          ; 3eh
io$1            equ     0ff04h          ; 7eh

;===> 80 column display equates
DS$index$reg    equ     DS8563
DS$status$reg   equ     DS8563
DS$data$reg     equ     DS8563+1
;===> register pointers
DS$cursor$high  equ     14
DS$cursor$low   equ     15
DS$rw$ptr$high  equ     18
DS$rw$ptr$low   equ     19
DS$rw$data      equ     31
DS$color        equ     26
;===> status bits
DS$ready        equ     80h
DS$lt$pen       equ     40h
;===> display memory layout  (16K) 0-3fffh
DS$screen       equ     0000h
DS$attribute    equ     0800h
DS$char$def     equ     2000h
;
;===> VIC equates
; vic colors
black           equ     0
white           equ     1
red             equ     2
cyan            equ     3
purple          equ     4
green           equ     5
blue            equ     6
yellow          equ     7
orange          equ     8
brown           equ     9
lt$red          equ     10
dark$grey       equ     11
med$gray        equ     12
lt$green        equ     13
lt$blue         equ     14
lt$grey         equ     15

RM$status       equ     RAM$dsk$base    ;read only register
;   bit 7       Interrupt pending if 1
;       6       Transfer complete if 1
;       5       Block verify error if 1
;   note:  bits 5-7 are cleared when read
;       4       128K if 0,  512K if 1
;       3-0     Version #
```

```
;
RM$command      equ     RAM$dsk$base+1  ;r/w
;   bit 7       execute per current config. if set
;       6       reserved
;       5       enable auto reload if set (restores all register to
;                       value before command was done, else point to
;                       next byte to read/write.)
;       4       disable FF00 decode if set (do operation after command writen)
;       3,2     reserved
;       1,0     00 = transfer C128 --> Ram Disk
;               01 = Transfer C128 <-- Ram Disk
;               10 = swap     C128 <-> Ram Disk
;               11 = Verify   C128  =  Ram Disk
;
RM$128$low      equ     RAM$dsk$base+2  ;r/w
;       bits 0 to 7 of C128 address
;
RM$128$mid      equ     RAM$dsk$base+3  ;r/w
;       bits 8 to 15 of the C128 address
;
RM$ext$low      equ     RAM$dsk$base+4  ;r/w
;       bits 0 to 7 of Ram Disk address
;
RM$ext$mid      equ     RAM$dsk$base+5  ;r/w
;       bits 8 to 15 of Ram Disk address
;
RM$ext$hi       equ     RAM$dsk$base+6  ;r/w
;       bit  16       of Ram Disk address if 128K version
;       bits 16 to 18 of Ram Disk address if 512K version
;
RM$count$low    equ     RAM$dsk$base+7  ;r/w
;       low byte transfer count (bits 0-7)
;
RM$count$hi     equ     RAM$dsk$base+8  ;r/w
;       hi byte transfer count  (bits 8-15)
;
RM$intr$mask    equ     RAM$dsk$base+9  ;r/w
;   bit 7       1=enable chip interrupts
;       6       1=enable end of block interrupts
;       5       1=enable verify error interrupts
;
RM$control      equ     RAM$dsk$base+10 ;r/w
;   bit 7,6     00      Increment both addresses  (default)
;               01      Fix expansion address
;               10      Fix C128 address
;               11      Fix both addresses
;

;===> CIA equates

Data$a          equ     00h
Data$b          equ     01h
Data$dir$a      equ     02h
Data$dir$b      equ     03h
timer$a$low     equ     04h
timer$a$high    equ     05h
timer$b$low     equ     06h
timer$b$high    equ     07h
tod$sec$60      equ     08h
tod$sec         equ     09h
tod$min         equ     0ah
tod$hrs         equ     0bh
sync$data       equ     0ch
int$ctrl        equ     0dh
cia$ctrl$a      equ     0eh
cia$ctrl$b      equ     0fh
CIA$hours       equ     CIA1+tod$hrs

key$row         equ     CIA1+Data$a     ; output
```

```
key$col         equ     CIAl+Data$b     ; input
VIC$key$row     equ     0d02fh          ; output


data$hi         equ     4               ; RS232 data line HI
data$low        equ     0               ; RS232 data line LOW

lf$shift$key    equ     80h
rt$shift$key    equ     10h
commodore$key   equ     20h
control$key     equ     04h

type$lower      equ     0
type$upper      equ     1
type$shift      equ     2
type$cntrl      equ     3
type$field      equ     00000011b

bnkl    equ     1
page0   equ     0
page1   equ     1

MMU$tbl$M       macro
        db      3fh,3fh,7fh,3eh,7eh             ; config reg's
        db      z80$on,common$8K                ; mode & mem
        db      page0,bnkl,page1,bnkl           ; page reg's
        endm
;
;       ROM functions
;
TJMP    macro   x
        rst 2 ! db x
        endm

TCALL   macro   x
        mvi l,x ! rst 4
        endm

RJMP    macro   x
        rst 3 ! db x
        endm

RCALL   macro   x
        mvi l,x ! rst 5
        endm

FR$40                   equ     2       ; offset to 40 column ROM functions

FR$wr$char              equ     00h     ; D=char auto advance
FR$cursor$pos           equ     04h     ; B=row, C=column
FR$cursor$up            equ     08h
FR$cursor$down          equ     0Ch
FR$cursor$left          equ     10h
FR$cursor$rt            equ     14h
FR$do$cr                equ     18h
FR$CEL                  equ     1Ch
FR$CES                  equ     20h
FR$char$ins             equ     24h
FR$char$del             equ     28h
FR$line$ins             equ     2Ch
FR$line$del             equ     30h
FR$color                equ     34h     ; B=color
FR$attr                 equ     38h     ; B=bit to set/clear, C=bit value
FR$rd$chr$atr           equ     3Ch     ; in  D=row, E=col
                                        ; out H=row, L=col, B=char, C=attr(real)
FR$wr$chr$atr           equ     40h     ; in  D=row, E=col, B=char, C=attr(real)
                                        ; out H=row, L=col
FR$rd$color             equ     44h
;FR$wr$color            equ     48h
;                       equ     4Ch
```

```
;
FR$trk$sect             equ     50h
FR$check$CBM            equ     52h
FR$bell                 equ     54h
;                       equ     56h
;                       equ     58h
;                       equ     5Ah
;                       equ     5Ch
;                       equ     5Eh

FR$trk$40               equ     60h
FR$set$cur$40           equ     62h
FR$line$paint           equ     64h
FR$screen$paint         equ     66h
FR$prt$msg$both         equ     68h
FR$prt$de$both          equ     6Ah
FR$update$it            equ     6Ch
;                       equ     6Eh

FR$ASCII$to$pet         equ     70h
FR$cur$adr$40           equ     72h
FR$cur$adr$80           equ     74h
FR$look$color           equ     76h
;                       equ     78h
FR$blk$fill             equ     7Ah     ; HL passed on the stack
FR$blk$move             equ     7Ch     ; '
FR$char$inst            equ     7Eh     ; '

;
;       fixed ROM locations
;
R$cmp$hl$de             equ     100h-6
R$write$memory          equ     180h+0
R$read$memory           equ     180h+3
R$set$update$adr        equ     180h+6
R$wait                  equ     180h+9

R$status$color$tbl      equ     1000h-246-16
R$color$convert$tbl     equ     1000h-230-16
;
;       Disk type byte definition
;
;       bit     7       0=GCR,  1=MFM
;
;           if bit 7 is 1 (MFM)
;               6       C0=0,   C1=1    (side 2 #, 0 to (n/2)-1 or n/2 to n-1)
;               5,4     00=128, 01=256, 10=512, 11=1024  byte/sector
;               3,2,1   disk type (MFM)
;               0       starting sector # ( 0 or 1)
;
;           if bit 7 is 0 (GCR)
;               6       unused (set to 0)
;               5,4     always 01  (256 byte sectors)
;               3,2,1   disk type (GCR)
;                       Type0 = none,  set track and sector as passed
;                       Type1 = C64  type disk
;                       Type2 = C128 type disk
;               0       unused (set to 0)

MFM     equ     1*128
C0      equ     0*64            ; 2nd side start at begining
C1      equ     1*64            ; 2nd side continues from first
C1$bit  equ     6
Type0   equ     0*2             ; (MFM) top, bottom then next track
                                ;       (TRK# 0 to (34 or 39))
Type1   equ     1*2             ; (MFM) top (trk 0 even), bottom (trk 1 odd)
                                ;       (TRK# 0 to (69 or 79))
Type2   equ     2*2             ; (MFM) top TRK# 0 to 39, bottom TRK# 40 to 79
                                ;       (TRK# on back start at 39 and go to 0)
```

```
Type7   equ     7*2             ; (MFM) pass the byte values supplied in `trk
                                ;       and `sect

TypeX   equ     7*2

S0      equ     0*1             ; start at sector 0
S1      equ     1*1             ; start at sector 1

S128    equ     0*16
S256    equ     1*16
S512    equ     2*16
S1024   equ     3*16

;
dsk$none        equ     Type0+S256      ; access to any sector on the disk
dsk$c64         equ     Type1+S256
dsk$c128        equ     Type2+S256

dir$track       equ     18              ; C64 disk dir track

;
;       6510 commands
;
vic$reset       equ     -1      ; reboot C128
vic$init        equ     0       ; initilize the bios8502
vic$rd          equ     1       ; read one sector of data (256 bytes)
vic$wr          equ     2       ; write one sector of data
vic$rdF         equ     3       ; set-up for fast read (many sectors)
vic$wrF         equ     4       ; set-up for fast write
vic$test        equ     5       ; test current disk in drive
vic$query       equ     6       ; get start sectors and #sector/trk
vic$prt         equ     7       ; print data character
vic$frmt        equ     8       ; format a disk (1541)
vic$user$fun    equ     9
vic$RM$rd       equ     10      ; RAM disk read
vic$RM$wr       equ     11      ; RAM disk write

;
;       control charactors
;
eom             equ     00h
bell            equ     07h
bs              equ     08h
lf              equ     0ah
cr              equ     0dh
xon             equ     11h
xoff            equ     13h
esc             equ     1bh
```

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 003D | ALPHATOGGLE | | | 0050 | ALTKEY | FD43 | ATEMP | 240D | ATTR40 | | |
| FF01 | BANK0 | 4000 | BANK0FREE | FF02 | BANK1 | FFFF | BANKED | | | | |
| 2400 | BANKPARMBLK | | | 240F | BDCOLOR40 | 0007 | BELL | 240E | BGCOLOR40 | | |
| 2416 | BGCOLOR30 | 3000 | BIOS02 | 2600 | BIOS8502 | 0000 | BLACK | 0020 | BLKPTRCNT | | |
| 3C04 | BLKUNLDPTR | 3400 | BLOCKBUFFER | | | 3C07 | BLOCKEND | 3C09 | BLOCKPTRS | | |
| 3C06 | BLOCKSIZE | 0006 | BLUE | 0001 | BNK1 | 3C00 | BOOTPARM | 3C77 | BOOTSTACK | | |
| 0009 | BROWN | 0008 | BS | 2420 | BUFFER80COL | | | 0019 | BUFFLARGE | | |
| 0007 | BUFFPOS | 0007 | BUFFSMALL | 0000 | C0 | 0040 | C1 | 0006 | C1BIT | | |
| 3000 | CCPBUFFER | FD21 | CCPCOUNT | 2411 | CHARADR | 2409 | CHARADR40 | 2413 | CHARCOL | | |
| 240B | CHARCOL40 | 2417 | CHARCOLOR80 | | | 2414 | CHARROW | 240C | CHARROW40 | | |
| DC00 | CIA1 | DD00 | CIA2 | 000E | CIACTRLA | 000F | CIACTRLB | DC0B | CIAHOURS | | |
| 000B | CMDINIT | 001F | CNTRINIT19200 | | | 0017 | CNTRINIT600 | | | | |
| 001E | CNTRINIT9600 | | | FD0D | COLORTBLPTR | | | | | | |
| DE02 | COMMAND6551 | | | 0020 | COMMODOREKEY | | | | | | |
| 2488 | COMMODOREMODE | | | 000B | COMMON16K | 0009 | COMMON4K | 000A | COMMON8K | | |
| D500 | CONFREG | D501 | CONFREG1 | 0502 | CONFREG2 | 0503 | CONFREG3 | D504 | CONFREG4 | | |
| DE03 | CONTROL6551 | | | 0004 | CONTROLKEY | 000D | CR | FD07 | CURDRV | | |
| 2402 | CUROFFSET | 248C | CURPOS | 2415 | CURRENTATR | 0003 | CYAN | 000B | DARKGREY | | |
| 0000 | DATAA | 0001 | DATAB | 0002 | DATADIRA | 0003 | DATADIRB | 0004 | DATAHI | | |
| 0000 | DATALOW | 0B1C | DATEHEX | FD45 | DESTBNK | FD41 | DETEMP | 0012 | DIRTRACK | | |
| D600 | DS8563 | 0800 | DSATTRIBUTE | | | 2000 | DSCHARDEF | 001A | DSCOLOR | | |
| 000E | DSCURSORHIGH | | | 000F | DSCURSORLOW | | | D601 | DSDATAREG | | |
| D600 | DSINDEXREG | 0014 | DSKC128 | 0012 | DSKC64 | 0010 | DSKNONE | 0040 | DSLTPEN | | |
| 0080 | DSREADY | 001F | DSRWDATA | 0012 | DSRWPTRHIGH | | | 0013 | DSRWPTRLOW | | |
| 0000 | DSSCREEN | D600 | DSSTATUSREG | | | 09FD | DTHXYR | | | | |
| FD23 | EMULATIONADR | | | FFE0 | ENABLE6502 | 00F1 | ENABLEC64 | FFD0 | ENABLEZ80 | | |
| 0000 | EOM | 001B | ESC | 3C35 | EXTNUM | 0000 | EXTSYS | 0000 | FALSE | | |
| FD08 | FAST | 00B0 | FASTRDEN | 00B8 | FASTWREN | 2406 | FLASHPOS | FF00 | FORCEMAP | | |
| 0002 | FR40 | 0070 | FRASCIITOPET | | | 0038 | FRATTR | 0054 | FRBELL | | |
| 007A | FRBLKFILL | 007C | FRBLKMOVE | 001C | FRCEL | 0020 | FRCES | 0028 | FRCHARDEL | | |
| 0024 | FRCHARINS | 007E | FRCHARINST | 0052 | FRCHECKCBM | 0034 | FRCOLOR | 0072 | FRCURADR40 | | |
| 0074 | FRCURADR80 | 000C | FRCURSORDOWN | | | 0010 | FRCURSORLEFT | | | | |
| 0004 | FRCURSORPOS | | | 0014 | FRCURSORRT | 0008 | FRCURSORUP | 0018 | FRDOCR | | |
| 0030 | FRLINEDEL | 002C | FRLINEINS | 0064 | FRLINEPAINT | | | | | | |
| 0076 | FRLOOKCOLOR | | | 006A | FRPRTDEBOTH | | | | | | |
| 0068 | FRPRTMSGBOTH | | | 003C | FRRDCHRATR | 0044 | FRRDCOLOR | | | | |
| 0066 | FRSCREENPAINT | | | 0062 | FRSETCUR40 | 0060 | FRTRK40 | 0050 | FRTRKSECT | | |
| 006C | FRUPDATEIT | 0000 | FRWRCHAR | 0040 | FRWRCHRATR | FD0F | FUNOFFSET | FD0B | FUNTBL | | |
| 0005 | GREEN | FD3F | HLTEMP | 3C29 | INFOBUFFER | FC00 | INTBLOCK | 2470 | INTCOUNT | | |

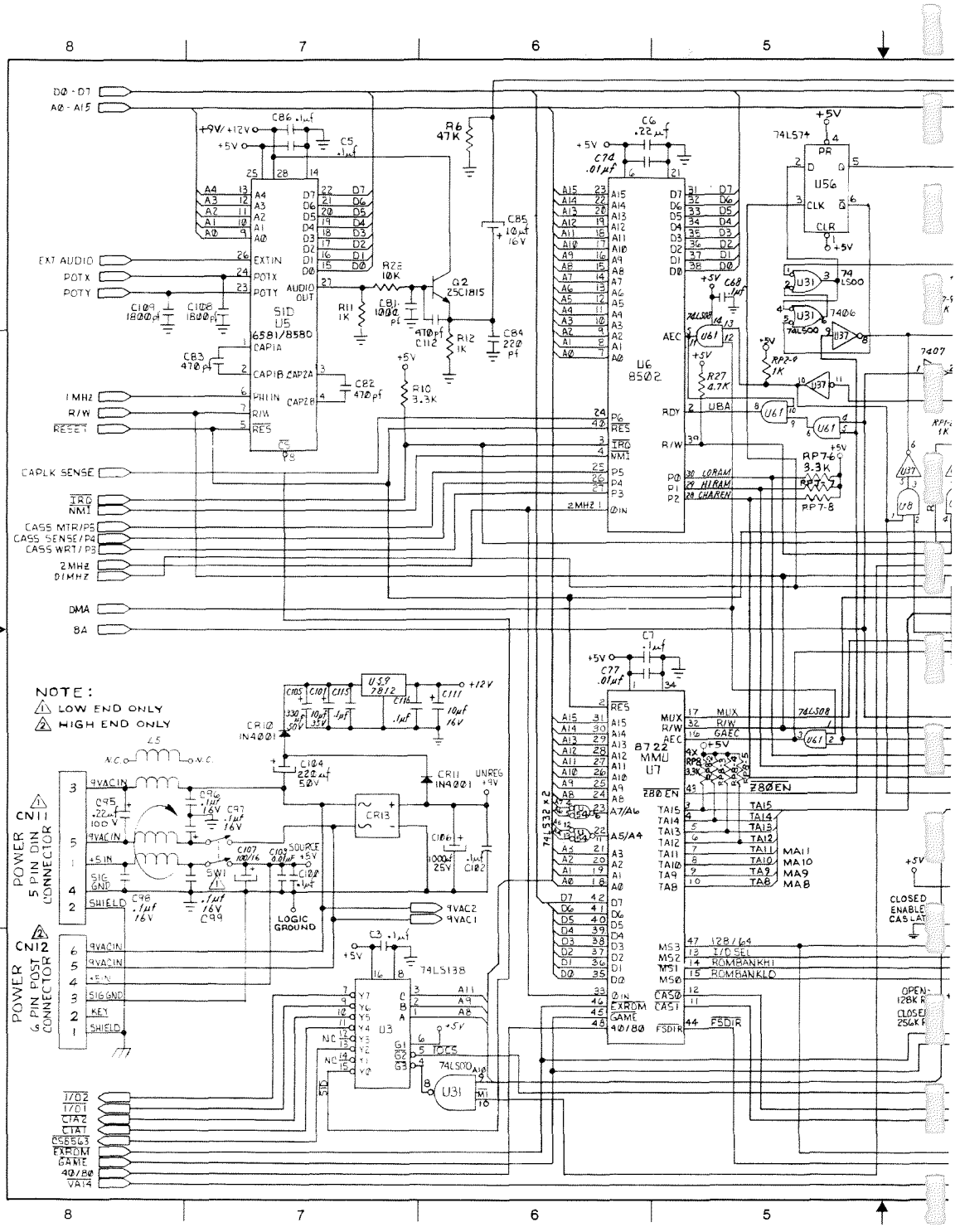
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 000D | INTCTRL | FD27 | INTHL | FD52 | INTRATE | FD3D | INTSTACK | FDFD | INTVECTOR |
| FF03 | IO | FF03 | IO0 | FF04 | IO1 | 2471 | KEYBUF | FD63 | KEYBUFFER |
| 0010 | KEYBUFSIZE | DC01 | KEYCOL | 2472 | KEYDOWNTBL | FD4C | KEYFXFUNCTION | | |
| FD5F | KEYGETPTR | FD61 | KEYPUTPTR | DC00 | KEYROW | FD53 | KEYSCANTBL | FD09 | KEYTBL |
| 3C02 | LDBLKPTR | 000A | LF | 0055 | LFARROW | 0080 | LFSHIFTKEY | 3C00 | LOADCOUNT |
| 000E | LTBLUE | 000D | LTGREEN | 000F | LTGREY | 000A | LTRED | 000C | MEDGRAY |
| 0080 | MFM | FD46 | MFMTBLPTR | D500 | MMU | D500 | MMUSTART | D50B | MMUVERSION |
| D505 | MODEREG | 2489 | MSGPTR | 248B | OFFSET | 2404 | OLDOFFSET | 0008 | ORANGE |
| 0000 | PAGE0 | D508 | PAGE0H | D507 | PAGE0L | 0001 | PAGE1 | D50A | PAGE1H |
| D509 | PAGE1L | 2408 | PAINTSIZE | 241D | PARMAREA40 | 241A | PARMAREA80 | 2418 | PARMBASE |
| FD00 | PARMBLOCK | FD48 | PRTCONV1 | FD4A | PRTCONV2 | 2405 | PRTFLG | 0004 | PURPLE |
| DF00 | RAMDSKBASE | D506 | RAMREG | 00FA | RCMPHLDE | 0F0A | RCOLORCONVERTTBL | | |
| FD51 | RECVDATA | 0002 | RED | DE01 | RESET6551 | 3C36 | RETRY | FFEE | RETURN6502 |
| FFDC | RETURNZ80 | 2410 | REV40 | DF02 | RM128LOW | DF03 | RM128MID | DF01 | RMCOMMAND |
| DF0A | RMCONTROL | DF08 | RMCOUNTHI | DF07 | RMCOUNTLOW | DF06 | RMEXTHI | DF04 | RMEXTLOW |
| DF05 | RMEXTMID | DF09 | RMINTRMASK | DF00 | RMSTATUS | 0000 | ROM | | |
| 0183 | RREADMEMORY | | | FD4F | RS232STATUS | | | | |
| 0186 | RSETUPDATEADR | | | 0EFA | RSTATUSCOLORTBL | | | 0056 | RTARROW |
| 0010 | RTSHIFTKEY | 0189 | RWAIT | 0180 | RWRITEMEMORY | | | DE00 | RXD6551 |
| FD73 | RXDBUFCOUNT | | | FD76 | RXDBUFFER | FD75 | RXDBUFGET | FD74 | RXDBUFPUT |
| 003C | RXDBUFSIZE | 0008 | RXRDY | 0000 | S0 | 0001 | S1 | 0030 | S1024 |
| 0000 | S128 | 0010 | S256 | 0020 | S512 | 1400 | SCREEN40 | 002B | SFDELETE |
| 0001 | SFEXIT | 0028 | SFINSERT | 0055 | SFLEFT | 0056 | SFRIGHT | D400 | SID |
| FD10 | SOUND1 | FD12 | SOUND2 | FD14 | SOUND3 | FD44 | SOURCEBNK | 0017 | SPECIAL |
| FD22 | STATENABLE | DE01 | STATUS6551 | 248D | STRINGINDEX | | | 000C | SYNCDATA |
| 248E | SYSFREQ | 1000 | SYSKEYAREA | 2400 | TEMP1 | 0005 | TIMERAHIGH | 0004 | TIMERALOW |
| 0007 | TIMERBHIGH | 0006 | TIMERBLOW | 000B | TODHRS | 000A | TODMIN | 0009 | TODSEC |
| 0008 | TODSEC60 | FFFF | TRUE | DE00 | TXD6551 | 0010 | TXRDY | 0000 | TYPE0 |
| 0002 | TYPE1 | 0004 | TYPE2 | 000E | TYPE7 | 0003 | TYPECNTRL | 0003 | TYPEFIELD |
| 0000 | TYPELOWER | 0002 | TYPESHIFT | 0001 | TYPEUPPER | 000E | TYPEX | DE00 | USART |
| FD25 | USARTADR | FD3D | USERHLTEMP | D000 | VIC | D800 | VICCH | 1000 | VICCL |
| FD01 | VICCMD | 1000 | VICCOLOR | FD05 | VICCOUNT | FD06 | VICDATA | FD02 | VICDRV |
| 0008 | VICFRMT | 0000 | VICINIT | D02F | VICKEYROW | 0007 | VICPRT | 0006 | VICQUERY |
| 0001 | VICRD | 0003 | VICRDF | FFFF | VICRESET | 000A | VICRMRD | 000B | VICRMWR |
| 2C00 | VICSCREEN | FD04 | VICSECT | 0005 | VICTEST | FD03 | VICTRK | 0009 | VICUSERFUN |
| 0002 | VICWR | 0004 | VICWRF | 0001 | WHITE | FD50 | XMITDATA | FD4E | XXDCONFIG |
| 0007 | YELLOW | 00B1 | Z80OFF | 00B0 | Z80ON | FD1F | @ADRV | FE00 | @BUFFER |
| FD1D | @CBNK | FD1C | @CNT | FD1E | @DBNK | FD18 | @DMA | 2402 | @OFF40 |
| FD20 | @RDRV | FD1A | @SECT | FD16 | @TRK | | | | |

**Variables**

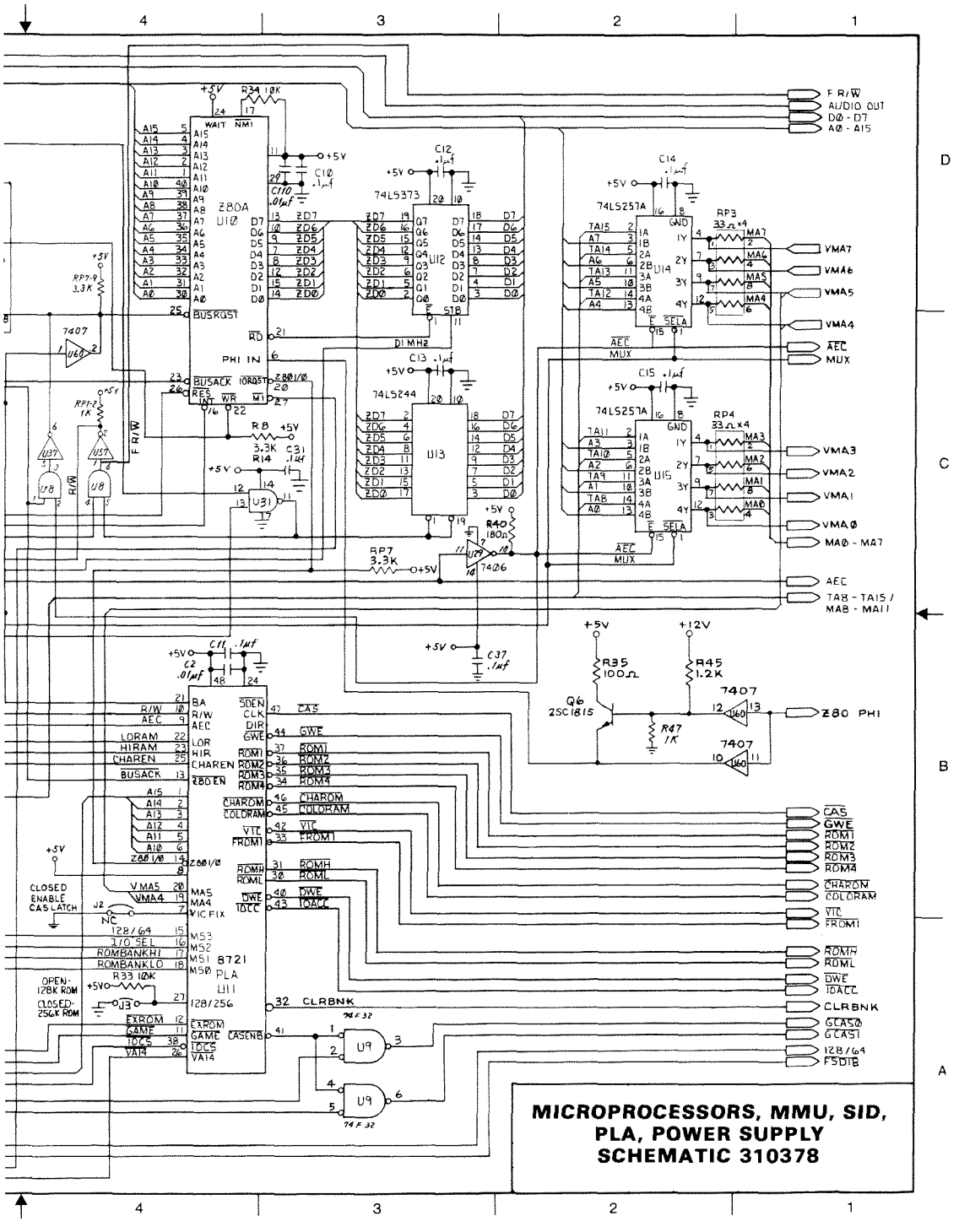
# APPENDIX L

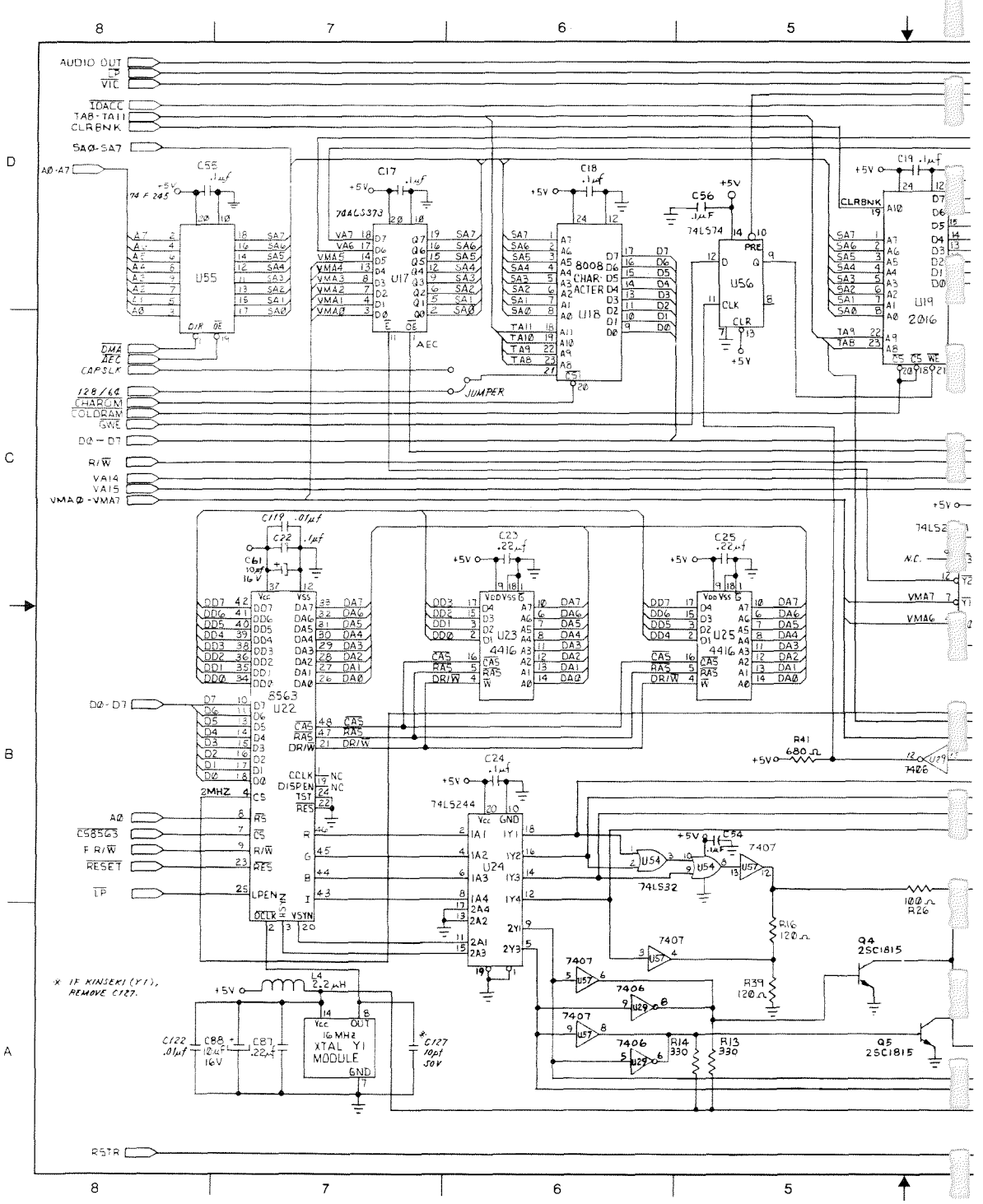
## COMMODORE 128 SYSTEM SCHEMATICS

The following eight pages contain the full system schematics for the Commodore 128. Each two-page spread represents one full-size engineering schematic sheet. For easier readability, the right edge of the left-hand page and the left edge of the right-hand page have portions of the schematic that are duplicated. This overlap is provided so you can read the circuit diagram from either half of the two-page spread, then move to the adjacent page and pick up where you left off from the point where the opposite page ends. The arrow at the top of each page provides a frame of reference to mark the portion of the diagram that is overlapped.

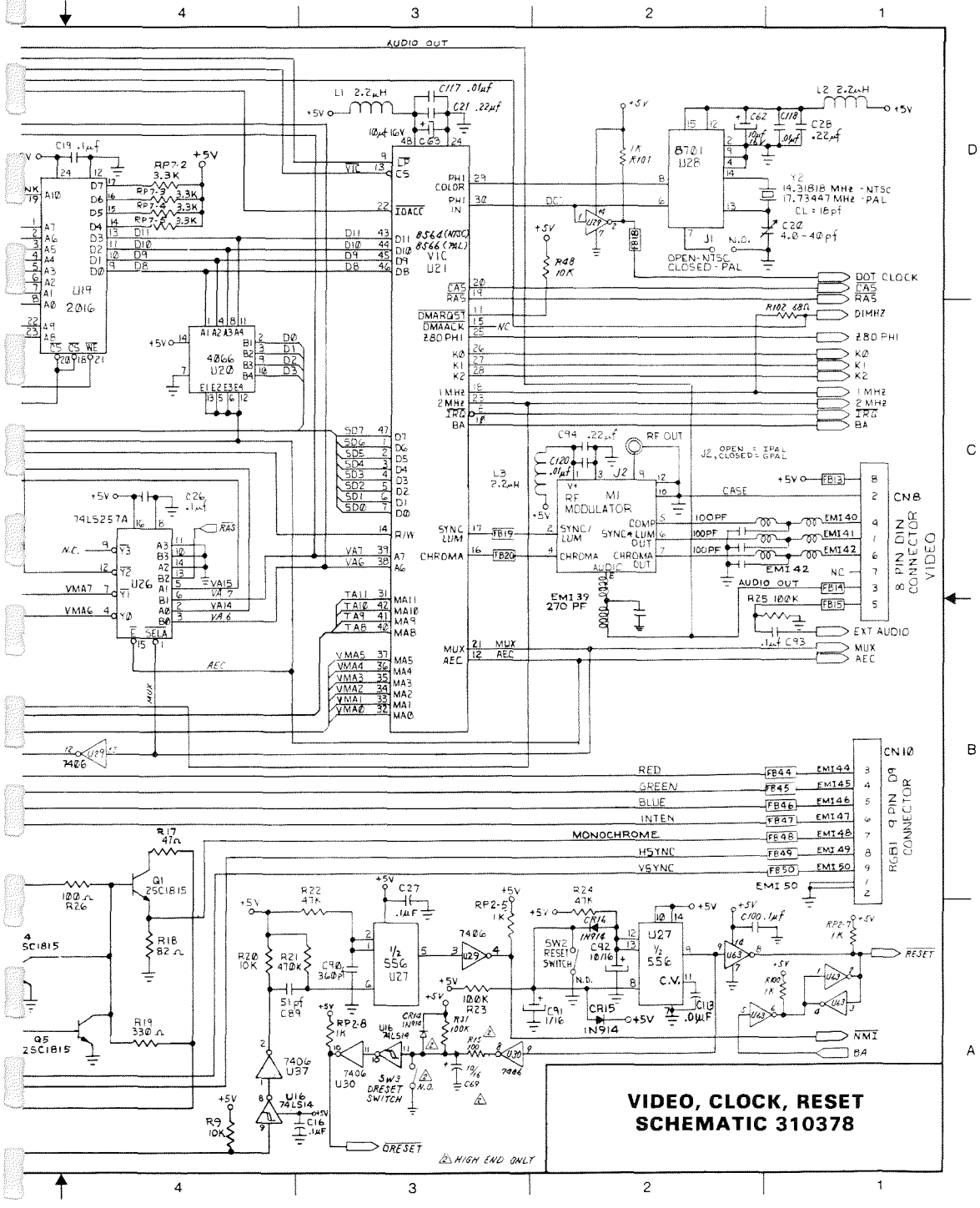


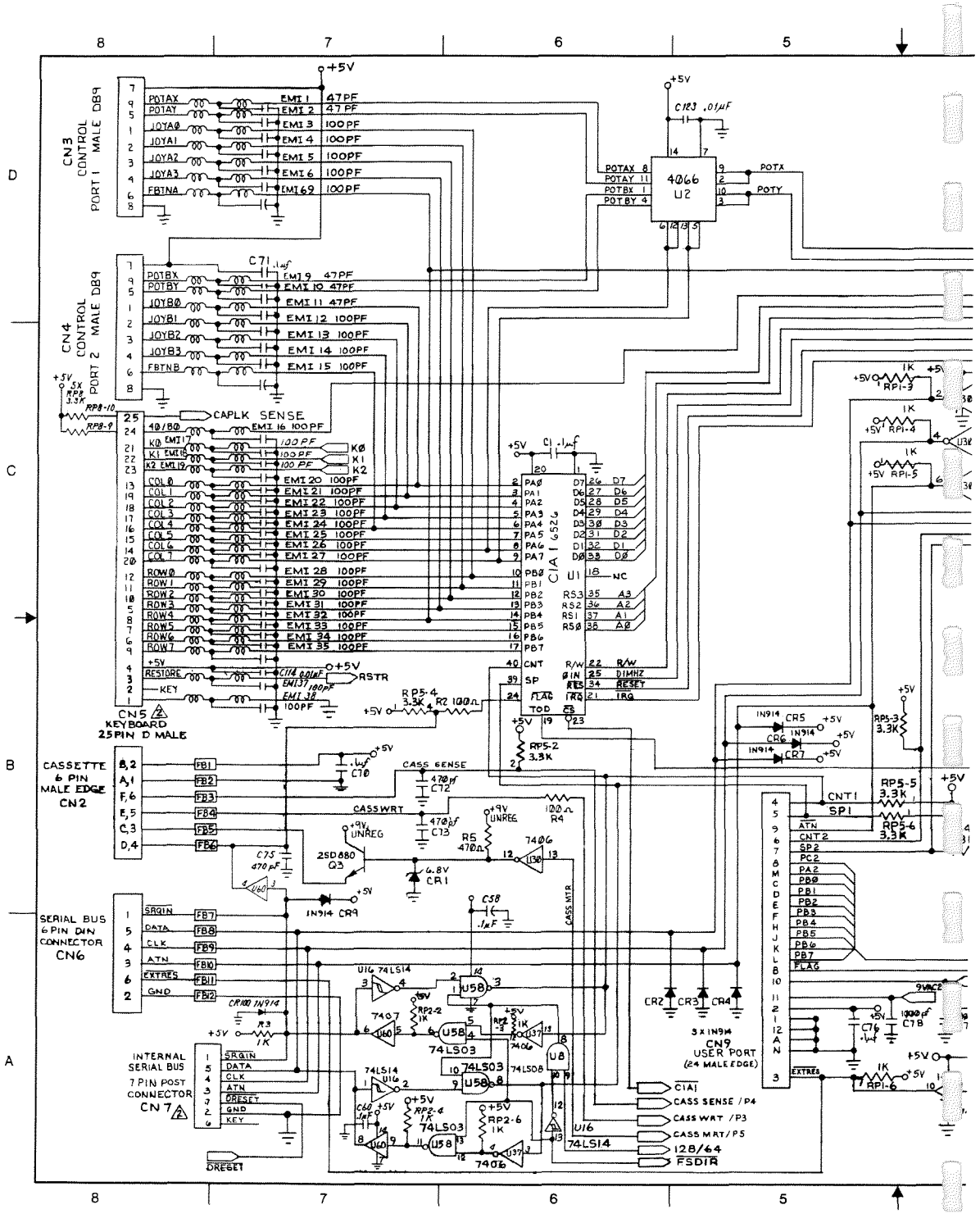
NOTE:
1 LOW END ONLY
2 HIGH END ONLY
POWER 5 PIN DIN CONNECTOR
POWER 6 PIN POST CONNECTOR
CN11
CN12
SID U5 6581/8580
U6 8502
8722 MMU U7
74LS138
74LS74
74LS00
74LS08
74LS32 x 2
U3
U56
U59 7812
Q2 2SC1815
LOGIC GROUND
CLOSED ENABLE CAS LAT
OPEN 128K R
CLOSED 256K R
D0 - D7
A0 - A15
EXT AUDIO
POTX
POTY
1MHZ
R/W
RESET
CAPLK SENSE
IRQ
NMI
CASS MTR/P5
CASS SENSE/P4
CASS WRT/P3
2MHZ
DIMHZ
DMA
8A
I/O2
I/O1
CIA2
CIA1
CS8563
EXROM
GAME
40/80
VA14


4
3
2
1
D
C
B
A
Z80A U10
8721 PLA U11
74LS373 U12
74LS244 U13
74LS257A U14
74LS257A U15
F R/W
AUDIO OUT
D0 - D7
A0 - A15
VMA7
VMA6
VMA5
VMA4
AEC
MUX
VMA3
VMA2
VMA1
VMA0
MA0 - MA7
AEC
TA8 - TA15 / MA8 - MA11
Z80 PHI
CAS
GWE
ROM1
ROM2
ROM3
ROM4
CHAROM
COLORAM
VIC
FROM1
ROMH
ROML
DWE
IOACC
CLRBNK
GCAS0
GCAS1
128/64
F5DIR
MICROPROCESSORS, MMU, SID, PLA, POWER SUPPLY SCHEMATIC 310378

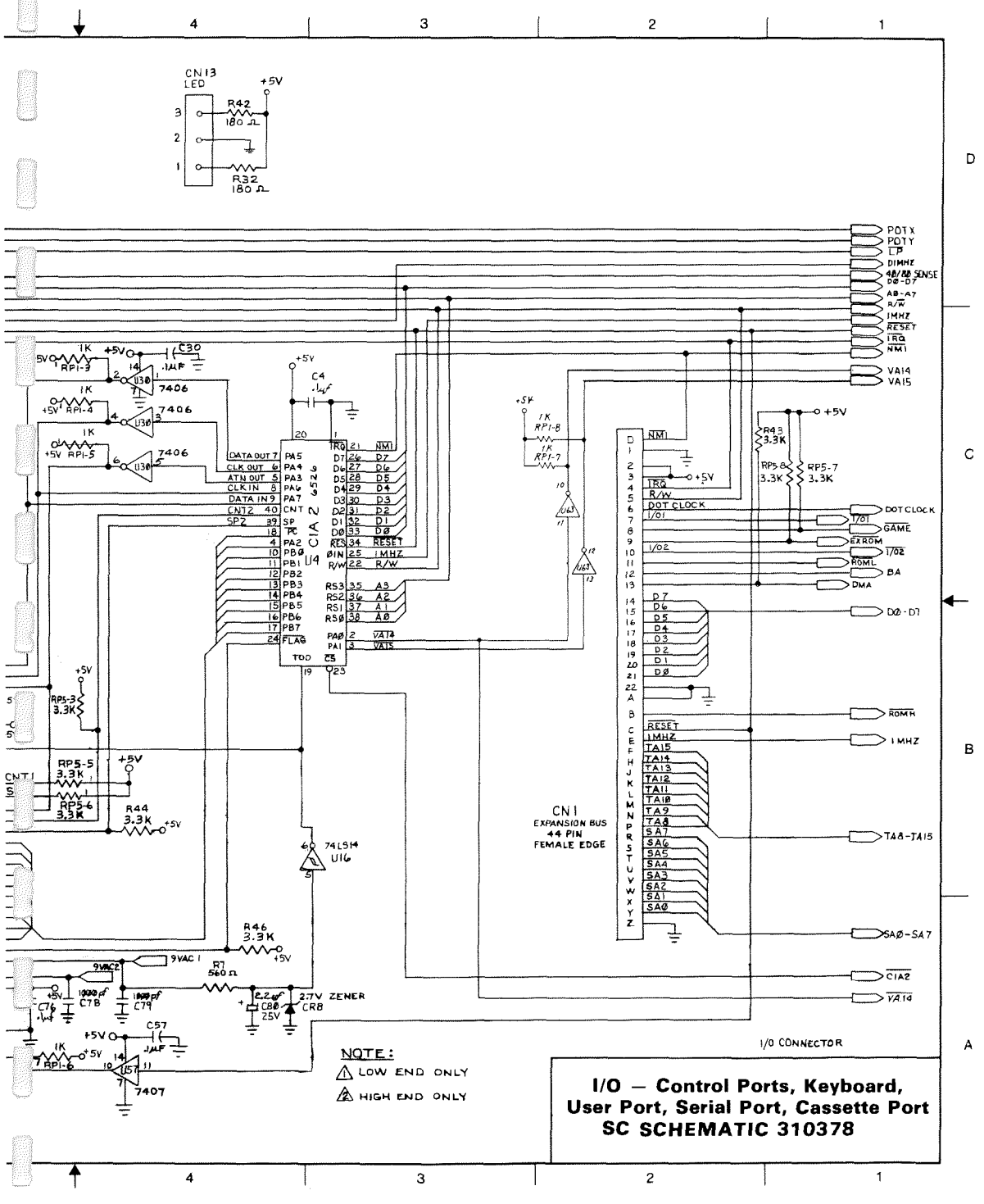


AUDIO OUT
LP
VIC
IDACC
TA8-TA11
CLRBNK
SA0-SA7
A0-A7
74 F 245
U55
74ALS373
U17
8008 CHARACTER
U18
74LS74
U56
U19
2016
DMA
AEC
CAPSLK
128/64
CHARGM
COLDRAM
GWE
D0-D7
R/W
VA14
VA15
VMA0-VMA7
JUMPER
8563
U22
4416
U23
U25
74LS244
U24
74LS32
U54
7407
7406
U29
R41 680Ω
R16 120Ω
R39 120Ω
R14 330
R13 330
R26 100Ω
Q4 2SC1815
Q5 2SC1815
L4 2.2μH
16 MHz XTAL Y1 MODULE
C122 .01μf
C88 10μf 16V
C87 .22μf
C127 10pf 50V
* IF KINSEKI (Y1), REMOVE C127.
RSTR

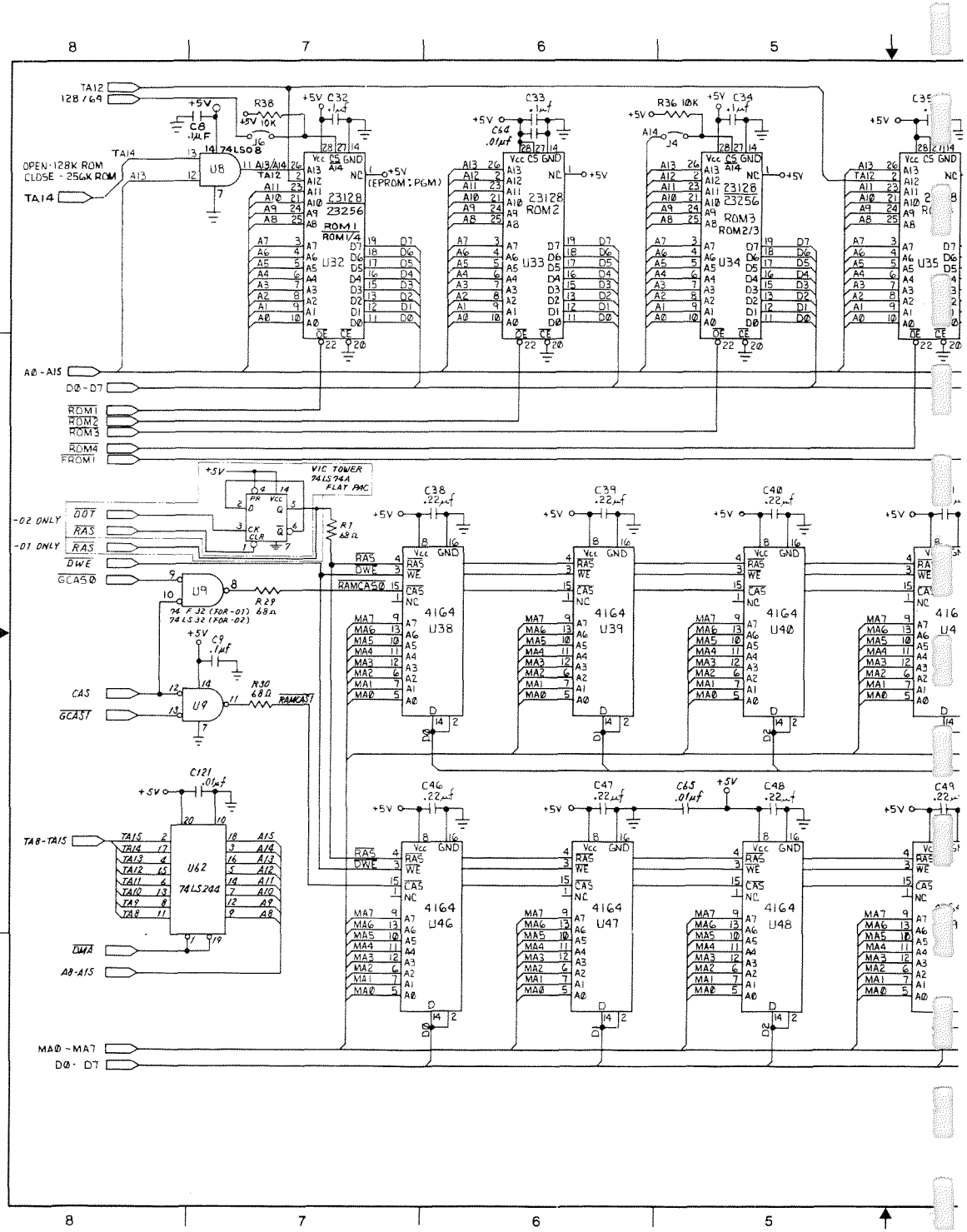

AUDIO OUT
L1 2.2μH
C117 .01μf
C21 .22μf
10μf 16V
C63
+5V
C19 .1μf
RP7-2 3.3K
RP7-3 3.3K
RP7-4 3.3K
RP7-5 3.3K
U19 2016
4066 U20
8564 (NTSC)
8566 (PAL)
VIC U21
PHI COLOR
PHI IN
IOACC
CAS
RAS
DMARQST
DMAACK
Z80 PHI
K0
K1
K2
1 MHz
2 MHz
IRQ
BA
SD7
SD6
SD5
SD4
SD3
SD2
SD1
SD0
R/W
SYNC LUM
CHROMA
MUX
AEC
74LS257A
U26
VMA7
VMA6
VA15
VA14
VA7
VA6
TA11
TA10
TA9
TA8
MA11
MA10
MA9
MA8
VMA5
VMA4
VMA3
VMA2
VMA1
VMA0
MA5
MA4
MA3
MA2
MA1
MA0
7406
U29
8701 U28
L2 2.2μH
C62
C118
C28 .22μf
Y2 14.31818 MHz - NTSC
17.73447 MHz - PAL
CL = 18pf
C20 4.0 - 40pf
J1
OPEN-NTSC
CLOSED-PAL
R48 10K
R101 1K
DOT CLOCK
DIMHZ
R102 68Ω
RF OUT
J2, OPEN = IPAL
CLOSED = GPAL
L3 2.2μH
C94 .22μf
C120 .01μf
RF MODULATOR
SYNC/LUM
CHROMA
COMP SYNC+LUM OUT
CHROMA OUT
AUDIO
EMI 39 270 PF
CASE
100PF
EMI 40
EMI 41
EMI 42
AUDIO OUT
R25 100K
EXT AUDIO
C93 .1μf
CN8
8 PIN DIN CONNECTOR VIDEO
CN10
RGBI 9 PIN D9 CONNECTOR
RED
GREEN
BLUE
INTEN
MONOCHROME
HSYNC
VSYNC
EMI 44
EMI 45
EMI 46
EMI 47
EMI 48
EMI 49
EMI 50
R17 47Ω
Q1 2SC1815
100Ω R26
R18 82Ω
R19 330Ω
Q5 2SC1815
R22 47K
R20 10K
R21 470K
C90 360pf
51pf C89
C27 .1μF
1/2 556 U27
7406 U29
RP2-5 1K
R24 47K
SW2 RESET SWITCH
CR14 1N914
C92 10/16
C91 1/16
CR15 1N914
U27 1/2 556 C.V.
C113 .01μF
C100 .1μf
RP2-7 1K
RESET
R100 1K
NMI
BA
RP2-8 1K
U16 74LS14
R31 100K
R15 100
C69
7406 U30
SW3 DRESET SWITCH
7406 U37
U16 74LS14
R9 10K
C16 .1μF
DRESET
HIGH END ONLY
VIDEO, CLOCK, RESET
SCHEMATIC 310378



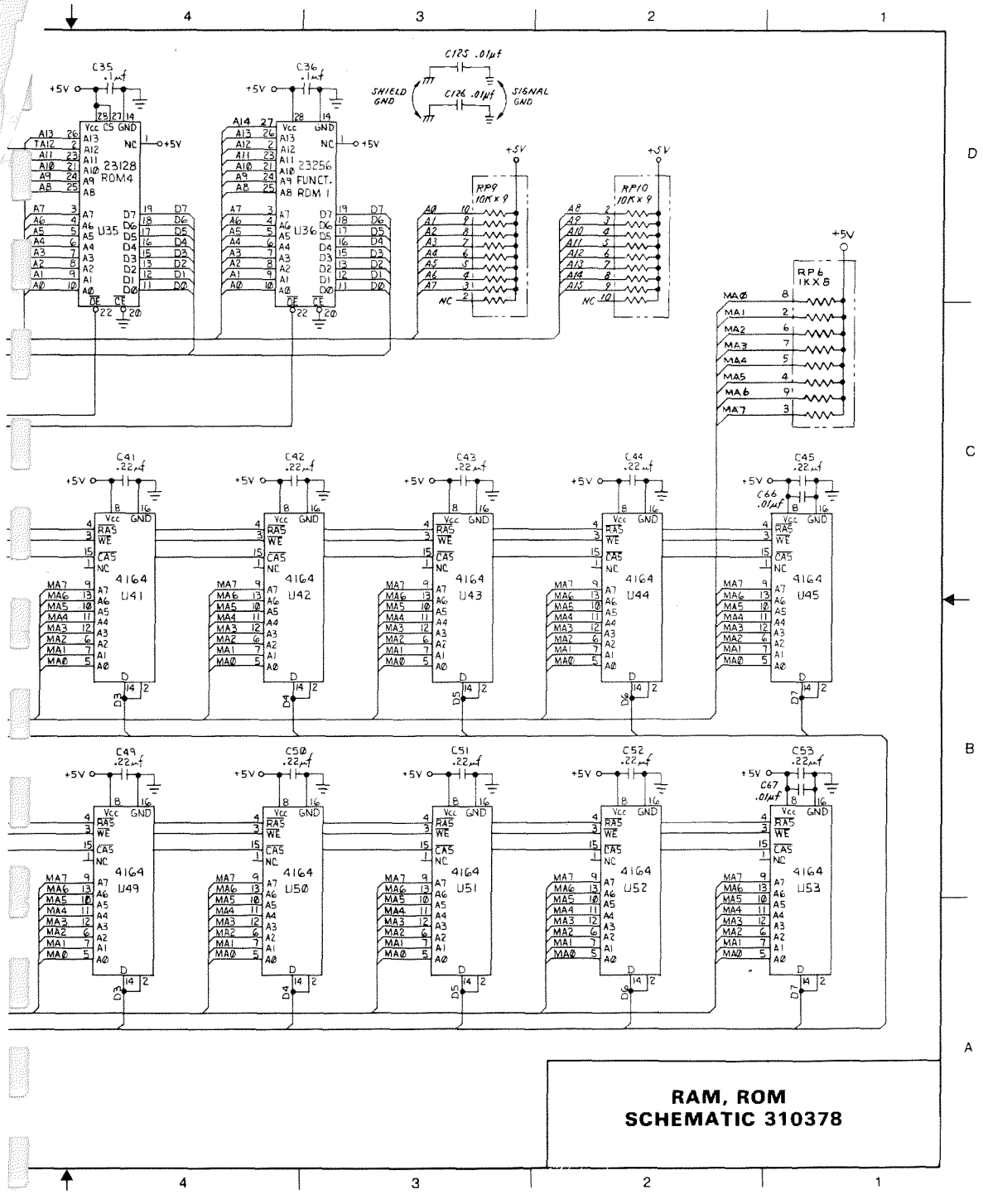
CN3 CONTROL PORT 1 MALE DB9
CN4 CONTROL PORT 2 MALE DB9
CN5 KEYBOARD 25PIN D MALE
CASSETTE 6 PIN MALE EDGE CN2
SERIAL BUS 6 PIN DIN CONNECTOR CN6
INTERNAL SERIAL BUS 7 PIN POST CONNECTOR CN7
CN9 USER PORT (24 MALE EDGE)
4066 U2
CIA 1 6526 U1
CAPLK SENSE
CASS SENSE
CASSWRT
2SD880 Q3
CASS SENSE / P4
CASS WRT / P3
CASS MRT / P5
128/64
FSDIR

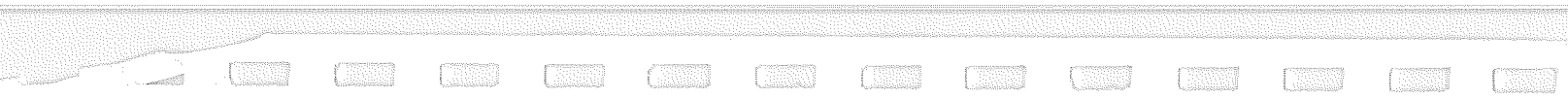



**I/O — Control Ports, Keyboard, User Port, Serial Port, Cassette Port SC SCHEMATIC 310378**



TA12
128/69
OPEN-128K ROM
CLOSE - 256K ROM
TA14
74LS08
U8
23128
23256
ROM1
ROM1/4
U32
ROM2
U33
ROM3
ROM2/3
U34
U35
(EPROM : PGM)
A0 - A15
D0 - D7
ROM1
ROM2
ROM3
ROM4
FROM1
VIC TOWER
74LS74A
FLAT PAC
-02 ONLY
-01 ONLY
DOT
RAS
DWE
GCAS0
U9
74 F 32 (FOR -01)
74 LS 32 (FOR -02)
CAS
GCAS1
RAMCAS1
4164
U38
U39
U40
U46
U47
U48
U62
74LS244
TA8-TA15
DMA
A8-A15
MA0 - MA7
D0 - D7


4
3
2
1
D
C
B
A
C35 .1μf
C36 .1μf
C125 .01μf
C126 .01μf
SHIELD GND
SIGNAL GND
+5V
23128 ROM4
U35
23256 FUNCT. ROM 1
U36
RP9 10KX9
RP10 10KX9
RP6 1KX8
C41 .22μf
C42 .22μf
C43 .22μf
C44 .22μf
C45 .22μf
C66 .01μf
4164
U41
U42
U43
U44
U45
C49 .22μf
C50 .22μf
C51 .22μf
C52 .22μf
C53 .22μf
C67 .01μf
U49
U50
U51
U52
U53
RAM, ROM
SCHEMATIC 310378



# GLOSSARY

This glossary provides brief definitions of frequently used computing terms.

**Acoustic Coupler or Acoustic Modem:** A device that converts digital signals to audible tones for transmission over telephone lines. Speed is limited to about 1200 baud, or bits per second (bps). Compare **Direct Connect Modem.**

**Address:** The label or number identifying the register or memory location where a unit of information is stored.

**Alphanumeric:** Letters, numbers and special symbols found on the keyboard, excluding graphic characters.

**ALU:** Arithmetic Logic Unit. The part of a Central Processing Unit (CPU) where mathematical operations are performed.

**Animation:** The use of computer instructions to simulate motion of an object on the screen through gradual, progressive movements.

**Array:** A data storage structure in which a series of related constants or variables is stored in consecutive memory locations. Each constant or variable contained in an array is referred to as an element. An element is accessed using a subscript. See **Subscript.**

**ASCII:** Acronym for American Standard Code for Information Interchange, which is a seven-bit code used to represent alphanumeric characters. It is a useful communication code for such things as sending information from a keyboard to the computer, and from one computer to another. See **Character String Code.**

**Assembler:** A program that translates assembly language instructions into machine language instructions.

**Assembly Language:** A machine-oriented language in which mnemonics are used to represent each machine language instruction. Each CPU has its own specific assembly language. See **CPU** and **Machine Language**.

**Assignment Statement:** A BASIC statement that sets a variable, constant or array element to a specific numeric or string value.

**Asynchronous Transmission:** A communication scheme in which data characters are sent at time intervals, independent of the system clock. Limits phone line transmission to about 2400 baud (bps). See **Synchronous Transmission.**

**Attack:** The rate at which the volume of a musical note rises from zero to peak volume.

**Background Color:** The color of the portion of the screen that the characters are placed upon.

**BASIC:** Acronym for Beginner's All-purpose Symbolic Instruction Code.

**Baud:** A unit of serial-data transmission speed. The term was originally used for measuring telegraph transmission speed. Three hundred baud is approximately equal to a transmission speed of 30 bytes, or characters, per second.

**Binary:** A base-2 number system. All numbers are represented as a sequence of 0s and 1s.

**Bit:** The abbreviation for Binary digIT. A bit is the smallest unit in a computer's memory. Each binary digit can have one of two values, 0 or 1. A bit is referred to as set or "on" if it equals 1. A bit is clear or "off" if it equals 0.

**Bit Control:** A means of transmitting serial data in which each bit has a significant meaning and a single character is surrounded with start and stop bits.

**Bit Map Mode:** An advanced graphic mode in the Commodore 128 in which you can control every pixel on the screen.

**Border Color:** The color of the edges around the screen.

**Branch:** To jump to a section of a program and execute it. GOTO and GOSUB are examples of BASIC branch instructions.

**Bubble Memory:** A relatively new type of computer memory, it uses tiny magnetic "pockets" or "bubbles" to store data.

**Burst Mode:** A special high-speed mode of communication between a 1571 disk drive and a C128 computer, in which information is transmitted at many times the speed of the 1541 Disk Drive.

**Bus:** Parallel or serial lines used to transfer signals between devices. Computers are often described by their bus structure (i.e., S-100-bus computers, etc.).

**Bus Network:** A system in which all stations or computer devices communicate by using a common distribution channel or bus.

**Byte:** A group of 8 bits that make up the smallest unit of addressable storage in a computer. Each memory location in the Commodore 128 contains 1 byte of information. One byte is the unit of storage needed to represent one character in memory. See **Bit.**

**Carrier Frequency:** A constant signal transmitted between communicating devices that is modulated to encode binary information.

**Character:** Any symbol on the computer keyboard that is printed on the screen. Characters include numbers, letters, punctuation and graphic symbols.

**Character Memory:** The area in Commodore 128's memory that stores the encoded character patterns that are displayed on the screen.

**Character Set:** A group of related characters. The Commodore 128 character sets consist of upper case letters, lower case letters and graphic characters.

**Character String Code:** The numeric value assigned to represent a Commodore 128 character in the computer's memory.

**Chip:** A miniature electronic circuit that performs a computer operation such as graphics, sound and input/output.

**Clock:** The timing circuit for a microprocessor.

**Clocking:** A technique used to synchronize the sending and receiving of data that is modulated to encode binary information.

**Coaxial Cable:** A transmission medium, usually employed in local networks.

**Collision Detection:** Determination of collision between two or more sprites or between sprites and data.

**Color Memory:** The area in the Commodore 128's memory that controls the color of each location in screen memory.

**Command:** A BASIC instruction used in direct mode to perform an action. See **Direct Mode.**

**Compiler:** A program that translates a high-level language, such as BASIC, into machine language.

**Composite Monitor:** A device used to provide the C128 40-column video display.

**Computer:** An electronic, digital device that inputs, processes, and outputs information.

**Condition:** Expression(s) between the words IF and THEN, in an IF . . . THEN statement, evaluated as either true or false. The conditional IF . . . THEN statement gives the computer the ability to make decisions.

**Coordinate:** A single point on a grid having vertical (Y) and horizontal (X) values.

**Counter:** A variable used to keep track of the number of times an event has occurred in a program.

**CPU:** Acronym for Central Processing Unit, the part of the computer containing the circuits that control and perform the execution of computer instructions.

**Crunch:** To minimize the amount of computer memory used to store a program.

**Cursor:** The flashing square that marks the current location on the screen.

**Data:** Numbers, letters or symbols that are input into the computer and are to be processed.

**Database:** A large amount of related data stored in a well-organized manner. A database management system is a program that allows access to the information.

**Data Link Layer:** A logical portion of data communications control that mainly ensures that communication between adjacent devices is error-free.

**Data Packet:** A means of transmitting serial data in an efficient package that includes an error-checking sequence.

**Data Rate or Data Transfer Rate:** The speed at which data is sent to a receiving computer—given in baud, or bits per second (bps).

**Datassette:** A device used to store programs and data files sequentially on tape.

**Debug:** To correct errors in a program.

**Decay:** The rate at which the volume of a musical note decreases from its peak value to a midrange volume called the sustain level. See **Sustain.**

**Decrement:** To decrease an index variable or counter by a specific value.

**Dedicated Line or Leased Line:** A special telephone line arrangement supplied by the telephone company, and required by certain computers or terminals, whereby the connection is always established.

**Delay Loop:** An empty FOR . . . NEXT loop that slows the execution of a program.

**Dial-Up Line:** The normal switched telephone line that can be used as a transmission medium for data communications.

**Digital:** Of or relating to the technology of computers and data communications where all information is encoded as bits of 1s or 0s that represent on or off states.

**Dimension:** The property of an array that specifies the size and direction along an axis in which the array elements are stored. For example, a two-dimensional array has an X-axis for rows and a Y-axis for columns. See **Array.**

**Direct Connect Modem:** A device that converts digital signals from a computer into electronic impulses for transmission over telephone lines. Contrast with Acoustic Coupler.

**Direct Mode:** The mode of operation that executes BASIC commands immediately after the **RETURN** key is pressed. Also called Immediate Mode. See **Command.**

**Disable:** To turn off a bit, byte or specific operation of the computer.

**Disk Drive:** A random access, mass-storage device that saves and loads files to and from a floppy diskette.

**Disk Operating System:** A program used to transfer information to and from a disk. Often referred to as a DOS.

**Duration:** The length of time a musical note is played.

**Electronic Mail, or E-Mail:** A communications service for computer users in which textual messages are sent to a central computer, or electronic "mailbox," and later retrieved by the addressee.

**Enable:** To turn on a bit, byte or specific operation of the computer.

**Envelope Generator:** Portion of the Commodore 128 that produces specific waveforms (sawtooth, triangle, pulse width and noise) for musical notes. See **Waveform**.

**EPROM:** A PROM that can be erased by the user, usually by exposing it to ultraviolet light. Abbreviation for Erasable Programmable Read Only Memory. See **PROM.**

**Error Checking or Error Detection:** Software routines that identify, and often correct, erroneous data.

**Execute:** To perform the specified instructions in a command or program statement.

**Expression:** A combination of constants, variables or array elements acted upon by logical, mathematical or relational operators that return a numeric value.

**File:** A program or collection of data treated as a unit and stored on disk or tape.

**Firmware:** Computer instructions stored in ROM, as in a game cartridge.

**Frequency:** The number of sound waves per second of a tone. The frequency corresponds to the pitch of the audible tone.

**Full-Duplex Mode:** In this mode, two computers can transmit and receive data at the same time.

**Function:** A predefined operation that returns a single value.

**Function Keys:** The four keys on the far right of the Commodore 128 keyboard. Each key can be programmed to execute a series of instructions. Since the keys can be SHIFTed, you can create eight different sets of instructions.

**GCR:** The abbreviation for Group Code Recording, a method of storing information on a disk. The 1541 and 1571 disk drives can read, write and format GCR disks.

**Graphic Characters:** Nonalphanumeric characters on the computer's keyboard.

**Graphics:** Visual screen images representing computer data in memory (i.e., characters, symbols and pictures).

**Grid:** A two-dimensional matrix divided into rows and columns. Grids are used to design sprites and programmable characters.

**Program:** A series of instructions that direct the computer to perform a specific task. Programs can be stored on diskette or cassette, reside in the computer's memory, or be listed on a printer.

**Program Line:** A statement or series of statements preceded by a line number in a program. The maximum length of a program line on the Commodore 128 is 160 characters.

**Programmable:** Capable of being processed with computer instructions.

**PROM:** The acronym for Programmable Read Only Memory. A semiconductor memory chip whose contents can be changed.

**Protocol:** The rules under which computers exchange information, including the organization of the units of data to be transferred.

**Random Access Memory (RAM):** The programmable area of the computer's memory that can be read from and written to (changed). All RAM locations are equally accessible at any time in any order. The contents of RAM are erased when the computer is turned off.

**Random Number:** A nine-digit decimal number from 0.000000001 to 0.999999999 generated by the RaNDom (RND) function.

**Read Only Memory (ROM):** The permanent portion of the computer's memory. The contents of ROM locations can be read, but not changed. The ROM in the Commodore 128 contains the BASIC language interpreter, character-image patterns and the operating system.

**Register:** Internal storage compartments with the microprocessor that communicate between system ROM, RAM, and themselves.

**Release:** The rate at which the volume of a musical note decreases from the sustain level to 0.

**Remark:** Comments used to document a program. Remarks are not executed by the computer, but are displayed in the program listing.

**Resolution:** The fineness of detail of a displayed image, determined by the density of pixels on the screen.

**RGBI Monitor:** A high-resolution display device necessary to produce the C128 80-column screen format. RGBI stands for Red/Green/Blue/Intensity.

**Ribbon Cable:** A group of attached parallel wires, usually made up of 25 lines for RS-232 communication.

**Ring Network:** A system in which all stations are linked to form a continuous loop or circle.

**RS-232:** A recommended standard for electronic and electromechanical specifications for serial communication. The Commodore 128 parallel user port can be treated as a serial port if accessed through software, sometimes with the addition of an interface device.

**Screen:** A video display unit, which can be either a television or a video monitor.

**Screen Code:** The number assigned to represent a character in screen memory. When you type a key on the keyboard, the screen code for that character is entered into screen memory automatically. You can also display a character by storing its screen code directly into screen memory with the POKE command.

**Screen Memory:** The area of the Commodore 128's memory that contains the information displayed on the video screen.

**Serial Port:** A port used for serial transmission of data; bits are transmitted one bit after the other over a single wire.

**Serial Transmission:** The sending of sequentially ordered data bits.

**Software:** Computer programs (set of instructions) stored on disk, tape or cartridge that can be loaded into random access memory. Software, in essence, tells the computer what to do.

**Sound Interface Device (SID):** The MOS 6581 sound synthesizer chip responsible for all the audio features of the Commodore 128.

**Source Code:** A nonexecutable program written in a higher-level language than machine code. A compiler or an assembler must translate the source code into an object code (machine language) that the computer can understand.

**Sprite:** A programmable, movable, high-resolution graphic image. Also called a Movable Object Block (MOB).

**Standard Character Mode:** The mode the Commodore 128 operates in when you turn it on and when you write programs.

**Start Bit:** A bit or group of bits that identifies the beginning of a data word.

**Statement:** A BASIC instruction contained in a program line.

**Stop Bit:** A bit or group of bits that identifies the end of a data word and defines the space between data words.

**String:** An alphanumeric character or series of characters surrounded by quotation marks.

**Subroutine:** An independent program segment separate from the main program that performs a specific task. Subroutines are called from the main program with the GOSUB statement and must end with a RETURN statement.

**Subscript:** A variable or constant that refers to a specific element in an array by its position within the array.

**Sustain:** The midranged volume of a musical note.

**Synchronous Transmission:** Data communications using a synchronizing, or clocking, signal between sending and receiving devices.

**Syntax:** The grammatical rules of a programming language.

**Tone:** An audible sound of specific pitch and waveform.

**Transparent:** Describes a computer operation that does not require user intervention.

**Variable:** A unit of storage representing a changing string or numeric value. Variable names can be any length, but only the first two characters are stored by the Commodore 128. The first character must be a letter.

**Video Interface Controller (VIC):** The MOS chip (8564) responsible for the 40-column graphics features of the Commodore 128.

**Voice:** A sound-producing component inside the SID chip. There are three voices within the SID chip so the Commodore 128 can produce three different sounds simultaneously. Each voice consists of a tone oscillator/waveform generator, an envelope generator and an amplitude modulator.

**Waveform:** A graphic representation of the shape of a sound wave. The waveform determines some of the physical characteristics of the sound.

**Word:** Number of bits treated as a single unit by the CPU. In an 8-bit machine, the word length is 8 bits; in a 16-bit machine, the word length is 16 bits.

# INDEX


**A**
Abbreviations, 670–673
ABS, 73
Accumulator, 127–129
  addressing, 138
  loading, 147–148
ACPTR, 422–423
ADC, 162
Addition, 19
Addressing
  absolute, 138–139, 143
  accumulator, 138
  immediate, 138
  implied, 139
  indexed, 141–143, 144–145
  indirect, 143–145
  modes, 137, 141–142
  relative, 140–141
  16-bit, 133–135
  table, 161–179
  zero-age, 139, 142
ALT Mode, 497
AND, 152, 162
APPEND, 27
Arithmetic
  instructions, 151–152
  operations, 18–20
Arrays, 13, 16–18
ASC, 73, 660–662
ASL, 163
Assembler, 126–127
ATN, 73
AUTO, 27

**B**
BACKUP 27–28
BANK, 28
BASIC
  advanced programming techniques, 103–107
  color RAM in C128, 218
  color RAM in C64, 218–219
  C128 bit map mode, 221
  crunching of programs, 95
  C64 bit map mode, 222
  C64 character modes, 222
  entering machine language subroutines through, 198–202
  error messages, 644–647
  intelligent use of, 97
  mixed with machine language, 198–205
  placement of machine language routines with, 203–205
  relocating, 106
  screen memory in C128, 215–217
  screen memory in C64, 217
BASIN, 423–433
BCC, 163
BCS, 163
BEGIN/BEND, 28–29
BEQ, 164
BIOS (Basic Input Output System), 486–489, 500, 677–683, 704–705
BIT, 153–154, 164
Bit map mode 112, 221, 222
  data, 241–243
  80-column (8563) chip, 314–320
  multi-color, 243–245
  standard, 239–243
  standard sprites, 283–284
  video matrix, 240–241, 244
Bits
  masking, 97–98
  16-bit addressing, 133–135
  values in a byte, 98–99
BLOAD, 29
BMI, 164
BNE, 164
BOOT, 29–30, 446–447
BOX, 30, 113–114
BPL, 165
BRK, 165
BSAVE, 31
BSOUT, 433
Buffer
  control block, 685
  routine, 93
BUMP, 73–74
Bus
  architecture, 560–562
  color data, 562
  display, 562
  expansion, 635–637
  loading, 567–568
  multiplexed address, 561
  processor, 560
  serial, 633–634
  shared address, 561–562
  translated address, 560–561
BVC, 165
BVS, 165

**C**
Cassette connector, 398
CATALOG, 31
CHAR, 31–32, 115
Character mode
  accessing character ROM, 229
  character memory, 226–229, 234–235
  color data, 225–226, 235–237
  color memory, 226
  C128 BASIC, 219–220
  C64 BASIC, 222
  multi-color, 233–237
  programmable characters, 230–233
  screen location, 224, 234
  screen memory data, 224–225
  standard, 223–233
CHKIN, 429–430
CHR$, 74, 660–662
CIA (6526) chip, 611–623
  control registers 622–623
  description, 611, 618–622
  electrical characteristics, 613–615
  interface signals, 615–616
  interrupt control, 621–622
  serial port, 620–621
  timing, 616–617, 618–620
CINT, 410, 414
CIOUT, 423
CIRCLE, 32–33, 115–116
CKOUT 430–431
CLALL, 439
CLC, 166
CLD, 166
CLI, 166
CLOSE, 33, 428–429
CLOSE ALL, 443
CLR, 33
CLRCH, 431–432
CLV, 166
CMD, 33
CMP, 167
CMPSTA, 456
COLLECT, 34
COLLISION, 34, 267
COLOR, 34–35, 116–117
Color mode
  extended background, 237–239
  memory map, 664
  sprites, 283–285
  *See also* Memory, color RAM
Commands, 12
  basic, 27–72
  CP/M, 481–482, 483
  format, 25–27
  graphics, 113–122
  machine language monitor, 186–194
  sprites, 267–270
  summary, 674–675
  *See also* specific commands
Commodore 128. *See* C128 Mode
Commodore 64. *See* C64 Mode
Complex Interface Adapter. *See* CIA (6526) chip
CONCAT, 35–36
C128 Mode, 2–3, 5
  BASIC bit map mode, 221

C128 Mode *(continued)*
character memory, 219–220
character set availability, 222
color RAM in BASIC, 218
CP/M disk format, 493–494
memory map, 502–540
ROM cartridge, 471–472
screen memory in BASIC, 215–217
switching from mode to mode, 6
using C64 function key values, 95
Configuration Register. *See* Memory
Connectors, 652–657
Constants, 12–15
floating-point, 13
integer, 13
string, 14–15
CONT, 36
Control codes, 666–668
COPY, 36
COS, 74
CP/M, 676
BIOS routines, 677–683, 704–705
calling user function, 702–704
commands, 481–482, 483
control characters for line editing, 482, 484
copies of disks and files, 482, 485
disk organization, 491–495
enhancements, 479
files, 479–481, 482, 485
keyboard scanning, 496–497
memory map, 709–720
mode, 4
requirements for system, 478
switching from mode to mode, 7
system layout, 486–487
system memory organization, 489–491
system operations, 500
system release, 8
CPX, 167
CPY, 167
C64 Mode, 3, 5, 444
BASIC bit map mode, 222
BASIC character modes, 222
color RAM in BASIC, 218–219
CP/M disk format, 492–493
input/output assignments, 546–554
memory map, 540–554
ROM cartridge, 472
screen memory in BASIC, 217
switching from mode to mode, 7
using function key values, 95
Cursor, 313–314, 326

**D**

Daisy wheel printer, 378
DATA, 36
Datassette, 389–390
Data structures, 684–685
DCLEAR, 37
DCLOSE, 37
Debugging. *See* Programming
DEC, 168
DEF FN, 37
DELETE, 38
Device numbers, 457
DEX, 168
DEY, 168
DIM, 38
DIRECTORY, 38–39
Directory, 375
Disk drive
copies, 482, 485
device number, 378
directory, 375
formatting, 372–373
replacing files or programs, 374
retrieving files or programs, 375–376
saving programs, 373–374
verifying files or programs, 374–375
Disk Parameter Block, 685
Division, 19–20
DLCHR, 450
DLOAD, 39
DMA CALL, 444–445
DO/LOOP/WHILE/UNTIL/EXIT, 39–40
DOPEN, 40
DOS errors, 101, 648–651
Dot matrix printer, 378–379
DRAW, 41, 117–118
Drive Table, 684–685
DSAVE, 41
DVERIFY, 42

**E**

Editor. *See* Screen editor
80-column (8563) chip, 292–334
bit map mode, 314–320
Block Write and Block Copy, 312–313, 333
characters, 296–297, 301–304, 325, 328, 333
cursor, 313–314, 326
display, 299–301, 326, 327–334
frames, 297–299
RAM, 304–305, 309–313, 327–334
registers, 304–309, 324–334
scrolling of screen, 320–323, 328–331
8502 microprocessor, 569–574
description, 569
electrical specification, 569–571
processor timing, 571–574
END, 42
ENVELOPE, 42, 336–337, 347–348
Environmental specifications, 568
EOR, 152, 168
Errors
BASIC messages, 644–647
DOS, 101, 648–651
functions, 101
logic, 99
syntax, 99
tracing of, 101
trapping of, 100
*See also* Programming, debugging
Escape codes, 669
Exponentiation, 20
Expressions, 18
arithmetic, 18
string, 24

**F**

FAST, 43
FETCH, 43
Files
CP/M, 479–481
creating and storing, 376–378
disk drive, 374–376
merging, 106–107
FILTER, 43, 337–338, 348–351
FNxx, 74
FOR/TO/STEP/NEXT, 44
FRE, 75
Function keys
changing, 95
programming, 94
using C64 values, 95
Functions, 72–86
errors, 101
user, 685–704
*See also* specific functions

**G**

GET, 44–45
GET#, 45
GETCFG, 452
GETIN, 438–439
GETKEY, 45
GO64, 45
GOSUB, 45–46
GOTO/GO TO, 46
GRAPHIC, 46, 119
Graphics
commands, 113–122
power behind, 208–263
programming, 110–122
system, 215–223
GSHAPE, 46, 119–120

**H**

Hardware
components, 4–5
specifications, 556–641
system architecture, 557–558
*See also* specific components
HEADER, 47
HELP, 47
HEX$, 75
Hexadecimal notation, 136–137

**I**

IF/THEN/ELSE, 47–48
INC, 169
INDFET, 454–455
INDSTA, 455
INPUT, 48
INPUT#, 49
Input/output, 5, 372–400, 727
BIOS (Basic Input Output System), 486–489, 500, 667–683, 704–705
controller ports input, 390–393
C64 assignments, 546–554
Datassette output, 389–390
disk drive, 372–376
files, 376–378
modem output, 381
output control, 393–394
pinouts, 394–400
printer output, 378–381
RS-232 channel, 382–388

Input/output *(continued)*
screen outpuut, 388–389
INSTR, 75
Instructions
arithmetic, 151–152
branching, 154–156
compare, 150–151
counter, 148–149
entering machine language in monitor, 183–184
jump, 159–160
logical, 151, 152–153
machine language, 145–179, 183–184
multiple, 96
register to memory, 147–148
register transfer, 156
return, 160
rotate, 156, 157
set and clear, 158–159
shift, 156–157
stack, 160
table, 161–179
*See also* specific instructions
INT, 75–76
Interrupt service routine, 258–263
INX, 169
INY, 169
IOBASE, 442
IOINIT, 409–410, 415–416
IRQ pin, 411–414

**J**
JMP, 169
JMPFAR, 453–454
JOY, 76
Joysticks, 390–392
JSR, 170
JSRFAR, 453–454

**K**
Kernal calls, 414–457
Kernal/Editor flags, 539–540
Kernal jump table, 537–539
Kernal routines, 403–406
KEY, 49, 421–422
Keyboard, 640–642, 727
connector pinout, 640–641
scanning, 496–497, 588–589
Keywords. *See* Reserved system words

**L**
LDA, 147–148, 170
LDX, 170
LDY, 171
LEFT$, 76
LEN, 77
LET, 49–50
Light pen, 393, 593
LIST, 50
LISTN, 425
LKULPA, 448–449
LKUPSA, 448–449
LOAD, 50, 434–435
Loading
accumulator, 147–148
bus, 567–568
routine, 93–94
LOCATE, 51, 120
LOG, 77
Logic
errors, 99
instructions, 151, 152–153
Logical operators, 21–22
LSR, 171

**M**
Machine language, 124–179
character memory, 223
color RAM, 219
definition, 124
entering programs, 182–195
entering subroutines through BASIC, 198–202
executing programs, 184–186
instructions, 145–179, 183–184
mixed with BASIC, 198–205
monitor, 127
monitor commands, 186–194
operand field, 126
operation code field, 125
placement of programs in memory, 202–203
placement of routines with BASIC, 203–205
programming of SID chip, 352–358
screen memory, 217–218
Z80, 702–708
MEMBOT, 420–421
Memory, 4
banked, 208–213, 218, 490
character (ROM), 219–222, 226–229
color RAM, 218–219, 225–226, 236, 238, 243, 245
Configuration Register, 460–463
CP/M system memory organization, 489–491
crunching, 95
dynamic RAM, 624–626
80-column (8563) chip, 299–301, 304–305, 309–313, 327–334
management, 5, 458–471, 583–587
maps, 502–554, 663–664, 709–720
Mode Configuration Register, 465–466
placement of machine language programs, 202–203
preconfiguration, 462–465
RAM Configuration Register, 467–469
RAM and 80-column (8563) chip, 304–305, 309–313 327–334
RAM organization, 566–567
RAM and system architecture, 557–558, 729
ROM, 627–632
ROM banking, 627
ROM cartridge startup, 471–472
ROM chip, 630, 632
ROM organization, 564
ROM pinout, 629, 631
ROM and system architecture, 557–558, 729
ROM timing, 628
RS-232 channel, 387–388
screen (RAM), 215–218
16K video banks, 210–212
64K RAM banks, 208–210
split-screen mode, 246–247
storage, 12–13, 148
switching banks, 459–460
system organization, 562–567
*See also* Character mode
Memory Management Unit (MMU). *See* Memory, management
MEMTOP, 419–420
Menu, 92
MFM. *See* Modified Frequency Modulation
MID$, 77
Modem, 381
Modified Frequency Modulation (MFM), 705–708
Module communication, 486–487
MONITOR, 51
Monitor, 388–389
entering machine language programs in, 183–184
field descriptors, 187–188
machine language commands, 186–194
manipulating text within, 194–195
Mouse, 392
Movable object blocks. *See* Sprites
MOVSPR, 51–52, 267–268
Multiplication, 19
Music, 336–369
coding a song from sheet music, 341–344
equal-tempered scale values, 366, 607–608
instruments, 336–337
notes, 338–340, 341–343, 366–369
statements, 336–341

**N**
NEW, 52
Non-Maskable Interrupt (NMI) vector, 407–408
NOP, 160, 171

**O**
ON, 53
OPEN, 53–54, 427–428
Operating system, 402–475
CP/M components, 486
Kernal calls, 414–457
Kernal routines for programs, 403–406
vectors, 407–414
Operations
arithmetic, 18–20
hierarchy of, 22–23
string, 24
ORA, 152, 172
Output. *See* Input/output

**P**
Paddles, 392
Page pointers, 470–471
PAINT, 54–55, 120–121
PEEK, 77
PEN, 78

Performance specifications, 567–568
PFKEY, 450–451
PHA, 172
PHOENIX, 448
PHP, 172
Pinouts, 394–400
PLA, 172
  *See also* Programmed Logic Array
PLAY, 55–56, 338–339
PLOT, 441
PLP, 173
POINTER, 78
POKE, 56
Ports, 372, 727
  controller, 390–393, 398–399
  expansion, 399–400
  for peripheral equipment, 652–657
  serial, 394
  user (RS-232 channel), 394–396
  *See also* specific controller devices
POS, 79
POT, 79
PRIMM, 456–457
PRINT, 56, 100
PRINT#, 57, 380
Printer, 378–380
  control, 380–381
  daisy wheel, 378
  dot matrix, 378–379
PRINT USING, 57–58
Processors, 4, 560
Program counter, 132
Programmed Logic Array, 581–582, 723
Programming
  advanced BASIC techniques, 103–107
  debugging, 99–101
  of 80-column (8563) chip, 292–334
  escape, 106
  of function keys, 94
  graphics, 110–122
  of SID chip in machine language, 352–358
Programs
  definition, 376
  Kernal routines, 403–406
  printing program listing, 379
  printing through a program, 380
  replacing, 374
  retrieving from disks, 375–376
  saving, 373–374
  verifying, 374–375
PUDEF, 58–59

**R**

RAM. *See* Memory
RAMTAS, 416
Raster interrupt split screen program, 248–258
RCLR, 79–80
RDOT, 80
RDTIM, 437
READ, 59
READSS, 426
RECORD, 59–60
Registers, 126
  CIA (6526) chip, 622–623
  80-column (8563) chip, 304–309, 324–334
  8502 microprocessor, 127
  8563 video controller, 595
  interrupt, 591–592
  raster, 591
  shadow, 213–214, 539–540
  SID chip, 359–365, 528–530
  status, 130–132
  System Version, 471
  VIC chip, 524–527, 591–592
  X and Y index, 129–130, 141–143
Relational operators, 20–21
REM, 60, 96
RENAME, 60
RENUMBER, 60–61
Reserved system symbols, 88–89
Reserved system words, 86–88
RESET, 408
RESTOR, 416–417
RESTORE, 61, 407–408
RESUME, 61–62
RETURN, 62
RGBI video connector, 397–398
RGR, 80–81
RIGHT$, 81
RND, 81
ROL, 173
ROM. *See* Memory
ROR, 173
RSPCOLOR, 82
RSPPOS, 82
RSPRITE, 83
RS-232 channel, 382–388, 394–396
  closing, 385–387
  data, 385
  memory locations, 387–388
  opening, 382–385
  sample program, 387
RTI, 174
RTS, 174
RUN, 62
RWINDOW, 83

**S**

SAVE, 63, 435–436
SBC, 174
SCALE, 63, 121–122
Schematics, 721–729
SCNCLR, 64
SCRATCH, 64
Screen editor
  control codes, 474
  escape codes, 473
  intermediate storage, 213–214
  interrupt-driven, 214–215, 247–248
  jump table, 474–475
Screen output, 388–389
  *See also* Graphics; Video
Scrolling
  of 8563 screen, 320–323, 328–331
  of 8564 VIC chip, 593
SCRORG, 440
SEC, 175
SECND, 418–419
SED, 175
SEI, 175
SETBNK, 451
SETLFS, 426–427
SETMSG, 418
SETNAM, 427
SETTIM, 436–437
SETTMO, 422
SGN, 84
SID (Sound Interface Device) chip, 336, 723
  audio input, 351–352
  electrical characteristics, 605–606
  envelope generators, 608–610
  filter, 337–338, 348–351, 363
  pins, 600, 602–604
  programming in machine language, 352–358
  registers, 359–365, 528–530
  specifications, 599–604
  synchronization and ring modulation, 358–359
  and system architecture, 558
  timing, 606–607
SIN, 84
SLEEP, 64, 99
SLOW, 64
SOUND, 65, 339–340
Sound, 5, 336–369
  characteristics, 345–348
  statements, 336–341
  volume, 347
Sound Interface Device. *See* SID chip
Space elimination, 95
SPC, 84
SPINP, 442–443
Split-screen mode, 245–248
  organization in memory, 246–247
  raster interrupt program, 248–258
SPOUT, 442–443
SPRCOLOR, 65–66, 268–269
SPRDEF, 66, 269–270, 276, 279
SPRITE, 66–67, 272–273
Sprites, 266–290
  adjoining, 274–276
  collision priorities, 289–290
  color, 283–285
  commands, 267–270
  creation of image, 279–281
  creation procedure in definition mode, 270–274
  display priorities, 288–289
  enablement, 282
  expansion of size, 287–288
  inner workings, 279–290
  pointers, 281–282
  positioning on screen, 285–287
  program examples, 276–278
SPRSAV, 67, 273
SQR, 84
SSHAPE/GSHAPE, 68–69, 122, 273–274
STA, 175
Stack pointer, 132–133
STASH, 69
Statements, 12
  basic, 27–72
  format, 25–27
  *See also* specific statements
STOP, 69, 100, 437–438
Storage. *See* Files; Memory, storage

**Half-Duplex Mode:** In this mode, data can be transmitted in only one direction at a time; if one device is sending, the other must simply receive data until it's time for it to transmit.

**Hardware:** Physical components in a computer system, such as the keyboard, disk drives and printer.

**Hexadecimal:** Refers to the base-16 number system. Machine language programs are often written in hexadecimal notation.

**Home:** The upper-left corner of the screen.

**IC:** The abbreviation for Integrated Circuit. A silicon chip containing an electrical circuit made up of components such as transistors, diodes, resistors and capacitors. Integrated circuits are smaller, faster and more efficient than the individual circuits used in older computers.

**Increment:** To increase an index variable or counter with a specified value.

**Index:** The variable counter within a programming loop.

**Input:** Data fed into the computer to be processed. Input sources include the keyboard, disk drive, Datassette or modem.

**Integer:** A whole number (i.e., a number containing no fractional part), such as 0, 1, 2, etc.

**Interface:** The point of meeting between a computer and an external entity, whether an operator, a peripheral device or a communications medium. An interface may be physical, involving a connector, or logical, involving software.

**I/O:** The abbreviation for Input/Output. Refers to the process of entering data into the computer, or transferring data from the computer to a disk drive, printer or storage medium.

**Keyboard:** Input component of a computer system.

**Kilobyte (K):** 1024 bytes.

**Local Network:** One of several short-distance data communications schemes typified by common use of a transmission medium by many devices at high-data speeds. Also called a Local Area Network, or LAN.

**Loop:** A program segment executed repetitively a specified number of times.

**Machine Language:** The lowest-level language the computer understands. The computer converts all high-level languages, such as BASIC, into machine language before executing any statements. Machine language is written in binary form, which a computer can execute directly. Also called machine code or object code.

**Matrix:** A two-dimensional rectangle with row and column values.

**Memory:** Storage locations inside the computer. ROM and RAM are two different types of memory.

**Memory Location:** A specific storage address in the computer. There are 131,072 memory locations (0–131,071) in the Commodore 128.

**MFM:** The abbreviation for Modified Frequency Modulation, a method of storing information on disks. The 1571 disk drives can read and write to MFM disks.

**Microprocessor:** A CPU that is contained on a single integrated circuit (IC). Microprocessors used in Commodore personal computers include the 6510, the 8502 and the Z80.

**Mode:** A state of operation.

**Modem:** The acronym for MOdulator/DEModulator. A device that transforms digital signals from the computer into analog electrical impulses for transmission over telephone lines, and does the reverse for reception.

**Monitor:** A display device resembling a television set but with a higher-resolution (sharper) image on the video screen.

**Motherboard:** In a bus-oriented system, the board that contains the bus lines and edge connectors to accommodate the other boards in the system.

**Multi-Color Bit Map Mode:** A graphic mode that allows you to display one of four colors for each pixel within an 8 by 8 character grid. See **Pixel.**

**Multi-Color Character Mode:** A graphic mode that allows you to display four different colors within an 8 by 8 character grid.

**Multiple-Access Network:** A flexible system by which every station can have access to the network at all times; provisions are made for times when two computers decide to transmit at the same time.

**Null String:** An empty character (""). A character that is not yet assigned a character string code. Produces an illegal quantity error if used in a GET statement.

**Octave:** One full series of eight notes on the musical scale.

**Operating System:** A built-in program that controls everything a computer does.

**Operator:** A symbol that tells the computer to perform a mathematical, logical or relational operation on the specified variables, constants or array elements in the expression. The mathematical operators are +, –, *, / and ↑. The relational operators are <, =, >, < =, > = and <>. The logical operators are AND, OR NOT and XOR.

**Order of Operations:** Sequence in which computations are performed in a mathematical expression. Also called Hierarchy of Operations.

**Parallel Port:** A port used for transmission of data 1 byte at a time using 8 data lines, one for each bit.

**Parity Bit:** A 1 or 0 added to a group of bits that identifies the sum of the bits as odd or even, for error checking purposes.

**Peripheral:** Any accessory device attached to the computer such as a disk drive, printer, modem or joystick.

**Pitch:** The highness or lowness of a tone that is determined by the frequency of the sound wave. See **Frequency.**

**Pixel:** Computer term for picture element. Each dot that makes up an image on the screen is called a pixel. Each character on the screen is displaced within an 8 by 8 grid of pixels. The entire screen is composed of a 320 by 200 pixel grid. In bit-map mode, each pixel corresponds to a bit in the computer's memory.

**Pointer:** A register used to indicate the address of a location in memory.

**Polling:** A communications control method used by some computer/terminal systems whereby a "master" station asks many devices attached to a common transmission medium, in turn, whether they have information to send.

**Port:** A channel through which data is transferred to and from the CPU.

**Printer:** Peripheral device that outputs the contents of the computer's memory onto a sheet of paper. This paper is referred to as a hard copy.