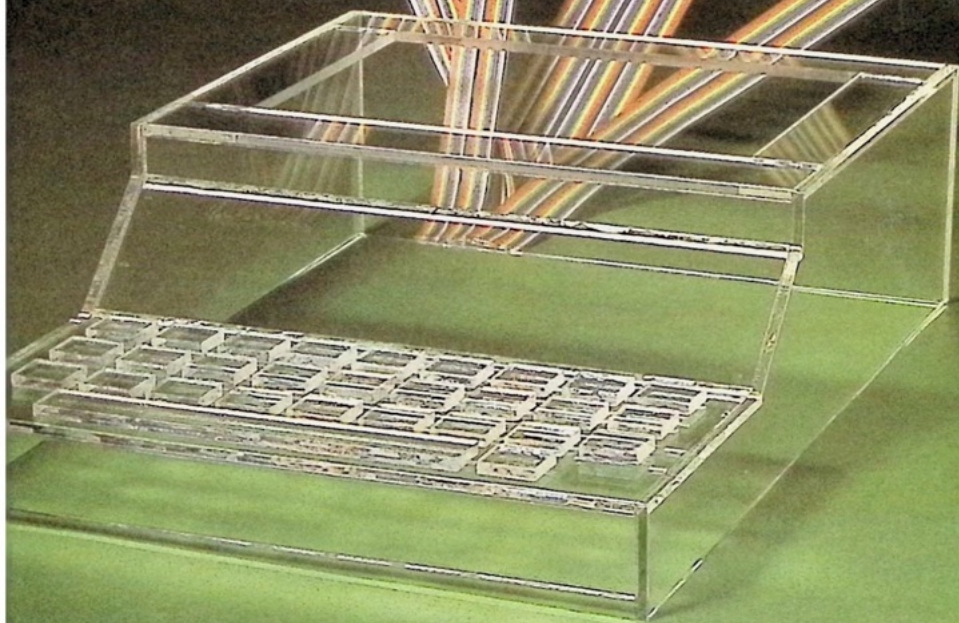




21893

TRS-80® Color Computer Interfacing, With Experiments

Andrew C. Staugaard, Jr.



BLACKSBURG

CONTINUING EDUCATION SERIES™

edited by Larsen, Titus & Titus

The Blacksburg Continuing Education™ Series

The Blacksburg Continuing Education Series™ of books provide a Laboratory—or experiment-oriented approach to electronic topics. Present and forthcoming titles in this series include:

- Advanced 6502 Interfacing
- Analog Instrumentation Fundamentals
- Apple Assembly Language
- Apple Interfacing
- Basic Business Software
- BASIC Programmer's Notebook
- Circuit Design Programs for the Apple II
- Circuit Design Programs for the TRS-80
- Computer Assisted Home Energy Management
- Design of Active Filters, With Experiments
- Design of Op-Amp Circuits, With Experiments
- Design of Phase-Locked Loop Circuits, With Experiments
- Design of Transistor Circuits, With Experiments
- 8080/8085 Software Design (2 Volumes)
- 8085A Cookbook
- Electronic Music Circuits
- 555 Timer Applications Sourcebook, With Experiments
- Guide to CMOS Basics, Circuits, & Experiments
- How to Program and Interface the 6800
- Introduction to Electronic Speech Synthesis
- Introduction to FORTH
- Microcomputer—Analog Converter Software and Hardware Interfacing
- Microcomputer Data-Base Management
- Microcomputer Design and Maintenance
- Microcomputer Interfacing With the 8255 PPI Chip
- NCR Basic Electronics Course, With Experiments
- NCR EDP Concepts Course
- PET Interfacing
- Programming and Interfacing the 6502, With Experiments
- Real Time Control With the TRS-80
- 16-Bit Microprocessors
- 6502 Software Design
- 6801, 68701, and 6803 Microcomputer Programming and Interfacing
- The 68000: Principles and Programming
- 6809 Microcomputer Programming & Interfacing, With Experiments
- STD Bus Interfacing
- TEA: An 8080/8085 Co-Resident Editor/Assembler
- TRS-80 Assembly Language Made Simple
- TRS-80 Color Computer Interfacing
- TRS-80 Interfacing (2 Volumes)
- TRS-80 More Than BASIC

In most cases, these books provide both text material and experiments, which permit one to demonstrate and explore the concepts that are covered in the book. These books remain among the very few that provide step-by-step instructions concerning how to learn basic electronic concepts, wire actual circuits, test microcomputer interfaces, and program computers based on popular microprocessor chips. We have found that the books are very useful to the electronic novice who desires to join the "electronics revolution," with minimum time and effort.

Jonathan A. Titus, Christopher A. Titus, and David G. Larsen
"The Blacksburg Group"

TRS-80® Color Computer Interfacing, with Experiments

By Andrew C. Staugaard, Jr.

Howard W. Sams & Co., Inc.
4300 WEST 62ND ST. INDIANAPOLIS, INDIANA 46268 USA

Copyright © 1983 by Andrew C. Staugaard, Jr.

FIRST EDITION
FIRST PRINTING—1983

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-672-21893-3
Library of Congress Catalog Card Number: 82-60873

Edited by: C.W. Moody
Illustrated by: T.R. Emrick

Printed in the United States of America.

Preface

So, you have purchased, or are considering the purchase of, a Radio Shack TRS-80® Color Computer. If you have a TRS-80® Color Computer, you have most likely explored many of its software features and have found it to be a very capable, flexible, and enjoyable personal computer. Now, you are itching to use your computer to sense and control external events and are wondering how the TRS-80® Color Computer can be interfaced, or connected, to the "outside" world. However, you have probably never interfaced anything to a computer before and feel that this is a task which requires the expertise of an electronics engineer or computer technician—not so! As you will soon discover, your TRS-80® Color Computer is a relatively simple computer system. Once you learn some fundamental hardware and software interfacing concepts, you will find it easy to interface your computer to the outside world.

The material in this book assumes that you have very little background in the field of microcomputer interfacing. All that's required is a working knowledge of the Color BASIC programming language and a fundamental understanding of the binary number system. The Radio Shack book, *Getting Started with Color BASIC*, will provide you with the software background that is required. A review of the binary number system is provided in Appendix A of this book, if a review is needed.

We will begin in Chapter 1 by taking a functional look at the circuit and expansion capabilities of the TRS-80® Color Computer. Then, in Chapter 2, we will acquaint you with the 6809E microprocessor, or CPU. You do not have to have a background in microprocessors to

understand this chapter. Our intent will be to acquaint you with enough of the 6809E features to allow you to interface to your computer. In Chapter 3, you will learn the fundamental interfacing concepts of decoding, latching, and 3-state buffering. Using these concepts, you will learn how to construct input/output, or I/O, ports. You will then be acquainted with a programmable interface device called a PIA. You will use the PIA in Chapter 4 to develop a simple parallel I/O interface for the TRS-80® Color Computer expansion connector. The PIA interface will be used to transfer data between the computer and external devices such as switches, LEDs, and analog converters.

The world outside of the computer is an analog world. Consequently, to sense and control external events with a computer, conversions must be made between analog and digital signals. This is the topic of Chapter 5. You will learn how to use the internal analog conversion features of your computer, as well as how to connect external converters to the PIA interface developed in Chapter 4. Finally, the experiments in Chapter 6 have been designed to demonstrate the material in the first five chapters. These experiments can be performed using any TRS-80® Color Computer system (4K, 16K, or 32K).

I wish to acknowledge Jonathan A. Titus, Christopher A. Titus, and David G. Larsen for their permission to use material from *TRS-80® Interfacing, Book 2* in the preparation of Chapter 5 of this book. Enjoy!

ANDREW C. STAUGAARD, JR.

To a new member of our family, Andrew III. May he grow up in a world made better by technology. And to Scout, an old member of the family who is no longer with us. He remained faithfully by my side during the preparation of four previous manuscripts, and most of this one.

Contents

1 A LOOK INSIDE THE TRS-80®	
COLOR COMPUTER	9
Objectives—The TRS-80® Color Computer Circuit—	
The TRS-80® Color Computer Expansion Ports—	
Summary	
2 THE 6809E MICROPROCESSOR, OR CPU	27
Objectives—General Concepts and Evolution—The	
Internal Structure and Instruction Set of the 6809—	
The 6809E Interface Lines—I/O Instructions and	
Timing—Summary	
3 MICROCOMPUTER INTERFACING	
FUNDAMENTALS	47
Objectives—General I/O Concepts—Decoding—The	
6821 PIA—Summary	
4 EXPANSION CONNECTOR INTERFACING	69
Objectives—Expansion Connector Signals—The PIA	
Interface—I/O Device Control—Summary	

5 ANALOG CONVERSION	89
Objectives—Digital-to-Analog Converters—	
Analog-to-Digital Converters—A List of Representa-	
tive DAC and ADC Manufacturers	
6 EXPERIMENTS	129
Objectives—The Expansion Connector Breadboard—	
Experiment 1: Data Input and the PIA Interface—	
Experiment 2: Data Output—Experiment 3: Internal	
Digital-to-Analog Conversion—Experiment 4: DAC	
Interfacing—Experiment 5: Internal Analog-to-Digital	
Conversion—Experiment 6: ADC Interfacing	
APPENDIXES	171
A NUMBER SYSTEMS AND	
COMPUTER ARITHMETIC	171
Number Systems—Digital Computer Arithmetic—	
Negative Numbers and Twos Complement—Review	
Questions and Answers	
B DIGITAL REVIEW	185
Basic Logic Gates—Flip-Flops	
C DATA SHEETS	193
MC6809E Microprocessor—MC6821 Peripheral Inter-	
face Adapter (PIA)—SE/NE5018 D/A Converter—	
ADC0804 A/D Converter	
INDEX	199

1 A Look Inside the TRS-80® Color Computer

Before you can begin to interface electronic devices, controllers, actuators, etc., to any computer system, you must have a fundamental understanding of the internal hardware structure, or circuitry, and expansion capabilities of the system. This does not mean that you need to be an electronics engineer or technician to interface to the TRS-80® Color Computer. As you are about to discover, the Color Computer circuit is relatively simple, since many of the circuit functions are performed by large-scale integrated (LSI) devices. Once you understand the basic functions of these devices and some fundamental hardware/software interfacing concepts, you will find it easy to interface to your computer.

In this chapter, we will provide you with a "look" inside the TRS-80® Color Computer. We will acquaint you with the functions of its major LSI components and expansion ports in preparation for discussions that follow in later chapters.

OBJECTIVES

At the end of this chapter, you should be able to:

- State the functional regions of the TRS-80® Color Computer circuit.
- Explain how RAM and ROM are used by the Color Computer.
- Explain what is mean by *memory mapped I/O*.
- Describe the expansion port capabilities of the Color Computer.

THE TRS-80® COLOR COMPUTER CIRCUIT

A photograph of the TRS-80® Color Computer circuit board is provided in Fig. 1-1. Note that all of the system components, including the power supply, are on one circuit board. In fact, when you first see the inside of the computer, you may be a little disappointed by the lack of components. You may wonder how they squeeze all of that computing power out of one circuit board containing only 24 integrated circuits. The answer is found in the capabilities of the 6809E microprocessor and its related memory and peripheral devices. Before we go any further, we must provide you with a note of caution: *You will void your warranty by opening your TRS-80® Color Computer case.* One of the screws that needs to be removed to open the case lies under a paper label that gives this warning. However, there is no need for you to open the case, since we will show you the inside of the computer. In addition, all of the experiments in this book interface to the external connectors provided on the back and right side of the case.

A block diagram of the Color Computer circuit with numbered references is provided in Fig. 1-2. Aside from the power supply, the circuit can be functionally divided into four major sections, as indicated

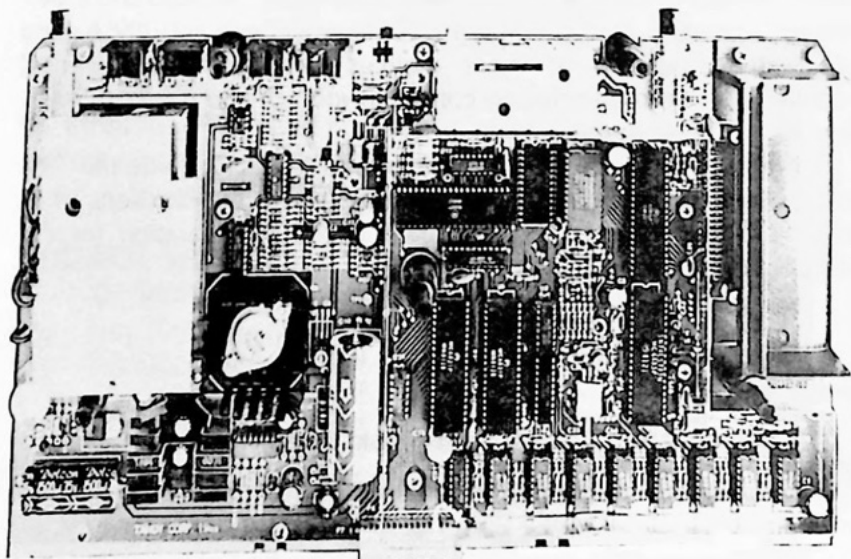


Figure 1-1 _____
The Color Computer circuit board.

by the dark outlines in Fig. 1-2—the *microprocessor*, *memory* and *memory control*, *video display*, and *input/output* (or *I/O*). At the heart of each section is one or more LSI device. Let us take a brief look at each major section in preparation for more detailed discussions later.

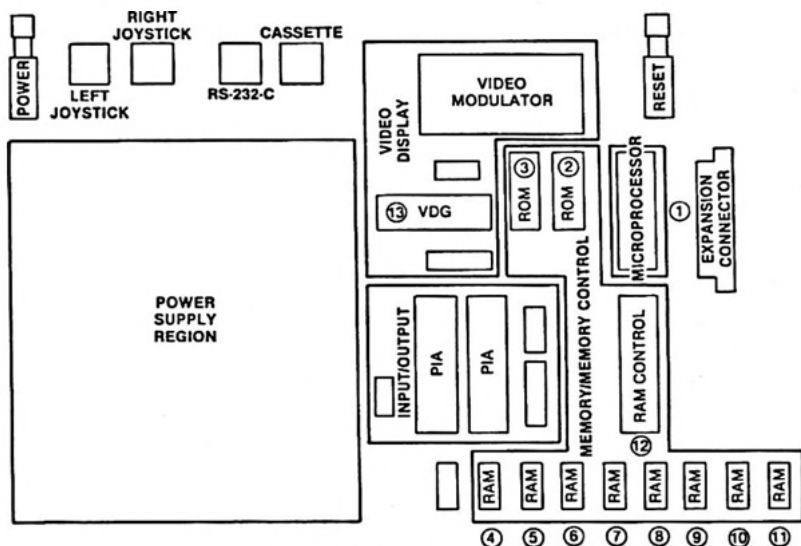


Figure 1-2
Functional block diagram of the Color Computer circuit.

The Microprocessor, or CPU

The brain of the entire system is the Motorola 6809E microprocessor, or central processing unit (CPU). (Refer to item ① in Fig. 1-2.) The 6809E is an 8-bit microprocessor, meaning that data is transferred between the 6809E and its memory and peripheral devices eight bits at a time.

It might be helpful at this point to go to Appendix A and review number systems and computer arithmetic. We will try to stay with decimal numbers as much as possible. However, since the microprocessor is a binary device, our discussions will involve binary numbers and arithmetic from time to time. Of course, you old-hand computer "nuts" can just keep on reading.

The 6809E is a *high performance* microprocessor that was designed specifically for the computer systems market. High-level computer languages, such as BASIC, Pascal, FORTRAN, and COBOL, can be

handled very efficiently by the 6809E, as compared to other 8-bit microprocessors. In addition, the TRS-80® Color Computer derives much of its color graphics capability from the power of the 6809E. We will defer further discussion of the 6809E until the next chapter, which is devoted entirely to the 6809E.

A microprocessor cannot do much by itself. In order to create a workable *microcomputer* system, you must add memory and peripheral input/output, or I/O, to the microprocessor. Memory is used to store programs and program data. Peripheral I/O allows you to communicate with the microprocessor. The TRS-80® Color Computer memory and peripheral I/O capabilities are the topics of our next discussion.

Memory and Memory Control

There are two general types of memory—*read/write memory*, commonly called *RAM*, and *read-only memory*, or *ROM*. Any computer system must include both types. In general, ROM is preprogrammed by the manufacturer and cannot be altered by the user (you). Programs in ROM are *nonvolatile*, meaning that they will not disappear when power is removed from the system.

The TRS-80® Color Computer uses ROM in several different ways. The first way the Color Computer uses ROM is to supply the *Color BASIC* interpreter. The Color BASIC interpreter is a program stored in ROM that interprets the BASIC language symbols into 6809 code.

The 6809 microprocessor does not operate directly on the BASIC language commands that you are familiar with from the Color Computer BASIC manual(s). Instead, the 6809 has its own set of instructions. A program written using the "native" instructions for a given microprocessor, like the 6809, is called an *assembly language program*. A 6809 assembly language program written to perform a given operation has very little resemblance to a BASIC command or program written to perform the same operation. For example, the 6809 does not have a PRINT command. Consequently, it cannot directly perform the following operation:

PRINT "TRS-80 COLOR COMPUTER"

The BASIC interpreter stored in ROM determines that a print operation is required and then executes a series of 6809 assembly language instructions that cause the message to be displayed. The assembly language instructions cause the necessary internal and external 6809 operations to take place to transfer the message to the display.

We will not require you to learn the 6809 assembly language instructions in this book. However, you should be aware that it is the native language of the TRS-80® Color Computer. It is possible to program the Color Computer directly in the assembly language code. Programs written in assembly language will execute up to 1000 times faster than similar programs written in BASIC because no interpretation process is required. In addition, assembly language programs usually require less memory space and allow you to perform operations that are not possible with BASIC. If you want to program your Color Computer in assembly language code, you can obtain an editor/assembler program from Radio Shack. To learn how to program in 6809 assembly language, we recommend the book: *6809 Microcomputer Programming and Interfacing, with Experiments*, published by Howard W. Sams & Co., Inc.

If your system uses standard Color BASIC, you were provided one ROM device that contains the Color BASIC interpreter. (Refer to item ② in Fig. 1-2.) If your system uses *Extended Color BASIC*, a second ROM device was added to provide the additional Extended Color BASIC commands. (Refer to item ③ in Fig. 1-2.) Each ROM is an 8K device, meaning that it contains 8192 bytes of program information ($K = 1024$).

The second way the Color Computer uses ROM is through the ROM cartridges, or Program Paks™, that are inserted in the side of the computer. Each Color Computer ROM cartridge contains two ROM devices that have been programmed for a specific function. Examples include game, personal finance, and word processing programs, among others. Providing such programs in ROM rather than on cassette tape or disk is a big improvement over the original TRS-80, Model I. Programs in ROM never lose their quality and are very durable. In addition, the ROM program is ready to run almost instantly when the cartridge is inserted. No lengthy loading routine is required.

Your system also contains a certain amount of read/write memory, or RAM, shown in Fig. 1-2 as items ④ through ⑪.

The acronym *RAM* stands for *random access memory* and is usually associated with read/write memory. However, ROM is also random access memory. Thus, the term RAM, when associated *only* with read/write memory, is not technically correct. However, this is the accepted convention and we will not try to fight the masses. From this point on, we will refer to read/write memory uniquely as RAM.

Depending on the system you purchased, or are about to purchase, you have been provided with 4K, 16K, or 32K bytes of RAM. The 4K systems use eight 4K-bit RAM devices. Each device provides 4096 storage locations for one bit position within an 8-bit data word. Thus, eight devices must be connected in parallel to provide 4096, or 4K, bytes of memory. The 16K systems use eight 16K-bit RAM devices connected in parallel to provide 16K bytes of RAM. The 32K systems use eight 32K-bit RAM devices connected in parallel to provide 32K bytes of RAM. It is relatively easy to upgrade one memory system to another. Radio Shack provides this service for a nominal parts and installation fee.

You might have already discovered that just because you have purchased a 16K system doesn't mean you have 16K bytes of memory available for writing programs. In fact, a 16K system with Extended Color BASIC only provides 8487 bytes of *user* RAM. This is because the Extended Color BASIC interpreter reserves 7897 bytes of RAM for use by the interpreter. To find out how much *user* RAM is provided with your system, turn the system on and enter the command:

PRINT MEM **ENTER**

The computer will return the amount of user RAM available.

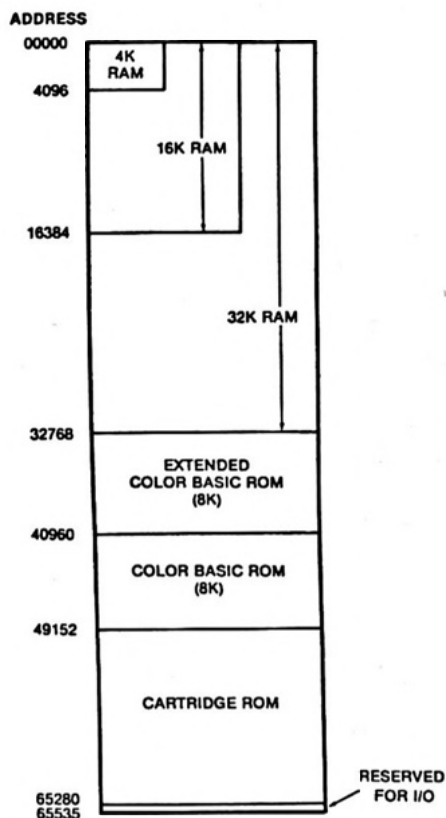
The TRS-80® Color Computer uses *dynamic* RAM. This type of RAM must be refreshed periodically to retain its information. The refreshing task is controlled by a dynamic RAM refresh controller, labeled as item ⑫ in Fig. 1-2. The refresh controller relieves the microprocessor of the RAM refresh task. You have probably already discovered that any information stored in RAM is lost when power is removed from the computer. This is because all RAM is *volatile*.

The 6809 allows for a 64K byte address space. *All memory and peripheral I/O devices must be assigned to unique addresses.* Like the post office, the microprocessor uses these addresses to deliver and collect data from memory and peripheral I/O devices. A memory map of the Color Computer is shown in Fig. 1-3.

A memory map is a table that provides the address assignments to memory and peripheral I/O devices for a given system.

For now, it is only important that you are aware of the general Color Computer address assignments as shown in Fig. 1-3. We will get into specific address assignments later. Notice from Fig. 1-3 that the computer RAM is assigned to the lower 32K addresses. The amount

Figure 1-3 _____
The Color Computer memory
map.



of address allocation depends on how much RAM the system contains—4K, 16K, or 32K. The next 16K addresses are assigned to the computer ROM. All 16K addresses are used for Extended Color BASIC. Only 8K addresses are used for Color BASIC. Finally, the upper 16K addresses are used for the ROM cartridges and control of peripheral data I/O.

The Video Display and Color Computer Input/Output

Your TV display is the major peripheral output device of the Color Computer. The heart of the computer video display circuit is the *6847 video display generator*, or VDG. (Refer to item ⑬ in Fig. 1-2.) The VDG controls all the TV display operations. To output alphanumeric information to a TV, the CPU will *write* the information to be displayed into a 512-byte block of RAM. This section of RAM is called the *text screen memory* and is located in the memory map between RAM

addresses 1024 and 1535. This area of RAM is reserved for the TV display. Recall that your computer displays text material in 16 lines of 32 characters per line. This provides a maximum display of 512 characters on the TV screen.

The display information for each character on the screen is represented by one byte of RAM within the text screen memory. The first byte of text screen memory contains the display information for the character position in the upper left hand corner of the TV screen. The second text screen memory address contains the display information for the character position immediately to the right of the first, and so on until the top line of TV characters is contained in the first 32 bytes of the text screen memory. The second line of the TV display is provided by the second 32 bytes of the text screen memory. This *mapping* continues until the 16th line of the TV display is provided by the last 32 bytes of the text screen memory. The text screen memory is, therefore, *mapped* to the TV screen as shown in Fig. 1-4.

As you will soon discover, with the 6809 CPU, *all peripheral I/O devices such as displays, keyboards, printers, etc., must be assigned to an address or group of addresses within the computer memory map.* This is called *memory mapped I/O*.

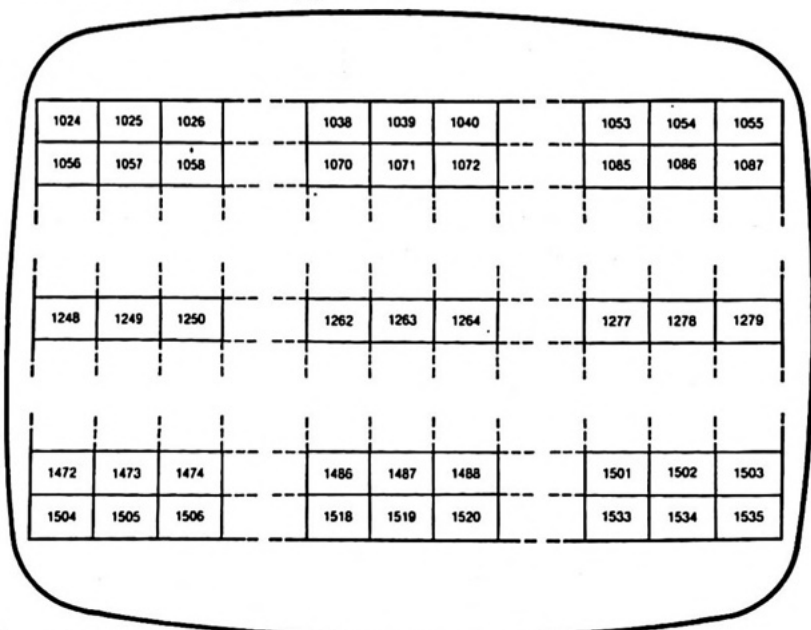


Figure 1-4 Mapping of the TV display to the text screen memory.

The TV display is a peripheral *output* device. Thus, to transfer information from the computer to the TV display, you must use a command that sends information *out* of the computer to an address assigned to the TV display. In the BASIC language, one such output command is the POKE instruction. Recall that the POKE instruction allows you to change the contents of any given address location. The format for the POKE instruction is:

POKE <address> , <data>

When the POKE instruction is executed, the data value in the instruction is *written*, or transferred, to the address contained in the instruction. The data value cannot be larger than 255, since a given address can only contain an 8-bit data value. The address in the POKE instruction must be between 0 and 65535, or 64K, since the 6809 CPU provides a 64K address space. For example, enter the following instruction in your computer:

POKE 1535,88 **ENTER**

You should observe an X in the lower right hand corner of the TV screen. This is because address 1535 is the last address in the text screen memory and is mapped to the lower right corner position of the TV display (see Fig. 1-4). Also, the data value 88 is a code for the letter X. Each character on your color computer keyboard has a *character code* as shown in Table 1-1. To transfer a character to a peripheral output device such as the TV display, you simply POKE its character code value to an address assigned to that device. There are character codes to display characters in *standard video* (black on green) or *reverse video* (green on black).

Now, change the address in the previous instruction to 1263. What should happen when the POKE instruction is executed? The X should now be displayed in the middle of the screen since address 1263 is mapped to the middle position of the TV screen (see Fig. 1-4).

Next, change the data value in the POKE instruction to 24. Now, the X should be displayed in the middle of the screen using reverse video. You might want to try different display characters and locations to verify the text screen memory mapping.

There are also *graphics characters* available to the computer as shown in Fig. 1-5. Like the alphanumeric keyboard characters, each graphics character has its own character code. To display a graphics character, you simply output its code to one of the text screen ad-

Table 1-1. The Alphanumeric Character Codes for the Color Computer

Character	Decimal Code	Character	Decimal Code	Character	Decimal Code
!	33	5	53	I	73
"	34	6	54	J	74
#	35	7	55	K	75
\$	36	8	56	L	76
%	37	9	57	M	77
&	38	:	58	N	78
'	39	;	59	O	79
(	40	<	60	P	80
)	41	=	61	Q	81
*	42	>	62	R	82
+	43	?	63	S	83
,	44	@	64	T	84
-	45	A	65	U	85
.	46	B	66	V	86
/	47	C	67	W	87
0	48	D	68	X	88
1	49	E	69	Y	89
2	50	F	70	Z	90
3	51	G	71	☐	94
4	52	H	72	☐	95

NOTE: See the Color Computer User's Manual for the reverse video character codes.

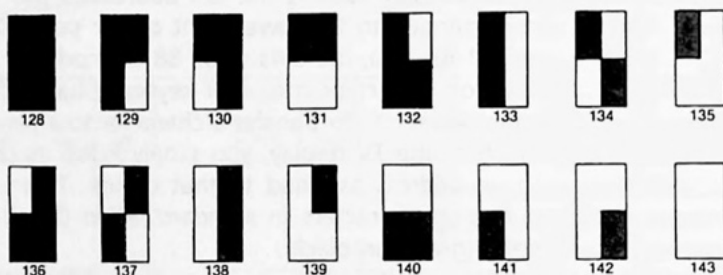


Figure 1-5 The graphic character codes for the Color Computer.

addresses. For example, enter the following instruction in your computer:

POKE 1263,137 **ENTER**

Graphics character #137 (see Fig. 1-5) should appear at the center of the screen. Now change the character code in the previous instruction to 185. Graphics character #137 is again displayed but the green area is now red.

All of the Color Computer alphanumeric and graphics characters

can be displayed on your TV screen by running the following program:

```
10 CLS
20 X = 1023
30 FOR Y = 0 to 255
40 X = X+1
50 POKE X,Y
60 NEXT Y
70 GOTO 70
```

Notice that this program simply writes all the Color Computer character codes to the first 256 text screen memory addresses.

The preceding discussion should give you a good idea of how information is output to a peripheral output device like your TV display. What about input? The idea is the same; each peripheral input device is assigned to an address or group of addresses within the computer memory map. To input data from the peripheral input device, you simply *read* its address. The BASIC command that allows you to input data is the PEEK instruction. The PEEK instruction format is:

PEEK <address>

When the PEEK instruction is executed, data contained at the address specified in the instruction is input, or read, into the computer. Again, the address contained in the PEEK instruction must be between 0 and 65535 since the CPU provides a 64K address space.

The simplest peripheral input device is a switch. If you have joystick controls for your computer, the push button on each control is a simple switch. Connect one of the joystick controls to your computer. It doesn't matter which one, right or left. Now enter and run the following program:

```
10 CLS
20 A = PEEK (65280)
30 POKE 1263,A
40 GOTO 20
```

You should observe a solid red/pink graphics character block in the middle of your TV display. Now, press the joystick push button. Each time you press the button, a black rectangle will appear inside the red/pink block.

The joystick push buttons are assigned to address 65280. The PEEK instruction in the program inputs, or reads, the data at this address,

then the POKE instruction outputs, or writes, the data to the center screen address of the TV display. When you press the joystick push button, you are changing the data being input from the joystick at address 65280. Consequently, the TV display character is changed.

This simple, but important, example shows how data can be read from a single input device using the PEEK instruction, then written to an output device using the POKE instruction. In addition, the addresses chosen for these instructions were *not* arbitrary since every peripheral input or output device must be assigned to a unique address, or group of addresses, within the computer memory map.

THE TRS-80® COLOR COMPUTER EXPANSION PORTS

The TRS-80® Color Computer has four *ports* that can be used to expand your system.

A *port* is an electrical connection point at which the system CPU can be interfaced to external memory or peripheral I/O devices.

The four ports provided by the Color Computer are the:

- Parallel expansion connector port.
- Serial RS-232-C port.
- Joystick interface, or analog input, port.
- Audio cassette port.

Any interfacing to the Color Computer must be accomplished through one of these I/O ports. In this section, we will provide an introductory discussion of each port. Then, in subsequent chapters, we will discuss the Color Computer ports in more detail as they are used for interfacing.

Parallel Expansion Port

The Color Computer parallel expansion connector is located on the right side of the case. It is used by Radio Shack as an interface for the ROM cartridges, or Program Paks™, and the Color Computer floppy disk system. You will use this port for many of the interfacing experiments provided in subsequent chapters.

The expansion connector is a 40-pin female connector. The connector pins and their functions are listed in Fig. 1-6. The expansion connector port is a parallel port since all eight CPU data lines (D0–D7) are provided on the connector. Furthermore, all 16 of the CPU address

Pin No.	Signal Name	Description
1	- 12 V	- 12 volts (100 mA)
2	+ 12 V	+ 12 volts (300 mA)
3	HALT	Halt Input to the CPU
4	NMI	Nonmaskable Interrupt to the CPU
5	RESET	Main Reset and Power-Up Clear Signal to the System
6	E	Main CPU Clock (0.89 MHz)
7	Q	Quadrature Clock Signal which Leads E
8	CART	Interrupt Input for Cartridge Detection
9	+ 5 V	+ 5 volts (300 mA)
10	D ₀	Data Bit 0
11	D ₁	Data Bit 1
12	D ₂	Data Bit 2
13	D ₃	Data Bit 3
14	D ₄	Data Bit 4
15	D ₅	Data Bit 5
16	D ₆	Data Bit 6
17	D ₇	Data Bit 7
18	R/W	Read/Write Signal
19	A ₀	Address Bit 0
20	A ₁	Address Bit 1
21	A ₂	Address Bit 2
22	A ₃	Address Bit 3
23	A ₄	Address Bit 4
24	A ₅	Address Bit 5
25	A ₆	Address Bit 6
26	A ₇	Address Bit 7
27	A ₈	Address Bit 8
28	A ₉	Address Bit 9
29	A ₁₀	Address Bit 10
30	A ₁₁	Address Bit 11
31	A ₁₂	Address Bit 12
32	CTS	Cartridge Select Signal
33	GND	Signal Ground
34	GND	Signal Ground
35	SND	Sound Input
36	SCS	Spare Select Signal
37	A ₁₃	Address Bit 13
38	A ₁₄	Address Bit 14
39	A ₁₅	Address Bit 15
40	SLENB	Input to Disable Device Selection

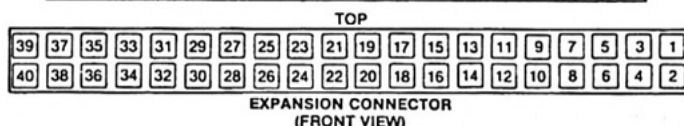


Figure 1-6 Pin assignments for the parallel expansion connector.

lines (A0–A15) are available. This allows the CPU to provide addresses to access external peripheral I/O devices assigned to any part of its 64K memory map.

Notice also from Fig. 1-6 that three power supply voltages are provided on the connector—+5 volts, +12 volts, and -12 volts. These three dc supply voltages will allow you to interface several devices to the computer without having to supply additional power. The remaining expansion connector pins provide for external peripheral device timing, addressing, and control. Each of these signals will be discussed at the appropriate time in subsequent chapters.

Serial RS-232-C Port

The RS-232-C port is located on the back of the Color Computer case and is labeled SERIAL I/O. Data communication through this port is serial, meaning that there is a sequential stream of bits, rather than eight data bits at a time as provided by the parallel expansion port. The advantage of serial communication is that only two lines are required to communicate any number of bits—a transmit line and a receive line. With parallel communication, a line must be provided for each data bit. This requirement is obviously a disadvantage, especially when communicating over long distances.

The disadvantage of serial communication is speed, since only one bit may be transmitted or received at a time. Printers are usually serial I/O devices since they are often located some distance from the computer. In addition, long distance communication over telephone lines is serial. Consequently, the device used to interface your computer to the telephone network, called a *modem*, is also a serial I/O device and must be connected to a serial port on your computer. Radio Shack uses the serial I/O port of the Color Computer for both printer and modem interfacing.

You are probably wondering what RS-232-C means, if you don't already know. RS-232-C is a serial communications standard written by the Electronics Industry Association (EIA). This standard specifies the electrical parameters for serial communication as well as the physical layout of the connector pins. Such a standard allows different manufacturer's equipment to be easily interfaced without any special adaptations. This is important to you because you are not locked into purchasing one manufacturer's, namely Radio Shack's, peripheral equipment for your computer. While Radio Shack has adhered to the electrical requirements of the RS-232-C standard, they have not followed the physical connector requirements of the standard. The RS-232-C standard specifies a 25-pin flat connector, and the Color Computer uses a 4-pin round connector for the RS-232-C serial interface. This means that you will most likely have to provide an adapter cable between the Color Computer and an RS-232-C serial peripheral manufactured by someone other than Radio Shack. Most peripheral manufacturers can supply such an adapter cable at a nominal cost.

Finally, the four RS-232-C serial I/O connector pins and their functions are shown in Fig. 1-7. Aside from the ground pin, notice that there is a serial transmit line, serial receive line, and a carrier detect line. The carrier detect line is used with modems to control the serial data transfer operation over the telephone system. The serial com-

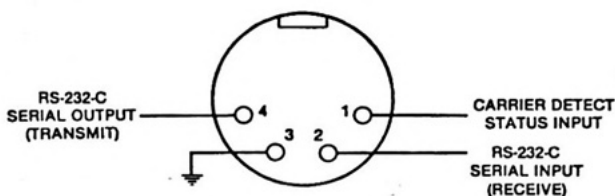


Figure 1-7
Pin assignments for the RS-232-C serial I/O connector.

munication rate provided by the Color Computer is 600 baud, meaning that 600 bits are transmitted or received per second. Any serial I/O device, such as a printer, connected to this port must match the 600 baud rate. However 120, 300, 1200, and 2400 baud rates are possible by modifying certain variables located in RAM.

Since the 6809 is a parallel device, conversions must be made from parallel to serial when transmitting information to a serial I/O device, and from serial to parallel when receiving information from a serial I/O device. The required conversions are provided by the internal software of the Color Computer. The parallel/serial conversion software is located in ROM.

Joystick Interface, or Analog Input, Ports

Signals coming from the joysticks are analog signals and must be converted to digital signals to be used by the computer.

An analog signal is one that can assume any value within a given range. Analog signals are usually changing continuously. *A digital signal* is one that can assume only one of two possible states—on or off, 1 or 0.

Two connectors are provided on the back of the Color Computer case for this purpose. Each connector has five pins as shown in Fig. 1-8. You can use the X and Y input pins of these connectors to

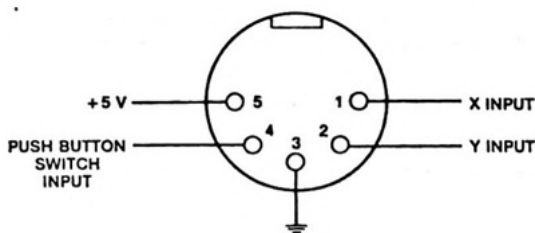


Figure 1-8
Pin assignments for the joystick interface, or analog input connector.

monitor four separate analog signals. The computer will convert analog input voltages from zero volts to 5 volts to decimal values between 0 and 63. The software that provides the analog to digital conversions is contained in the BASIC interpreter ROM. We will provide a more detailed discussion of these ports in Chapter 5, when we discuss analog interfacing to the Color Computer.

Audio Cassette Port

The audio cassette port is located on the back of the Color Computer and is labeled CASSETTE. As you are probably aware, this port is used to connect a low-cost audio cassette recorder to the Color Computer for nonvolatile data and program storage. The output of this port is used to turn the cassette motor on and off. In addition, when operating in the record (or save) mode, this port generates the necessary signals to represent binary information on the audio tape. When operating in the play (or load) mode, audio signals from the tape are fed into this port and converted into streams of binary bits that represent instructions and data to the CPU. The port connector pins and their functions are shown in Fig. 1-9. The software that controls the operation of this port is contained in the BASIC interpreter ROM. You will use the cassette port to provide an analog output from the computer. The Color Computer contains an internal digital-to-analog converter, or DAC. The output of the internal DAC is available

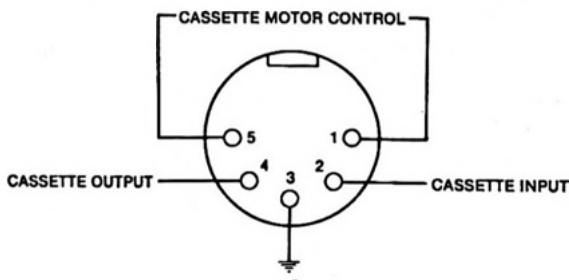


Figure 1-9
Pin assignments for the audio cassette interface connector.

at pin 4 of the cassette interface connector. In addition, you can control small dc loads via pin 5 of the cassette interface connector. These features will be discussed in Chapter 5 when we discuss analog interfacing.

SUMMARY

In summary, this chapter provided you with an introduction to the internal hardware and expansion ports of the TRS-80® Color Computer. You discovered that the brain of the Color Computer is an 8-bit microprocessor, or CPU, which is capable of providing a 64K address space. All memory and peripheral I/O devices must be assigned addresses within the address space. This is called *memory mapped I/O*. You have also learned that to communicate with a peripheral I/O device, you must use I/O commands such as PEEK and POKE. Finally, you found that the Color Computer has a parallel I/O expansion port, an RS-232-C serial I/O port, and analog ports that can be used for interfacing to external devices.

2 The 6809E Microprocessor, or CPU

The TRS-80® Color Computer uses the 6809E microprocessor integrated circuit. The 6809E is the *central processing unit*, or *CPU*, of the computer. All mathematical, logical, data transfer, and decision making operations for the Color Computer are performed by the 6809E. The 6809E was designed and is manufactured by Motorola Semiconductors, Austin, Texas.

In this chapter, you will become acquainted with the 6809E. We do not intend to make you an expert on programming and interfacing the 6809E. Our intent here is to acquaint you with the various features of the 6809E as they relate to the Color Computer. This will provide you with the background required for interfacing the Color Computer in subsequent chapters. If you desire an in-depth discussion of 6809E assembly language programming and interfacing, we recommend that you obtain a copy of *6809 Microcomputer Programming and Interfacing, with Experiments*, published by Howard W. Sams & Co., Inc.

OBJECTIVES

At the end of this chapter you should be able to:

- Explain the two evolutionary paths that the microprocessor industry has taken to satisfy the microcomputer application markets, and how the Color Computer fits in this scheme.
- Describe the internal register structure of the 6809E.

- State the function of the 6809E interface lines that can be used for Color Computer interfacing.
- Understand the timing relationships of several 6809E interface lines during the execution of the PEEK and POKE instructions.
- List several Color Computer BASIC instructions that are used for peripheral I/O control.

GENERAL CONCEPTS AND EVOLUTION

The 6809E is a *second-generation 8-bit microprocessor*. Let's analyze that statement. The 6809E is called a second-generation device since it was designed to enhance the performance of first-generation microprocessors like the 6800, Z80, 6502, and 8080, among others. These first-generation microprocessors are general-purpose programmable logic devices. Since the introduction of these first-generation devices in the early 1970s, two major microprocessor applications markets have developed. The first, and by far the largest, is the *dedicated applications market*. Because of the low cost of a microprocessor and its related components, it became possible to *dedicate* a computer intelligence to almost anything. You find these dedicated applications everywhere—toys, games, appliances, industrial controls, and automobiles, just to name a few. In all of these applications, the microprocessor is dedicated to performing one or more fixed tasks, determined by the application. There might be a certain degree of user programmability, but in general, the microprocessor program and, therefore, function is fixed.

The second major market that has developed around the microprocessor is the *microcomputer systems market*. The TRS-80® Color Computer is in this category. In general, the microcomputer systems market is limited to personal and small-business computers. In a systems application, the microprocessor is user-controlled and programmable to perform a variety of data processing tasks. Companies that manufacture microcomputer systems now include Radio Shack, Apple, Cromemco, IBM, Heath, and Xerox, among others. As the microprocessor has evolved, these systems have become more powerful. Future microcomputer systems will be *multiprocessor* systems that use a high-performance microprocessor as the CPU and dedicated microprocessors for peripheral control. Dedicating a microprocessor for peripheral control frees the central processor of the peripheral control task, and thus allows it to handle more important data processing and system supervisory tasks.

Each of the preceding application markets, dedicated and systems, requires different features of a microprocessor. The dedicated market requires a compact device with various *internal* hardware features such as RAM, ROM, and peripheral I/O. Power requirements and the system chip count must be kept to a minimum. The design must be simple. Consequently, *single-chip microcomputers* have evolved to fulfill these design requirements. That's right, the entire computer is integrated into one large scale integrated (LSI) or very large scale integrated (VLSI) circuit, or *chip*. The systems market, on the other hand, requires a fast and efficient software device. Chip count, power requirements, and design simplicity are not as important. The systems-related microprocessor must be capable of efficiently handling higher-level computer languages such as BASIC, Pascal, FORTRAN, and COBOL. Consequently, high-performance microprocessors like the 6809 have evolved to fulfill system design requirements.

The hardware and software limits of the first-generation microprocessors were stretched in order to meet the requirements of both application markets. The microprocessor manufacturers have, therefore, taken two evolutionary paths within their product lines—one to develop single-chip microcomputers for the dedicated market, and the other to provide high-performance microprocessors for the systems market. The evolution of one manufacturer's microprocessor product line, namely Motorola, is shown in Fig. 2-1. At one end of the spectrum is the 6801, a complete single-chip microcomputer. At the other end of the spectrum is the 68000 microprocessor. The 6809 is a high-performance microprocessor that approaches the performance of the 68000. However, the 6809 is an 8-bit microprocessor and the 68000 is a 16-bit microprocessor. That brings us to the second part of the statement we made earlier—the 6809 is a second-generation *8-bit* microprocessor.

The 6809 is an 8-bit processor since it uses an 8-bit data bus to transfer information between itself and memory or peripheral devices such as keyboards, printers, etc. Each data bit within the 8-bit data word, or *byte*, can be a logic 1 or a logic 0. Thus, the largest data value that can be transferred at any given time is $1111\ 1111_2$, or 255_{10} . If larger values are to be transferred, then multibyte operations are required. The 68000, on the other hand, is a 16-bit microprocessor. Thus, it can transfer information 16-bits, or 2 bytes, at a time. Consequently, the 68000 can transfer data values up to $1111\ 1111\ 1111\ 1111_2$, or 65535_{10} at one time.

The 68000 can, therefore, process data more efficiently than the

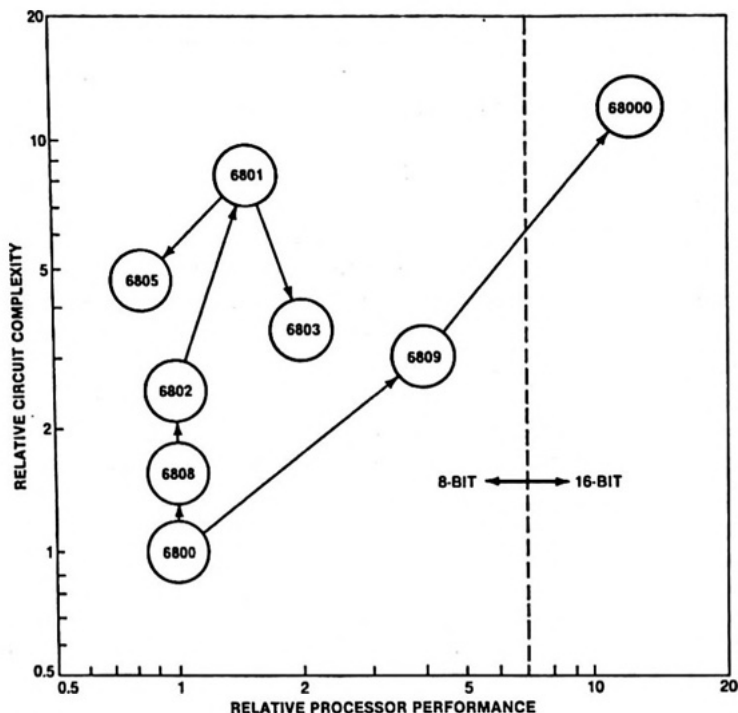


Figure 2-1 The evolutionary path of the Motorola microprocessor product line.

6809. However, as you are about to discover, the internal registers of the 6809 are 16 bits long. Thus, the internal performance of the 6809 approaches that of many 16-bit microprocessors, like the 68000. It is important to remember, though, that the Color Computer CPU, the 6809E, can only *transfer* eight bits, or one byte, of data at a time to memory and peripheral I/O devices.

The last part of the statement we made earlier said that the 6809 is an 8-bit *microprocessor*. The 6809 is called a microprocessor and not a microcomputer since it cannot do anything by itself. Memory and peripheral I/O devices must be added to the 6809 to create a workable microcomputer system.

THE INTERNAL STRUCTURE AND INSTRUCTION SET OF THE 6809

The internal register structure, or *architecture*, of the 6809 is shown in Fig. 2-2. These are the registers that are accessible to the program-

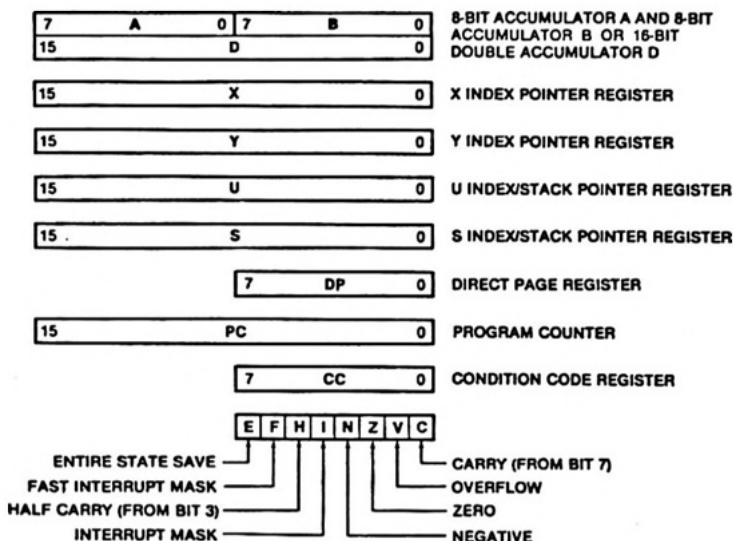


Figure 2-2 The internal register structure, or architecture, of the 6809.

mer with a 6809 assembly language program but not with a BASIC program. Notice first that most of the internal registers are 16-bit registers. Consequently, the 6809 can operate internally with 16-bit data values even though it can only transfer eight bits at a time externally. This internal 16-bit architecture is one reason the 6809 is a very-high-performance 8-bit microprocessor.

The 6809 uses an *accumulator architecture*, meaning that operands to be used in an operation, and the results of an operation, are temporarily placed in an accumulator. The 6809 has three *accumulators* — A, B, and D. Accumulators A and B are eight bits long and accumulator D is 16 bits long. Actually, accumulators A and B are combined to form accumulator D.

All arithmetic and logic operations performed by the 6809 are performed in its *arithmetic logic unit*, or ALU. The ALU is not shown in Fig. 2-2 since it is not accessible to you, the programmer, even with an assembly language program. Instead, operands to be used in an operation are loaded into one of the accumulators from external memory or a peripheral device. The 6809 then takes the operand contained in the accumulator and performs the operation, say addition, in the ALU. The operation result is placed back in the original ac-

cumulator. Thus, the accumulators are used as temporary storage registers for operands before an operation, and results after the operation. Once the result of an operation is generated by the ALU and placed in one of the accumulators, it is usually stored to external RAM or a peripheral device for semipermanent storage. This accumulator storage operation is usually required since a subsequent operation might destroy the result of an earlier operation.

The 6809 contains four 16-bit *index/stack pointer registers* labeled X, Y, U, and S in Fig. 2-2. These registers are used as *address pointer registers* when accessing external memory or peripheral devices. In other words, they are used by the 6809 to form, or point to, the address of an 8-bit operand within the 64K address space. In addition, these registers are used as *stack pointers* to locate a stack within memory.

A memory stack is an area of RAM that is used during a subroutine or interrupt to temporarily save the internal register contents of the processor while the subroutine or interrupt service routine is being executed.

Notice from this discussion that in general the X, Y, U, and S registers are used to supply addresses to locate data within the address space.

The remaining 16-bit register in Fig. 2-2 is the *program counter*, or PC. The PC is simply a binary counter that provides instruction sequencing. The count in the program counter is incremented by the microprocessor clock. The 6809 PC is also used to calculate program branching destinations and to locate data in memory.

The *direct page register* of the 6809 is used to break up the address space into 256 *pages* of 256 bytes each. Note that $256 \times 256 = 64K$. Thus, each address page contains 256 bytes. This memory paging is sometimes convenient when accessing routines and subroutines, since different routines can be assigned to different pages, by function, within the 64K address space using the direct page register.

Finally, the 6809 contains an 8-bit *condition code register*, or CCR. The CCR contains eight *status flags* that are used to make decisions during the execution of a program. The C, V, Z, N, and H flags indicate the status of an operation result. For example, if the result of an operation is zero, the Z flag automatically sets (logic 1). The 6809 can then make a decision based upon the Z-flag status if instructed to do so. The decision to be made is a program branch, or IF-THEN-GOTO type of decision. The I, F, and E flags of the CCR are used in conjunction with the 6809 interrupt input lines. These lines will be discussed shortly.

There are 59 fundamental instructions in the 6809 instruction set, which is listed in Table 2-1. Notice that these instructions allow you to perform various operations using the internal registers of the 6809. You must use these 59 fundamental instructions along with 19 *addressing modes* to write assembly language programs for the Color Computer.

An *addressing mode* is a method used by the microprocessor to form memory or peripheral addresses. Different addressing modes allow a given instruction to access memory of peripherals in different ways.

A summary listing of the 6809 addressing modes is provided in Table 2-2. The 59 fundamental instructions and 19 addressing modes of the 6809 allow you to perform over 1400 unique software operations with the 6809. Of course, the Color Computer BASIC interpreter allows you to program in BASIC without knowing the 6809 instruction set and addressing modes. Notice from Table 2-1 that the 6809 instructions do not resemble any of the Color Computer BASIC instructions that you are familiar with. Since we will not be programming the Color Computer in assembly language, we do not intend to discuss the 6809 instruction set and addressing modes any further. If you desire a complete discussion of this material and want to write 6809 assembly language programs, we suggest that you consult *6809 Microcomputer Programming and Interfacing, with Experiments*, published by Howard W. Sams & Co., Inc.

THE 6809E INTERFACE LINES

The interface lines of any microprocessor, or CPU, are used to form three external signal buses called the *data bus*, *address bus*, and *control bus* as illustrated in Fig. 2-3. The data bus is used to transfer information between the CPU and memory or peripheral I/O devices. Notice from Fig. 2-3 that the data bus is a bidirectional bus. The logic signals on the data bus tell the system *what* information is being transferred. The address bus is used by the CPU to tell the system *where* the data information is to be transferred to, or transferred from. Notice that the address bus is a unidirectional bus from the CPU to memory and peripheral I/O devices. Finally, the signals on the control bus are used to synchronize the data transfer by telling the system *when and how* the data is to be transferred.

Up to this point, we have used the designations 6809 and 6809E

Table 2-1. The 6809 Instruction Set

8-BIT ACCUMULATOR AND MEMORY INSTRUCTIONS	
Mnemonic(s)	Operation
ADCA, ADCB	Add memory to accumulator with carry
ADDA, ADDB	Add memory to accumulator
ANDA, ANDB	AND memory with accumulator
ASL, ASLA, ASLB	Arithmetic shift of accumulator or memory left
ASR, ASRA, ASRB	Arithmetic shift of accumulator or memory right
BITA, BITB	Bit test memory with accumulator
CLR, CLRA, CLRB	Clear accumulator or memory location
CMPA, CMPB	Compare memory from accumulator
COM, COMA, COMB	Complement accumulator or memory location
DAA	Decimal adjust A accumulator
DEC, DECA, DECB	Decrement accumulator or memory location
EORA, EORB	Exclusive-OR memory with accumulator
EXG R1, R2	Exchange R1 with R2 (R1, R2 = A, B, CC, DP, X, Y, S, U)
INC, INCA, INCB	Increment accumulator or memory location
LDA, LDB	Load accumulator from memory
LSL, LSLA, LSLB	Logical shift left accumulator or memory location
LSR, LSRA, LSRB	Logical shift right accumulator or memory location
MUL	Unsigned multiply ($A \times B \rightarrow D$)
NEG, NEGA, NEGB	Negate accumulator or memory
ORA, ORB	OR memory with accumulator
ROL, ROLA, ROLB	Rotate accumulator or memory left
ROR, RORA, RORB	Rotate accumulator or memory right
SBCA, SBCB	Subtract memory from accumulator with borrow
STA, STB	Store accumulator to memory
SUBA, SUBB	Subtract memory from accumulator
TST, TSTA, TSTB	Test accumulator or memory location
TFR R1, R2	Transfer R1 to R2 (R1, R2 = A, B, CC, DP, X, Y, S, U)

16-BIT ACCUMULATOR AND MEMORY INSTRUCTIONS	
Mnemonic(s)	Operation
ADDD	Add memory to D accumulator
CMPD	Compare memory from D accumulator

Table 2-1 — cont. The 6809 Instruction Set

Mnemonic(s)	Operation
EXG D, R	Exchange D with X, Y, S, U, or PC
LDD	Load D accumulator from memory
SEX	Sign Extend B accumulator into A accumulator
STD	Store D accumulator to memory
SUBD	Subtract memory from D accumulator
TFR D, R	Transfer D to X, Y, S, U, or PC
TFR R, D	Transfer X, Y, S, U, or PC to D

INDEX REGISTER/STACK POINTER INSTRUCTIONS

Mnemonic(s)	Operation
CMPS, CMPU	Compare memory from stack pointer
CMPX, CMPY	Compare memory from index register
EXG R1, R2	Exchange D, X, Y, S, U, or PC with D, X, Y, S, U, or PC
LEAS, LEAU	Load effective address into stack pointer
LEAX, LEAY	Load effective address into index register
LDS, LDU	Load stack pointer from memory
LDX, LDY	Load index register from memory
PSHS	Push A, B, CC, DP, D, X, Y, U, or PC onto hardware stack
PSHU	Push A, B, CC, DP, D, X, Y, S, or PC onto user stack
PULS	Pull A, B, CC, DP, D, X, Y, U, or PC from hardware stack
PULU	Pull A, B, CC, DP, D, X, Y, S, or PC from hardware stack
STS, STU	Store stack pointer to memory
STX, STY	Store index register to memory
TFR R1, R2	Transfer D, X, Y, S, U, or PC to D, X, Y, S, U, or PC
ABX	Add B accumulator to X (unsigned)

BRANCH INSTRUCTIONS

Mnemonic(s)	Operation
SIMPLE BRANCHES	
BEQ, LBEQ	Branch if equal
BNE, LBNE	Branch if not equal
BMI, LBMI	Branch if minus
BPL, LBPL	Branch if plus
BCS, LBCS	Branch if carry set

Table 2-1 — cont. The 6809 Instruction Set

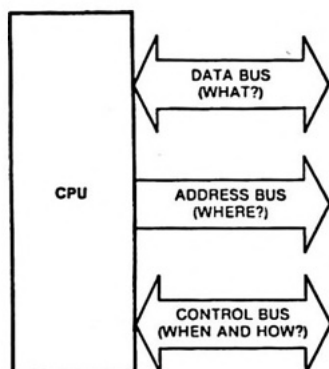
Mnemonic(s)	Operation
BCC, LBCC	Branch if carry clear
BVS, LBVS	Branch if overflow set
BVC, LBVC	Branch if overflow clear
SIGNED BRANCHES	
BGT, LBGT	Branch if greater (signed)
BGE, LBGE	Branch if greater than or equal (signed)
BEQ, LBEQ	Branch if equal
BLE, LBLE	Branch if less than or equal (signed)
BLT, LBLT	Branch if less than (signed)
UNSIGNED BRANCHES	
BHI, LBHI	Branch if higher (unsigned)
BHS, LBHS	Branch if higher or same (unsigned)
BEQ, LBEQ	Branch if equal
BLS, LBLs	Branch if lower or same (unsigned)
BLO, LBLO	Branch if lower (unsigned)
OTHER BRANCHES	
BSR, LBSR	Branch to subroutine
BRA, LBRA	Branch always
BRN, LBRN	Branch never
MISCELLANEOUS INSTRUCTIONS	
Mnemonic(s)	Operation
ANDCC	AND condition code register
CWAI	AND condition code register, then wait for interrupt
NOP	No operation
ORCC	OR condition code register
JMP	Jump
JSR	Jump to subroutine
RTI	Return from interrupt
RTS	Return from subroutine
SWI, SWI2, SWI3	Software interrupt (absolute indirect)
SYNC	Synchronize with interrupt line

Table 2-2. A Summary of the 6809 Addressing Modes

• Inherent (Implied) • Immediate • Direct • Extended • Relative - Branch Relative - Program Counter Relative	• Register • Indirect • Indexed - Constant Offset (0-, 5-, 8-, and 16-bit) - Accumulator Offset (A, B, and D) - Auto Increment/Decrement by 1 - Auto Increment/Decrement by 2
--	---

interchangeably. The 6809 and 6809E are identical as far as their internal structure, instruction set, and addressing modes are concerned. However, their interface lines are different. Since the Color Computer uses the 6809E, we will confine our discussion to the 6809E interface lines.

Figure 2-3
The three external buses of any CPU—data, address, and control.



A pin assignment diagram of the 6809E is shown in Fig. 2-4. We do not intend to discuss all of the 6809E pin functions in detail. Only those pins that will be used to interface to the Color Computer will be covered here. The 40 pins of the 6809E in Fig. 2-4 can be functionally divided into four categories—*power/clock*, *data/address*, *bus status*, and *bus control*.

Power/Clock

The 6809E requires a single +5-volt dc power supply voltage to operate. Maximum current drain is 200 milliamperes. The two power connections are V_{SS} at pin 1 (ground) and V_{DD} at pin 7 (+5 volts dc).

Many digital devices, including microprocessors, must be *clocked*.

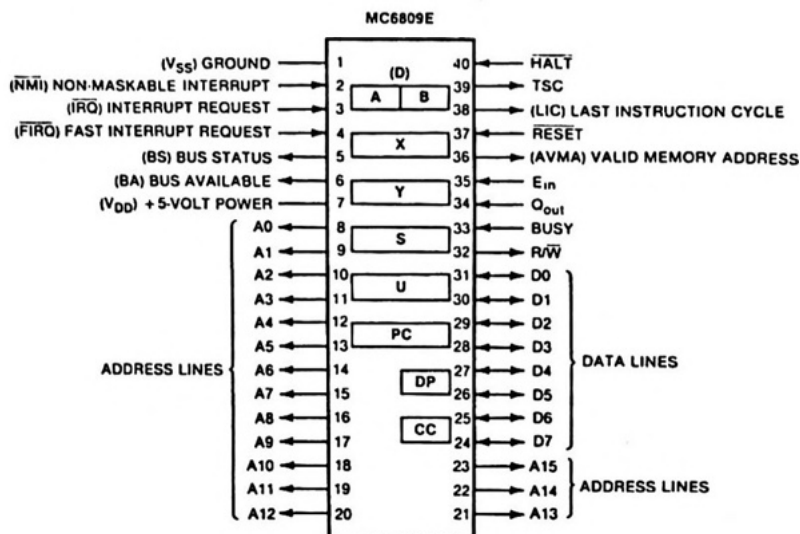
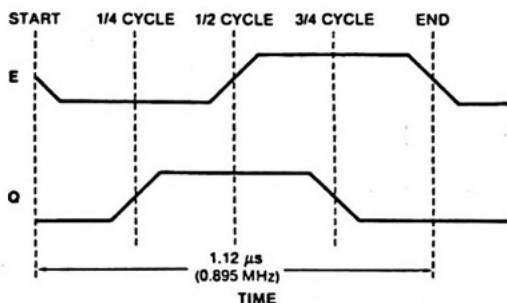
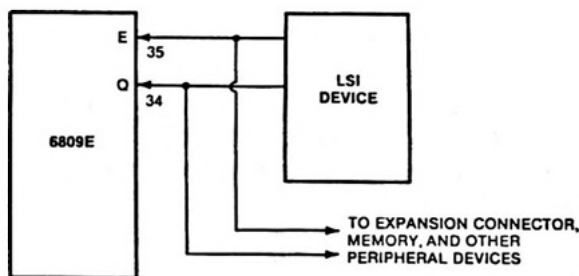


Figure 2-4 The interface lines and pin assignments of the 6809E.

Clocking synchronizes data transfers within the microprocessor as well as between the microprocessor and its external devices. A digital clock signal is simply a reproducible, periodic waveform at a specified frequency. The frequency of the clock signal depends on the microprocessor specification. The maximum allowable 6809E clock frequency is 1 megahertz (MHz). However, a lower clock frequency can be used. The Color Computer operates with a clock frequency of 0.895 MHz, which is equivalent to 895 kilohertz (kHz). The 6809E requires two clock signals, Q and E, as shown in Fig. 2-5A. Notice that Q and E are the same frequency (0.895 MHz) but are 90°, or ¼ clock cycle, out of phase. In the Color Computer, the Q and E clock signals are produced by another LSI device and applied to Q at pin 34 and E at pin 35 of the 6809E as shown in Fig. 2-5B. One or both of these clock signals must also be used when interfacing to the computer. Recall that all data transfers between the 6809E and external memory or peripheral devices are synchronized, or timed, by the microprocessor clock. This is why both the Q and E clock signals are provided on the Color Computer expansion connector (see Fig. 1-6). As you will see later, most of your interfacing applications will only require the use of the E clock signal.



(A) Phase relationship.



(B) Block diagram.

Figure 2-5
The Q and E clock signals required by the 6809E.

Data/Address

As you are now aware, the 6809E is an 8-bit microprocessor and, therefore, has eight data lines labeled D0–D7 in Fig. 2-4. These eight data lines make up the *data bus* of the Color Computer. The data bus is used to transfer information both to and from the computer. The data bus is *bidirectional* since it allows information to flow in two different directions, but only one direction at any given time.

The 6809E has 16 address output lines, labeled A0–A15, which form the *address bus* of the computer. The address bus is *unidirectional* since the 6809E address lines are output-only lines. Consequently, the 6809E supplies the addressing for the entire computer system. Since the address bus contains 16 lines, 2^{16} or 65536 addresses can be accessed. These addresses make up the 64K (K = 1024) address space of the 6809E that we discussed in Chapter 1. Of course, each address location contains 8 bits, or one byte, of data.

Bus Status

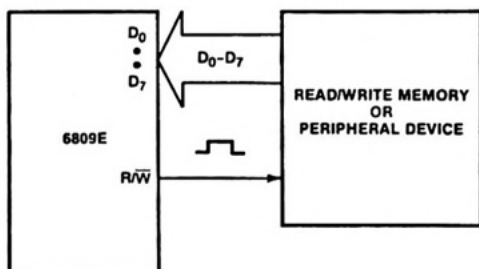
The 6809E bus status lines are used to communicate the status of the microprocessor to memory and peripheral devices. Some of these status lines would be used in multiprocessor computers, but since the Color Computer is not a multiprocessor computer, we won't discuss these signals. However, one of these lines is quite useful and very important to interface circuits. This is the *read/write*, or $R/\overline{W}$, line found at pin 32 on the CPU (see Fig. 2-4).

Recall that the CPU data bus is bidirectional. That is, data is transferred both to and from the microprocessor on the data bus. For the orderly transfer of data to and from the CPU and peripheral devices, the devices must know when the bus is being used for a read versus write operation. This careful synchronization between the CPU and all external devices, memory included, is the job of the $R/\overline{W}$ line. The $R/\overline{W}$ line *must* be used when interfacing to any peripheral device that is to use the data bus. During a read operation, data is transferred *from* memory or a peripheral device to the CPU. During a write operation the CPU transfers data to memory or a peripheral device. Notice the bar over the W in the $R/\overline{W}$ pin designation. When you see this bar over a pin designation, it means that the pin is *active low* during that operation, otherwise it is *active high*. Thus, during a read operation the $R/\overline{W}$ line goes "high" (logic 1), and during a write operation the $R/\overline{W}$ line goes "low" (logic 0). The read/write process is illustrated in Fig. 2-6. You will note from Fig. 1-6 that the $R/\overline{W}$ line is provided on the Color Computer expansion connector. Without it, you could not interface any peripheral device to the Color Computer.

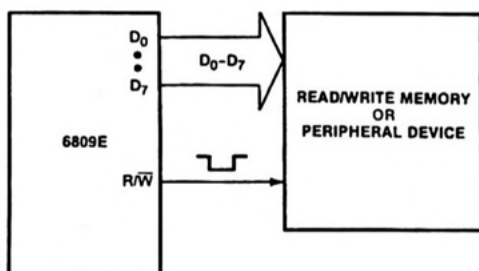
Bus Control

The bus control lines are all input lines that provide a means of externally controlling the microprocessor. These lines include three *interrupt lines*, the $\overline{RESET}$ line, and the $\overline{HALT}$ line as shown in Fig. 2-4.

The interrupt lines do exactly what the name implies—they provide a means of interrupting, or getting the attention of, the CPU. There are two general types of interrupts—*maskable* and *nonmaskable*. The CPU has the option of accepting or ignoring the maskable interrupt, while it has no choice but to accept a nonmaskable interrupt. Once an interrupt has been accepted, the CPU must determine which I/O device generated the interrupt. If several I/O devices generate interrupts simultaneously, they must be assigned a priority since only one device can be serviced at a time. The assignment of the priorities may be difficult and it usually requires extra hardware and software.



(A) The $\overline{R/W}$ line goes high (logic 1) when data is read.



(B) The $\overline{R/W}$ line goes low (logic 0) when data is written.

Figure 2-6
The read/write process.

Interrupts are analogous to a situation you have probably experienced—your supervisor talking with you in his office. While talking, his phone rings, representing an interruption. He answers the phone (servicing the interrupter), then continues his discussion with you—hopefully where he left off. If more than one call comes in at once, he must decide on which one to service first, second, and so on before getting back to you—if he ever does. However, before your talk begins, he could tell his secretary to “hold all calls” (maskable interrupt) until he completes his discussion with you.

The 6809E has two maskable interrupts—interrupt request ($\overline{IRQ}$) and fast interrupt ($\overline{FIRQ}$), as shown in Fig. 2-4. There is no point in discussing these interrupts since they are not provided on the Color Computer expansion connector for interfacing. However, they are used internally.

The two nonmaskable interrupts are the nonmaskable interrupt ($\overline{NMI}$) line and $\overline{RESET}$ line shown in Fig. 2-4 at pins 2 and 37, respec-

tively. Both of these nonmaskable interrupt lines are provided on the Color Computer expansion connector (see Fig. 1-6).

When the $\overline{\text{NMI}}$ input line is activated (low) by the application of an external signal, the 6809E will automatically break away from what it is doing and enter a special subroutine called an *interrupt service routine*. After the interrupt service routine has been executed, the 6809E will automatically return to what it was doing before the interrupt, as if nothing had happened.

Generally, an interrupt-causing device has a unique interrupt service routine associated with it. The interrupt service routine is written to provide the required software for the interrupt. For example, suppose the computer is controlling the operation of a drill press. The operation calls for drilling a series of holes in a wooden block. Once a hole has been drilled, the computer must direct the drill press to remove the bit from the hole and move to the next hole position. Obviously you must program the computer to remove the drill bit from each hole *before* it drills into the drill-press table. A simple way to do this is by using a nonmaskable interrupt. The nonmaskable interrupt line of the microprocessor could be connected to a limit switch that would be actuated just before the drill bit drills into the table. Thus, when the switch is actuated, the microprocessor would cause the drill press to stop the downward drilling operation when it executed the nonmaskable interrupt service routine. The interrupt service routine would also reverse the direction of movement of the drill press and move on to the next hole to be drilled. This is a simple example, but it shows you how the $\overline{\text{NMI}}$ input could be used to force the CPU to perform some specific task.

The Color Computer does not make use of the nonmaskable interrupt ($\overline{\text{NMI}}$) line internally. It is provided externally on the expansion connector for external interfacing. We will explain how it can be used in Chapter 4 when we discuss expansion-connector interfacing.

The $\overline{\text{RESET}}$ interrupt line is used to initialize the system. When the $\overline{\text{RESET}}$ line is activated, the CPU will stop what it is doing and go to a $\overline{\text{RESET}}$ interrupt service routine. After the service routine has been executed, the CPU will *not* return to what it was doing before the $\overline{\text{RESET}}$ interrupt, since it is assumed that the system is to be reinitialized. In the Color Computer, the $\overline{\text{RESET}}$ interrupt service routine causes the Color BASIC or Extended Color BASIC copyright notice to be displayed on the TV monitor. The reset button on the back of the Color Computer case is connected directly to the 6809E $\overline{\text{RESET}}$ input line. The $\overline{\text{RESET}}$ input line is also provided on the expansion connector

so that you can reinitialize the computer as the result of an external event.

Finally, the $\overline{\text{HALT}}$ control line is used to "disconnect" the 6809E from the internal system data and address bus structure. When the $\overline{\text{HALT}}$ input line is activated, the 6809E will finish its current instruction operation and internally disconnect its data, address, and R/W lines from the system bus structure as long as the $\overline{\text{HALT}}$ line is active. Programs or data in memory are not affected by activating $\overline{\text{HALT}}$. Furthermore, the 6809E will begin processing where it left off when the $\overline{\text{HALT}}$ line is deactivated, or returned high.

The $\overline{\text{HALT}}$ input is normally used for system troubleshooting, since it can be used to stop the microprocessor at any given time. Another use for the $\overline{\text{HALT}}$ line is to allow an external device, such as another microprocessor, to take control of the system bus structure. Since the $\overline{\text{HALT}}$ line is provided on the expansion connector, your Color Computer could be interfaced to another "host" computer to create a multiprocessor system. We think you will find that Radio Shack has this in mind for the future.

I/O INSTRUCTIONS AND TIMING

Before we go any further, we need to discuss the BASIC instructions that the Color Computer uses to transfer data between the CPU and memory or peripheral devices. These instructions are called input/output, or I/O, instructions. You are already familiar with PEEK and POKE. These are two general-purpose I/O instructions that allow the contents of a specific address location to be examined (displayed), and they also allow a value to be stored at a specific address. The data values that may be stored or retrieved are 8-bit values and, therefore, are limited to a range of 0 to 255, inclusive. The address that is accessed by the PEEK or POKE instruction may be any address within the 64K address space, from 0 to 65535.

To illustrate these range, or argument, limits enter the following instruction into your computer:

```
POKE 4000,256 ENTER
```

The computer will display "FC ERROR" since you have tried to POKE a data value larger than 255. An FC error is an out-of-range error. Now enter the following instruction into your computer:

```
POKE 4000, -1 ENTER
```

Again, the computer will display "FC ERROR." Why?

Now, enter the following PEEK instruction into your computer:

```
A = PEEK (65536) ENTER
```

The computer again displays "FC ERROR" since you have tried to PEEK at an address which is outside of the computer address map.

You can POKE a data value in any read/write memory location. However, you cannot POKE into a read-only memory location. For example, enter the following program into your computer:

```
10 POKE 4000,75
20 A = PEEK (4000)
30 PRINT A
```

After you run the program, the computer will display the value 75. This is because the POKE instruction *writes* the value 75 into memory location 4000. This is a read/write memory location and, thus, you were able to change its contents with the POKE instruction. The PEEK instruction simply *reads* the contents of memory location 4000. Notice that a variable must be assigned to the PEEK instruction in order to display the memory location contents. We have chosen the variable A in this example. Now, enter the following program into your computer:

```
10 POKE 45000,75
20 A = PEEK (45000)
30 PRINT A
```

After you run the program, the computer will display the value 38. Why didn't it display the value 75, which was POKEd into memory location 45000? Address 45000 is a read-only memory location and you cannot change its value with a POKE instruction. Consequently, the PEEK instruction returned the *fixed* contents of the ROM location.

You will use the PEEK and POKE instructions quite often for your interfacing software, so let's see how some of the interface lines react to the execution of these two instructions. A *timing diagram* for the 6809E clock, data, address, and R/W lines is shown in Fig. 2-7. A timing diagram simply illustrates the high/low activity on one or more signal lines during the execution of a specific operation. This timing diagram illustrates the activity on these lines during the execution of the PEEK instruction. The PEEK instruction is an input, or read, operation for the microprocessor. Consequently, the 6809E R/W line goes high at the beginning of the E clock cycle to indicate a read operation.

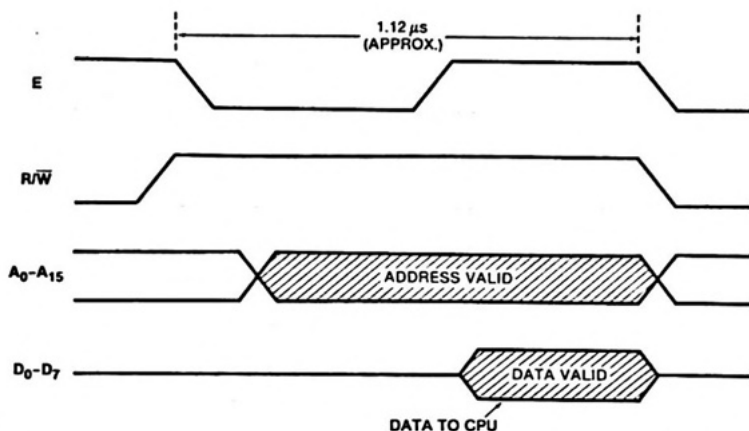


Figure 2-7
Timing diagram for the PEEK instruction.

After about $\frac{1}{4}$ clock cycle, the 6809E places the address to be PEEKed on the address bus. This is the address contained in the PEEK instruction. Now, at this point the external memory or peripheral device that is being accessed "knows" two things:

1. That the CPU is performing a read operation and data is to be transferred to the processor.
2. The specific address from which data is to be read.

The selected or "addressed" memory or peripheral device must then respond by transferring its 8 bits of data to the 8 data bus signal lines, D₀-D₇, before the end of the current clock cycle. The data transfer on data lines D₀-D₇ is represented by the data-valid portion of the timing diagram.

A timing diagram for the POKE operation is shown in Fig. 2-8. Here, the R/W line goes low at the beginning of the E clock cycle to indicate a write operation. Then, as for the PEEK operation, the address is placed on the address bus. At this point, the external memory or peripheral device that is being accessed "knows" that the CPU is performing a write operation, and the specific address that is to be written into. The CPU then places the data value contained in the POKE instruction on data bus lines D₀-D₇.

The Color Computer has a number of instructions that are used to control peripheral I/O devices. These instructions are not as general as PEEK and POKE since they are used to control specific peripheral devices or to perform specific functions. Two common peripheral I/O

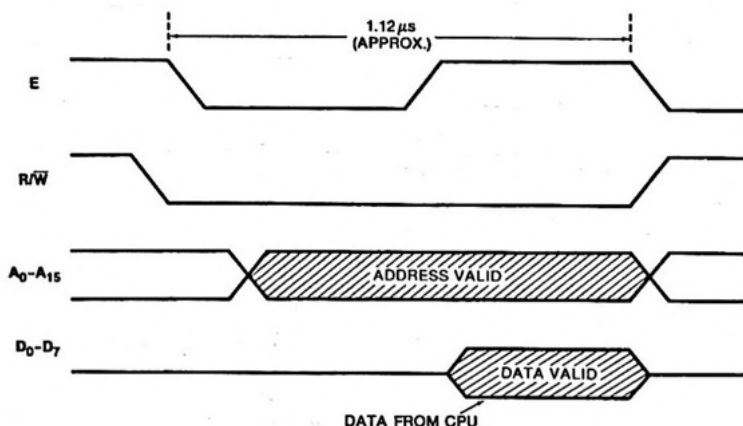


Figure 2-8
Timing diagram for the POKE instruction.

instructions are INPUT and PRINT. The INPUT instruction causes a BASIC program to stop and wait for an input from the keyboard. The PRINT instruction causes a string of characters to be printed on the TV monitor.

The Color Computer has several *graphic display* commands. Examples include SET, RESET, PSET, COLOR, CIRCLE, LINE, among others. All of these instructions are output instructions which control the TV display. Other I/O instructions are CLOAD, CSAVE, LPRINT, LIST, CLS, among others. You are familiar with the use of most of these instructions, so we will not discuss them. The main idea is that they are all I/O instructions that cause information to be read from or written to peripheral I/O devices which have specific address assignments.

SUMMARY

In this chapter, you were acquainted with the 6809E microprocessor—its evolution, internal structure, instruction set, and interface lines. You know how these things relate to the Color Computer. In addition, you now know what an interrupt is, and are aware of the external interrupt capabilities of the Color Computer. Finally, you saw how the 6809E data, address, and R/W signal lines react to the execution of I/O instructions, in particular the PEEK and POKE instructions. Now, you are ready to apply this knowledge and learn the fundamental concepts of microcomputer interfacing. Then, you will be ready to do some practical interfacing to your Color Computer.

3 Microcomputer Interfacing Fundamentals

Now that you have been acquainted with the I/O features of the 6809E and the TRS-80® Color Computer, you are ready to learn the three general fundamentals of microcomputer interfacing—*decoding*, *3-state buffering*, and *latching*. A good understanding of these three fundamental interfacing concepts is all that is required to perform simple interfacing tasks with the Color Computer, or any computer for that matter. We will show you how these three fundamental interfacing functions can be performed using digital logic design techniques. Consequently, we have assumed that you can “read” and interpret a logic circuit diagram, and that you are familiar with the more common 7400-series transistor-transistor logic, or TTL, digital devices. Consult Appendix B if you need to brush up on the basic logic gate symbols and their corresponding truth tables.

We do not intend to spend a great deal of time discussing digital logic design techniques for providing decoding, 3-state buffering, and latching, since most of these fundamental interfacing functions can be performed by using inexpensive LSI *programmable peripheral interface devices*. These devices are so inexpensive that it would cost more in hardware to replace their functions with conventional TTL digital logic devices. One such device is the *6821 peripheral interface adapter*, or *PIA*. The PIA is a *smart* device, meaning that it is programmable and, therefore, very flexible. The PIA is relatively easy to use and eliminates a great deal of interfacing aggravation. We will discuss the PIA in this chapter and use it extensively for the Color Computer interfacing experiments in Chapter 6.

OBJECTIVES

At the end of this chapter, you should be able to:

- Explain the functional requirements of an input port.
- Explain the functional requirements of an output port.
- Design a simple address decoder using NAND/NOR TTL logic devices.
- Design an address decoder using a 74LS154 decoder chip.
- Describe the I/O features of the 6821 peripheral interface adapter, or PIA.
- Describe the internal registers in the PIA.
- Write a BASIC program to initialize the PIA.
- Write a BASIC program to input and output data via a PIA interface.

GENERAL I/O CONCEPTS

We will use the simplest input device, a switch, to illustrate data input, and the simplest output device, a single LED, to illustrate data output. As you will discover later, many sophisticated peripheral I/O devices are simply a collection of "switches" and output "indicators" such as LEDs. An input interface is called an *input port* and an output interface is called an *output port*.

The circuit in Fig. 3-1 illustrates an input interface between four switches and the CPU. Here, the four switches are being used to feed four bits of binary data into the system. The switches are connected to a +5-volt dc supply through *pull-up resistors*. An open switch will cause a logic 1 to be present, while a depressed switch that makes contact with ground will cause a logic 0 to be present. The input port consists of two functional parts, as shown in Fig. 3-1—an *address decoder* and a series of *3-state buffers*. The address decoder provides an interface for the address bus, and the 3-state buffers provide an input interface for the data bus.

Recall that in a memory-mapped I/O system, peripheral I/O devices must be assigned to an address or group of addresses within the address space. In Fig. 3-1, the four binary switches are assigned to a unique address. We have chosen decimal address 50000 for this example. Thus, the CPU must *read* the input switch data by placing address 50000 on its address bus and performing a read operation. In addition, this address must be recognized as the address of the switches by external circuitry. This is the job of the address decoder.

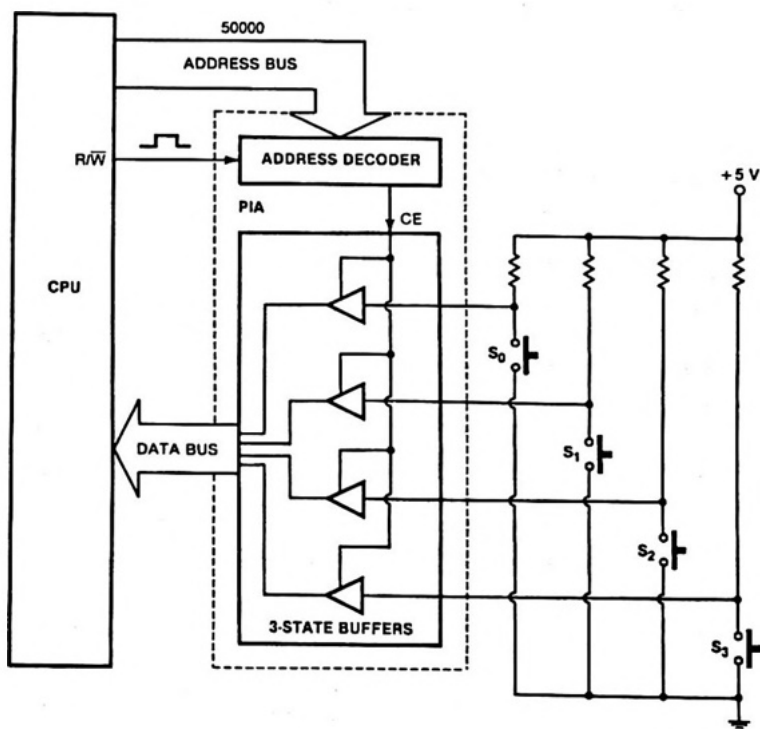


Figure 3-1 An input interface, or port, consists of an address decoder and 3-state buffers.

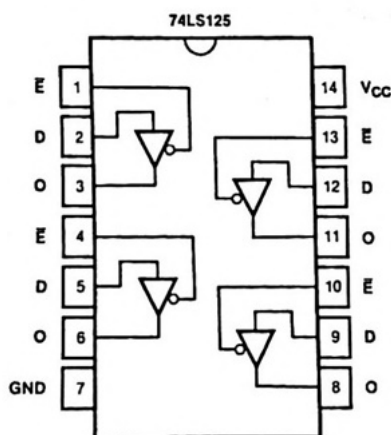
The address decoder is connected directly to the address bus. When the proper address is placed on the bus, the address decoder recognizes that address. It will ignore all other addresses. An address decoder can be a series of logic gates configured to recognize a particular address, or it can be a decoder chip that makes the logic design less complicated. There will be more about this later.

In the case of an *input* operation, the address decoder must enable the 3-state buffers when it recognizes the proper address on the address bus. The 3-state buffers will pass information from their inputs to their outputs only when they are enabled. The input devices (switches in this case) are connected to the buffer inputs and the buffer outputs are connected to the microprocessor data bus. When the 3-state buffers are not enabled, they represent an almost infinite impedance, or resistance, between the data bus and the switches.

This type of buffering is required to effectively isolate the switches from the data bus when they are not being addressed. Three-state buffering for input ports is a *must* when working with computer bus systems.

Notice that the address decoder output in Fig. 3-1 is connected directly to the enabling line (CE) of the 3-state buffers. Consequently, when the decoder recognizes the proper address (50000) on the address bus, it will enable the 3-state buffers such that the input switch data is passed to the CPU via the data bus. Here, only four lines of the 8-bit data bus are required, since only four switches are being interfaced. However, up to eight switches can be interfaced to the CPU data bus and assigned to one unique address using this technique. Each switch must be connected to a single data bus line via a 3-state buffer. Once the switch data is read, the on/off switch data is represented by the 1 or 0 status of the bits within the 8-bit data byte.

A typical 3-state buffer device is shown in Fig. 3-2. This device, a 74LS125, is a *quad* 3-state buffer, meaning that four separate buffers are provided within the chip package. Thus, two 74LS125 chips would be required to interface to the entire 8-bit data bus of the CPU. Notice that each buffer is enabled with a low-enable ($\bar{E}$) signal. A high-enable version, the 74LS126, is also available. There are 8-bit, or *octal*, 3-state buffers available such as the 74LS240, 74LS241, and 74LS244 devices shown in Fig. 3-3. The 74LS240 is an inverting 3-state buffer, meaning that its output logic is the 1s complement, or opposite, of its



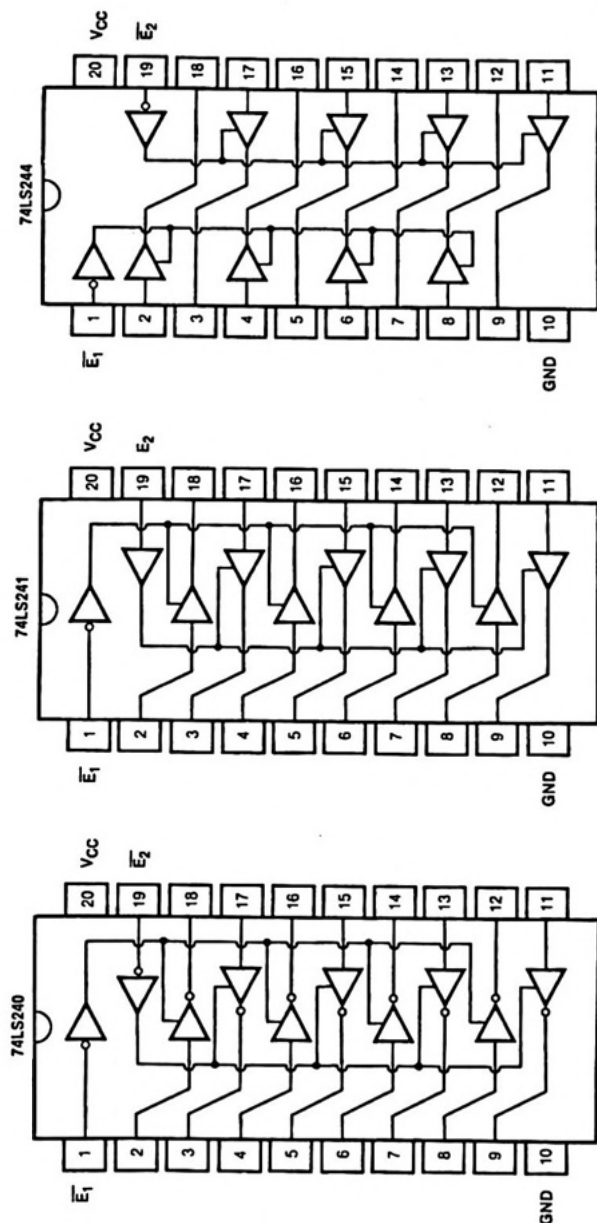
(A) Logic and connection diagram.

74LS125		
Inputs		Output
$\bar{E}$	D	
L	L	L
L	H	H
H	X	Z

H = High (logic 1)
 L = Low (logic 0)
 X = Don't Care (1 or 0)
 Z = High Impedance

(B) Truth table.

Figure 3-2
 The 74LS125 quad 3-state buffer.



74LS244		
Inputs		Output
$\overline{E_1}, \overline{E_2}$	D	
L	L	L
L	H	H
H	X	(Z)
H	X	(Z)

74LS241		
Inputs		Output
$\overline{E_1}$	E_2 , D	
L	L	L
L	H	H
H	X	(Z)
H	X	(Z)

H = High Voltage Level
 L = Low Voltage Level
 X = Don't Care
 Z = High Impedance

74LS240		
Inputs		Output
$\overline{E_1}, \overline{E_2}$	D	
L	L	L
L	H	H
H	X	(Z)
H	X	(Z)

(A) Logic and connection diagrams.

(B) Truth tables.

Figure 3-3
 The 74LS240, 74LS241, and 74LS244 octal 3-state buffers.

input logic. The 74LS241 and 74LS244 are noninverting 3-state buffers. Notice also from Fig. 3-3 that each device differs in the operation of its enabling logic.

The circuit in Fig. 3-4 illustrates an output interface between four LEDs and the CPU. The LEDs are connected in a common-anode configuration, meaning that a logic 0 output will cause the LED to illuminate. The output port consists of two functional parts—an *address decoder* and a *latch*. As before, the address decoder provides an interface for the address bus. The latch provides an output interface for the data bus. We have assigned the four LEDs to address 50002 for example purposes. Consequently, the CPU must *write* data to address 50002 to illuminate the LEDs. The address decoder in Fig. 3-4 must be designed to recognize address 50002 and ignore all other addresses.

When the address decoder recognizes address 50002, it will enable the latch. Simultaneously, the CPU places output information on the data bus. The latch will store the data bus information, causing the on/off pattern of the LEDs to match that of the latched information from the computer.

A latch is required for an output port since data is only present on the data bus for a very short period of time (less than 1 microsecond for the 6809E). However, once the data has been latched, it appears at the latch outputs and, hence, the LEDs until new information is output to the latch. Again, we are only using four lines of the 8-bit data bus in Fig. 3-3. Eight LEDs could be connected to the data bus at one unique address using all eight data bus lines and an 8-bit latch.

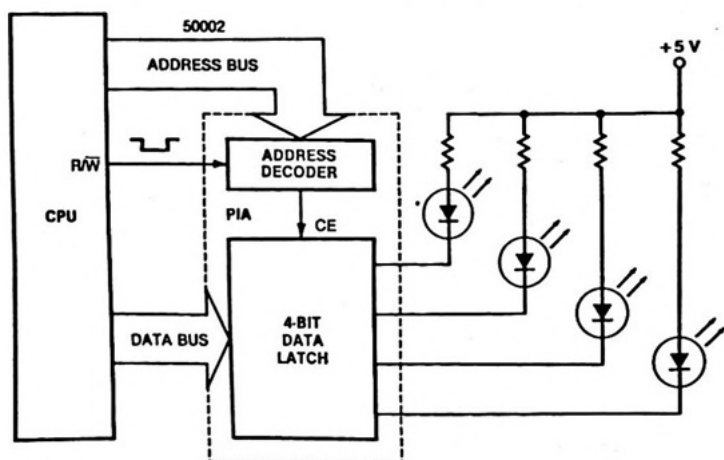
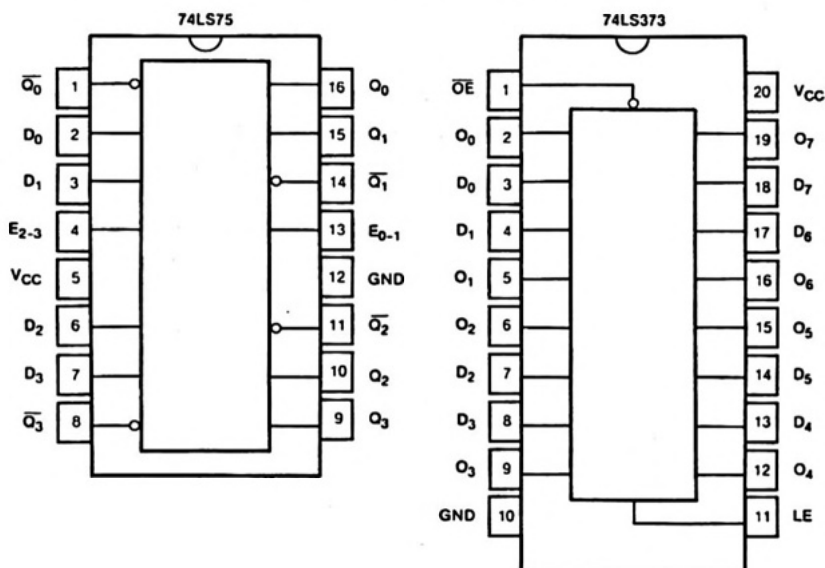


Figure 3-4
An output interface, or port, consists of an address decoder and a latch.

There are many different types of TTL latch integrated circuits. Two common devices are the 74LS75 and 74LS373 latches shown in Fig. 3-5. The 74LS75 is a 4-bit latch, and its two enabling lines must be tied together to provide 4-bit operation. Two 74LS75 devices would be required to create an 8-bit latch. You can obtain both inverted and noninverted outputs from the 74LS75.

The 74LS373 is an 8-bit latch, so only one device is required for an 8-bit output port. Only noninverting outputs are provided by this latch. There are two enable lines—latch enable (LE) and output enable (OE). Your address decoder output must be connected to the latch



(A) Connection diagrams.

74LS75			
Inputs		Outputs	
D	E	Q	$\overline{Q}$
L	H	L	H
H	H	H	L
X	L	X	X

74LS373			
Inputs			Output
$\overline{OE}$	LE	D	Q
L	H	H	H
L	H	L	L
L	L	X	X
L	X	X	X

H = High (logic 1)
 L = Low (logic 0)
 X = Don't Care (1 or 0)
 Z = High Impedance

(B) Truth tables.

Figure 3-5 The 74LS75 4-bit latch and the 74LS373 8-bit latch.

enable (LE) line. The output enable (OE) line provides 3-state output control for the latch. However, for most microprocessor output ports, this line is tied to ground.

In summary, you have seen that the two main ingredients of an input port are address decoding and 3-state buffering. An output port requires address decoding and latching. Now, let's combine these ideas and look at an input/output interface, or *I/O port*. An I/O device is shown interfaced to the CPU in Fig. 3-6. Here, the address decoder is common to both operations, input and output. However, it must determine when an input, or read, operation is being performed versus an output, or write, operation. To do this, the address decoder must decode the CPU $R/\overline{W}$ line along with the address bus lines to select a read or a write device at that address. Again, one address is assigned to the I/O device. However, during an input, or read, operation, the $R/\overline{W}$ line will be high (logic 1), and the address decoder must enable the 3-state buffers. During an output, or write, operation the $R/\overline{W}$ line will be low (logic 0) and the address decoder must enable the latch. The $R/\overline{W}$ line must be involved in the device selection process so that the proper flow of information on the data bus is preserved.

By this time you have probably noticed that the interface in Figs.

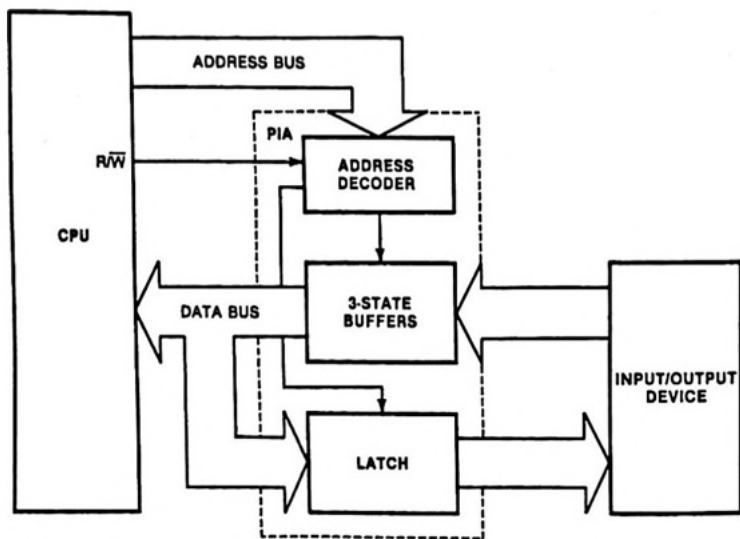


Figure 3-6

An input/output interface consists of an address decoder, 3-state buffers, and a latch.

3-1, 3-4, and 3-6 is contained in a dotted box labeled PIA. We have done this to show you that the peripheral interface adapter, or PIA, is capable of performing all three interfacing functions in part—decoding, 3-state buffering, and latching. We will discuss the PIA shortly. However, before we do, let's take a closer look at address decoding.

DECODING

In Figs. 3-1, 3-4, and 3-6 you saw the use of an address decoder. The address decoder is necessary to recognize the proper address from the address bus and then to enable the I/O circuitry for data transfer. An address decoder can be a series of logic gates configured to recognize a particular address, or it can be a series of decoder chips that make the logic design less complicated.

Using Logic Gates for Address Decoding

Suppose, as in Fig. 3-1, that your I/O device is assigned to address 50000. Your decoder circuit must be capable of detecting this address and then enabling the 3-state buffers. A typical TTL logic circuit that could perform this function is shown in Fig. 3-7. The *decoding chart* at the bottom of Fig. 3-7 shows the necessary address line binary bit

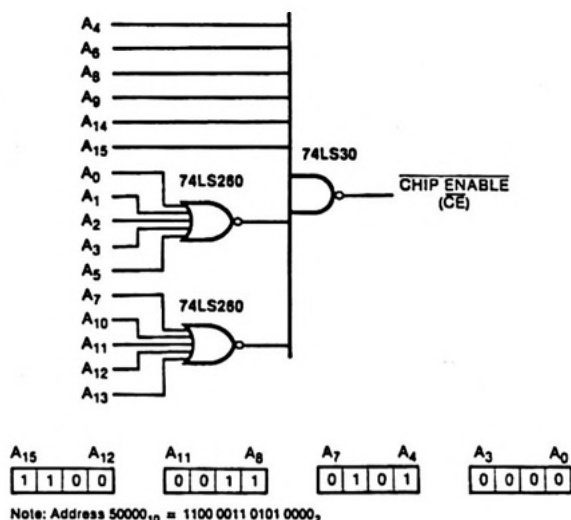


Figure 3-7
A full, or absolute, address decoder circuit to decode address 50000.

status for address 50000. Here, we are decoding *all* 16 of the address bus lines, such that the *only* address that will cause a chip enable (CE) is address 50000. When all of the address bus lines are decoded as in Fig. 3-7, we say that the address is *fully*, or *absolutely*, *decoded*.

You will notice from Fig. 3-7 that all the logic 0 address lines are NORED, since an all logic 0 input is the *unique* state of a NOR gate. This means that the only time the output of a NOR gate is high (logic 1) is when all of its inputs are low (logic 0). You will also notice from Fig. 3-7 that all logic 1 address lines are NANGED, since an all logic 1 input is the unique state of a NAND gate. Thus, to decode any address you can simply NOR the logic 0 bits of that address and NAND the logic 1 bits of the address. The NOR gate outputs are NANGED along with the logic 1 address lines, as shown in Fig. 3-7, to provide the final active-low chip enable output from the decoder. By tracing the logic through the circuit in Fig. 3-7, you will find that the chip enable output is low for one unique address—50000.

Another way to decode the same address is shown in Fig. 3-8. Here, we are decoding only part of the address bus. The lower eight address bus lines (A0–A7) are not being decoded. This type of decoding is called *partial*, or *nonabsolute decoding*. The advantage of partial decoding is that fewer logic gates are required. However, the disadvantage is that several addresses will cause an enable condition. In Fig. 3-8, a chip enable signal will be generated for addresses 49920 through 50175. Thus, 256 addresses within the address map must be assigned to this decoder. This is the price you must pay to reduce the required decoder hardware. However, since you have 64K addresses to use, it will probably not pose a problem to allocate 256 addresses to one I/O device. In the address decoding chart of Fig. 3-8, we show

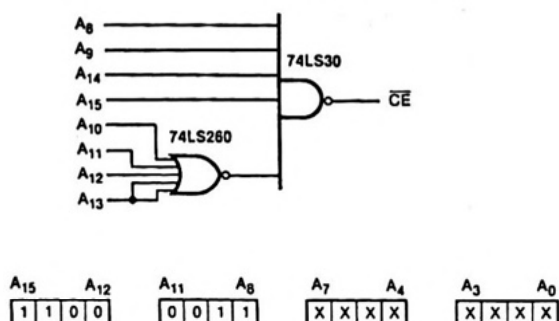


Figure 3-8 A partial, or nonabsolute, address decoder circuit to decode address 50000.

the lower half of the address bus lines as Xs, indicating a *don't care* state (either logic 1 or logic 0), since they are not being decoded by the address decoder.

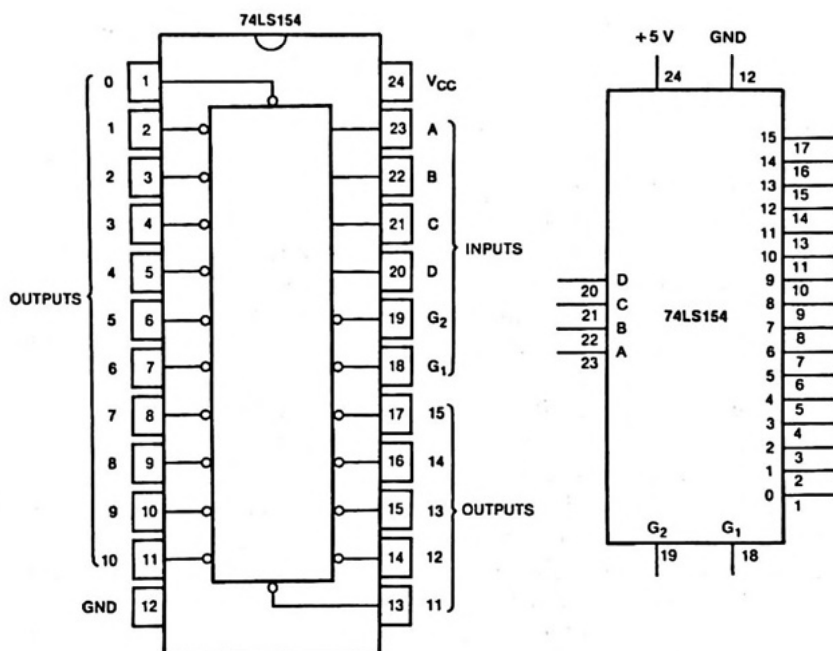
Another way to provide decoding is by the use of decoding chips. One very popular device is the 74LS154 decoder shown in Fig. 3-9. The outputs are lines 0 through 15, shown inside the block, while the input lines are lines A, B, C, and D. This type of decoder is called a *1-of-16 decoder*, since only one of the output lines is active at a logic 0 state for any binary representation of decimal values 0 through 15 on the input. Note from the truth table in Fig. 3-9 that there are also two enable lines labeled G1 and G2. These lines must be at a logic 0 state to provide an active output from the 74LS154.

The circuit in Fig. 3-10 illustrates how two 74LS154 decoder chips might be used to partially decode address 50000. By observing the address line decoding chart in Fig. 3-10 and the 74LS154 truth table in Fig. 3-9, you can determine that lines 12 and 3 are active when address 50000 is present on the address bus. These two lines are then NORED to provide an active-high chip enable output from the decoder circuit. You will also notice that the circuit in Fig. 3-10 does not decode the lower four address lines (A0–A3). Consequently, any address within the range of 50000 to 50015 will cause an active chip enable output.

The advantage of using decoder chips such as the 74LS154 is design simplicity and flexibility. The decoder circuit in Fig. 3-10 could also be used to decode other addresses by simply using different 74LS154 output lines. For example, addresses 18256 through 18271 could be decoded by using output line 4 from the top decoder chip and output line 7 from the bottom decoder chip in Fig. 3-10. These two lines must then be NORED together to provide a chip enable signal for this group of addresses.

There are additional decoder chips that might be useful to you when interfacing your Color Computer. These include the 74LS42 1-of-10 decoder, 74LS138 1-of-8 decoder, and 74LS139 1-of-4 decoder, among others. We have shown these devices along with their respective truth tables in Figs. 3-11 through 3-13.

Recall earlier we stated that there are programmable peripheral interface chips that provide some decoding, 3-state buffering, and latching in one LSI chip package. One such device is the Motorola 6821 peripheral interface adapter, or PIA. There are other devices made by different manufacturers that perform the same functions. However, the 6821 PIA is a member of the 6800/6809 "family." Con-



(A) Connection and logic diagrams.

Inputs						Outputs															
G ₁	G ₂	D	C	B	A	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	0	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	0	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	1	0	0	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
0	0	0	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1
0	0	0	1	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1
0	0	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1
0	0	1	0	0	0	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1
0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1
0	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1
0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1
0	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1
0	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
0	0	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
0	1	X	X	X	X	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	X	X	X	X	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	X	X	X	X	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

(B) Truth table.

Figure 3-9 The 74LS154 1-of-16 decoder.

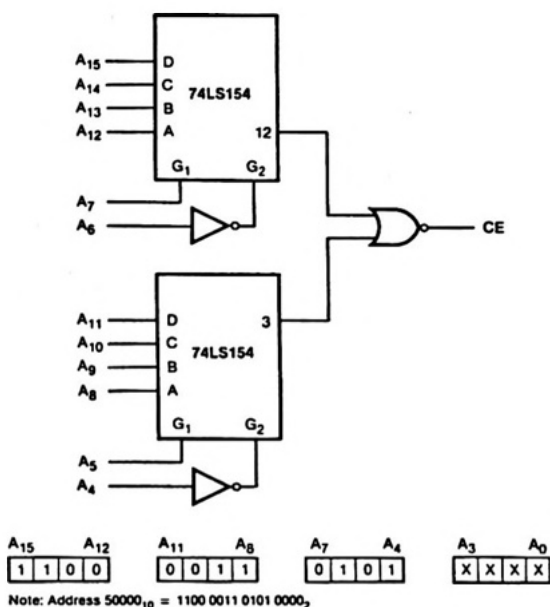


Figure 3-10

Partial decoding of address 50000 using two 74LS154 decoder chips.

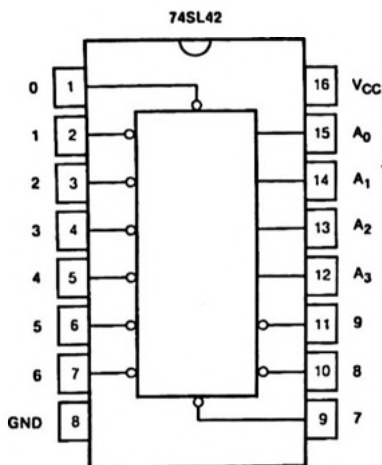
sequently, it is directly compatible with your Color Computer circuit. The 6821 PIA is the topic of our next discussion, since you will make extensive use of this device when interfacing to the Color Computer.

THE 6821 PIA

In this section, we will provide you with an introductory functional description of the PIA. Then, in Chapter 4, you will learn how to connect the PIA to the CPU bus lines available at the Color Computer parallel expansion connector port.

Functional Description of the PIA

A functional diagram of the PIA is provided in Fig. 3-14. First, you see that the PIA can be looked at functionally as having two sides, a CPU side and a peripheral side. The CPU side includes the data, address, and control lines that will interface to the CPU data, address, and control buses. The peripheral side contains two I/O ports labeled port A and port B. Each port is an 8-bit port, meaning that each port



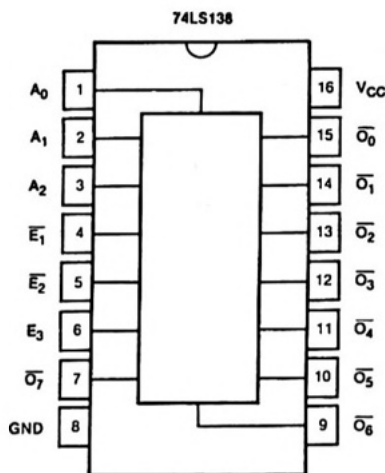
(A) Connection diagram.

[illegible]

H = High Voltage Level
L = Low Voltage Level

(B) Truth table.

Figure 3-11 _____
The 74LS42 1-of-10 decoder.



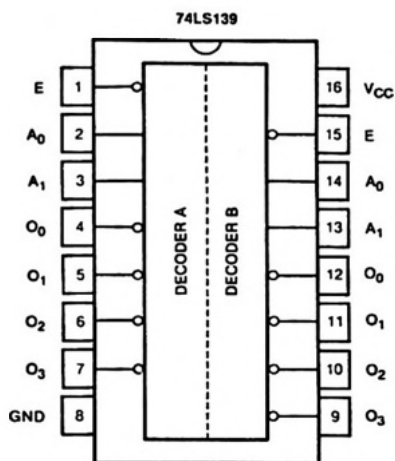
(A) Connection diagram.

[illegible]

H = High Voltage Level
L = Low Voltage Level
X = Don't Care

(B) Truth table.

Figure 3-12 _____
The 74LS138 1-of-8 decoder.



(A) Connection diagram.

Inputs			Outputs			
E	A ₀	A ₁	O ₀	O ₁	O ₂	O ₃
H	X	X	H	H	H	H
L	L	L	L	H	H	H
L	H	L	H	L	H	H
L	L	H	H	H	L	H
L	H	H	H	H	L	L

H = High Voltage Level
L = Low Voltage Level
X = Don't Care

(B) Truth table.

Figure 3-13 The 74LS139 1-of-4 decoder.

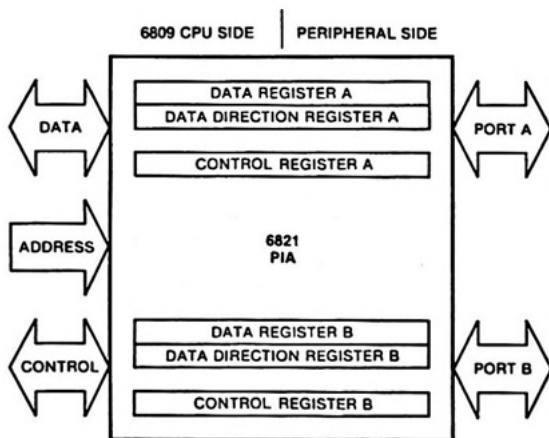


Figure 3-14 Functional diagram of the 6821 peripheral interface adapter, or PIA.

contains eight data lines. The eight data lines in a given port can be *configured* as all input, all output, or any combination of input/output. In any event, the PIA port lines are connected to one or more peripheral I/O devices. Thus, the PIA is *the* interface between the CPU and its peripheral I/O devices.

Internal Registers of the PIA

Internally, the PIA contains six 8-bit read/write registers, as shown in Fig. 3-14. Three registers are used with port A and three with port B. Each group of three registers contains a *data register* labeled DRA or DRB, a *data direction register* labeled DDRA or DDRB, and a *control register* labeled CRA or CRB.

The *data registers* (DRA and DRB) act as 3-state buffers for data input operations, and latches for data output operations. Each of the eight bits within a given data register is connected to one of the respective port data lines on the PIA chip. For example, the eight data lines of port A are labeled PA0 through PA7. Therefore, bit 0 of data register A is tied to PA0, bit 1 of data register A is tied to PA1, and so on. The same arrangement is used for the bits of the B data register (DRB), and the port B data lines (PB0–PB7). The relationship of the data register bits and the port data lines is illustrated in Fig. 3-15. As you will see shortly, each of the internal data registers is assigned to a unique address. To input information from a given port to the CPU, you simply perform a read operation using the address of that port's data register as the source for the information. To output information

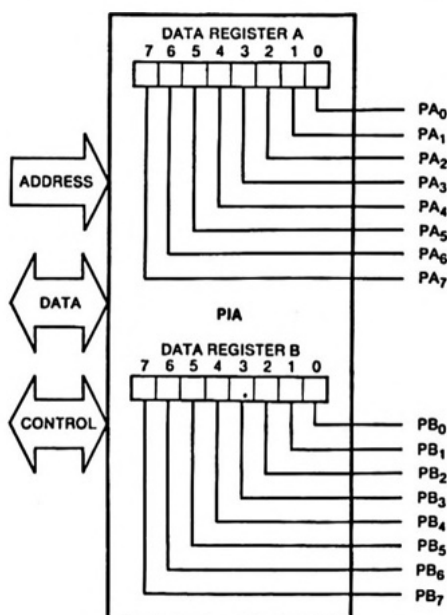


Figure 3-15 Relationship between the PIA data register bits and port I/O lines.

from the CPU to a given port, you simply perform a write operation to the address of that port's data register.

The port data lines must be *configured* for either input or output operations, not both. This is the job of the data direction registers. The *data direction registers*, labeled *DDRA* and *DDRB*, are each 8-bit registers which define the individual port lines as being used for either input or output operations. Each bit within a given data direction register configures its corresponding port data line. Here's how: A logic 1 in a given data direction register bit will cause its corresponding port line to be configured as an output line; a logic 0 in a given data direction register bit will cause its corresponding port line to be configured as an input line. To illustrate this idea, Fig. 3-16 shows port A configured as an 8-bit input port and port B configured as an 8-bit output port. Notice that the port A data direction register contains all 0s, and the port B data direction register contains all 1s.

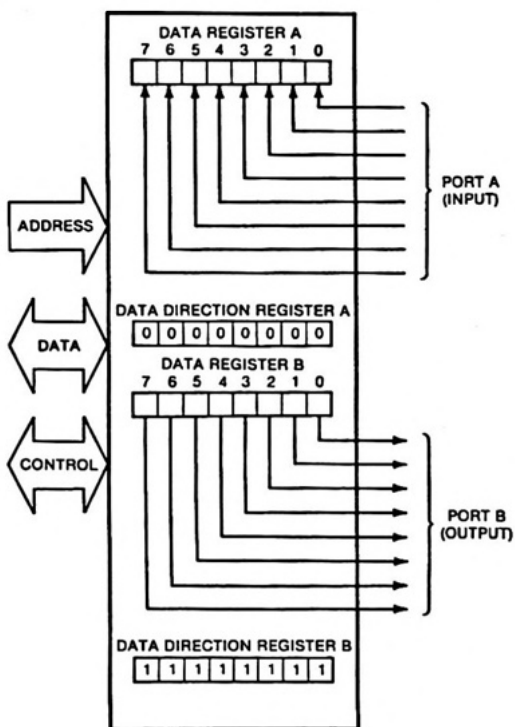


Figure 3-16 PIA port line configuration using the data direction registers (port A input; port B output).

Another PIA configuration is shown in Fig. 3-17. Here, port A is configured for input as before, but port B is configured for half input and half output by its data direction register. Each data direction register must have a specific address assigned to it. Thus, you can change the given port line configuration at any time by simply writing a different value to its respective data direction register.

The PIA control registers, labeled CRA and CRB, are 8-bit registers that are used for a variety of status and control functions. Control register A (CRA) and control register B (CRB) are identical in the roles they play for their respective ports. Each bit within a given control register has a specific function. Of particular interest at this time is bit 2 of each control register. Recall that we said that the data and data direction registers are assigned to specific addresses within the address map. Actually, the data register and data direction register of a

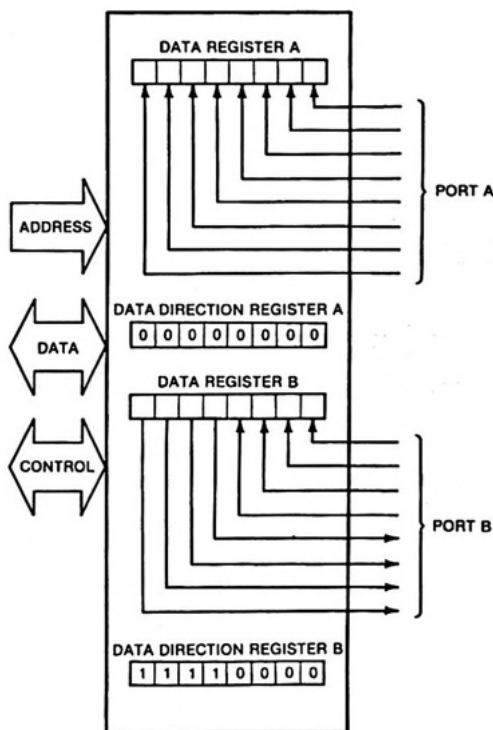


Figure 3-17 PIA port line configuration using the data direction registers (port A input; port B input and output).

given port are assigned to *one* address. Bit 2 of the respective control register determines whether the data register or the data direction register is selected when the assigned address is present on the address bus. When bit 2 is cleared (logic 0), the data direction register is selected. When bit 2 is set (logic 1), the data register is selected. This process is illustrated in Fig. 3-18.

The remaining control register bits are used to control I/O operations via four control lines that are available on the PIA. We will show you how to control I/O operations with the PIA control lines in the next chapter.

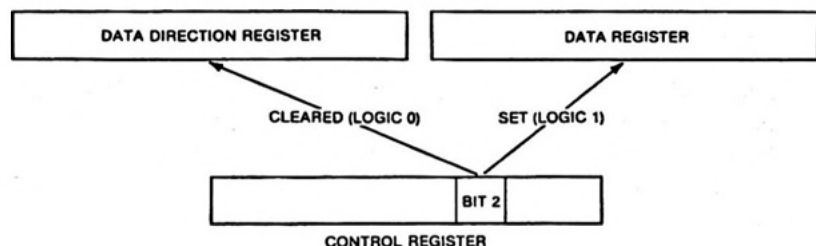


Figure 3-18 Register selection within the PIA.

PIA Initialization

Prior to using the PIA for data transfer, you must *initialize*, or *configure*, it by defining the port lines as either input or output lines. As you saw earlier in this section, this is accomplished by the 1s and 0s placed in the bits of each data direction register (DDRA and DDRB). Recall that if a logic 1 existed in a data direction register bit, its corresponding port line is configured as an output data line, while if a logic 0 existed, its port line is configured as an input data line. To initialize the PIA, you must execute an *initialization program* that will write a binary number in each data direction register and, thus, configure each port. The initialization procedure requires three major program steps:

1. Clear bit 2 of the control register(s) to access the data direction register(s).
2. Write values in the respective data direction register(s) to configure the port(s) as desired.
3. Set bit 2 of the control register(s) to access the data register(s) for subsequent data I/O transfer operations.

In this procedure, step 1 clears bit 2 of the control register(s) so that the data *direction* register is selected rather than the data register. The port(s) can then be configured by writing a value to the data direction register(s). After each port has been configured, bit 2 of the respective control register must be set so that the data register(s), and not the data direction register(s), is selected for subsequent data transfer operations through the port(s).

Now, let's see how this can be done with a BASIC program. As you will discover in the next chapter, the PIA requires four address assignments. Suppose, for example, that a PIA is assigned to the four addresses 65344 through 65347. This "peculiar" address assignment will make more sense when we show you how to interface to the Color Computer expansion connector in the next chapter. The PIA will be connected to the address bus such that its internal registers are assigned, or mapped, to this group of addresses as shown in Table 3-1. Now, suppose you wish to configure port A of the PIA as an 8-bit input port, and port B as an 8-bit output port. The following BASIC program will accomplish the required PIA initialization task:

```
10 POKE 65345, 00 } Step 1
20 POKE 65347, 00 }
```

```
30 POKE 65344, 00 } Step 2
40 POKE 65346, 255 }
```

```
50 POKE 65345, 4 } Step 3
60 POKE 65347, 4 }
```

Notice the extensive use of the POKE instruction in this program. Lines 10 and 20 will clear bit 2 of both control registers to select the data direction registers (Step 1). Port A is then configured for input by writing the value 00 in its data direction register located at address 65344 (line 30). Port B is configured for output by writing the value 255 to its data direction register located at address 65346 (line 40). Finally, bit 2 of both control registers is set to select the port data registers for subsequent data transfer operations (lines 50 and 60).

Once the PIA has been initialized, data I/O is simple. To input data from port A, you simply PEEK the port A data register (DRA) located at address 65344. To output data to port B, you must POKE a data value to the port B data register (DRB) located at address 65346. Thus, the command *PEEK (65344)* will input data from port A, and the command *POKE <data value>, 65346* will output a data value to port B.

Table 3-1. PIA Register Address Assignment for an Example PIA Initialization Program

Address	PIA Register Selected
65344	DDRA or DRA*
65345	CRA
65346	DDRB or DRB*
65347	CRB

**Depends on bit 2 of the respective control register.*

That's all there is to it! In the next chapter we will show you how to interface a PIA to the Color Computer to perform data input and output operations.

SUMMARY

In summary, this chapter acquainted you with the functional requirements of input and output interfaces, or ports. You discovered that an input port requires address decoding and 3-state buffering, while an output port requires address decoding and latching. Thus, input/output, or I/O, ports must include address decoding, 3-state buffering, and latching. The CPU $R/\overline{W}$ line must be decoded to determine the direction of data transfer on the data bus for I/O ports. Finally, you were acquainted with the I/O features and internal register structure of the 6821 PIA. You found that the PIA can be used in place of several digital devices to provide an I/O interface for the CPU. The PIA port lines must be configured by an initialization program. Once configured, data is transferred between the CPU and its peripheral I/O devices by simply reading and writing to the PIA data registers.

4 Expansion Connector Interfacing

In Chapter 1, you were introduced to the parallel expansion connector available on the right side of the TRS-80® Color Computer case. In this chapter, you will learn how to provide a parallel interface for your computer using a PIA connected to the expansion connector. We will discuss the PIA interface, then show you how to control parallel I/O operations with the PIA. Two control techniques will be presented—programmed I/O control, or polling, and interrupt I/O control.

OBJECTIVES

At the end of this chapter, you should be able to:

- Explain the function of the expansion connector signals.
- Design a PIA interface for the Color Computer.
- Explain how the PIA can be used to provide both programmed I/O, or polling, and interrupt control of parallel I/O operations.
- Explain what is meant by input and output handshaking.
- Write a BASIC program to configure the PIA for both programmed control and interrupt control of input and output handshaking operations.

EXPANSION CONNECTOR SIGNALS

The Color Computer expansion connector signals are listed again in Table 4-1 with their respective connector pin assignments. You should now be familiar with many of these signals as a result of your 6809E

Table 4-1. The Parallel Expansion Connector Pin Assignments

Pin No.	Signal Name	Description
1	-12 V	-12 volts (100 mA)
2	+12 V	+12 volts (300 mA)
3	HALT	Halt Input to the CPU
4	NMI	Nonmaskable Interrupt to the CPU
5	RESET	Main Reset and Power-Up Clear Signal to the System
6	E	Main CPU Clock (0.89 MHz)
7	Q	Quadrature Clock Signal which Leads E
8	CART	Interrupt Input for Cartridge Detection
9	+5 V	+5 volts (300 mA)
10	D0	Data Bit 0
11	D1	Data Bit 1
12	D2	Data Bit 2
13	D3	Data Bit 3
14	D4	Data Bit 4
15	D5	Data Bit 5
16	D6	Data Bit 6
17	D7	Data Bit 7
18	R/W	Read/Write Signal
19	A0	Address Bit 0
20	A1	Address Bit 1
21	A2	Address Bit 2
22	A3	Address Bit 3
23	A4	Address Bit 4
24	A5	Address Bit 5
25	A6	Address Bit 6
26	A7	Address Bit 7
27	A8	Address Bit 8
28	A9	Address Bit 9
29	A10	Address Bit 10
30	A11	Address Bit 11
31	A12	Address Bit 12
32	CTS	Cartridge Select Signal
33	GND	Signal Ground
34	GND	Signal Ground
35	SND	Sound Input
36	SCS	Spare Select Signal
37	A13	Address Bit 13
38	A14	Address Bit 14
39	A15	Address Bit 15
40	SLENB	Input to Disable Device Selection

I/O signal line knowledge. We can categorize the expansion connector signals as follows: *power/clock*, *data/address*, *address decode*, and *control*.

Power/Clock

Three dc power supply voltages are supplied on the expansion connector—+12 volts, -12 volts, and +5 volts. Notice the maximum current capabilities of each supply voltage as indicated in Table 4-1. These maximum current supply values cannot be exceeded without possible damage to the internal power supply of the Color Computer. You must, therefore, be careful when powering external devices from these supplies. There are two common ground pins provided on the expansion connector for system grounding. All devices interfaced to the Color Computer expansion connector must be grounded to one of these two pins.

You will notice from Table 4-1 that both the CPU E and Q clock signals are provided on the expansion connector. Recall that both of these signals have a frequency of 0.895 MHz. However, Q leads E by $\frac{1}{4}$ clock cycle, or 90°. You will use the E clock signal (pin 6) in all of your interfacing experiments. The Q clock signal will not be used.

Data/Address

You can see from Table 4-1 that all eight of the CPU bidirectional data lines (D0–D7) are provided on the expansion connector. Thus, the expansion connector is used for 8-bit *parallel* interfacing to the Color Computer.

The entire 16-bit address bus of the CPU is provided on the expansion connector. You can decode these address bus lines to access external devices interfaced to the computer via the expansion connector.

Address Decode

There are two address decode signals provided on the expansion connector. They are the *cartridge select*, or *CTS*, signal located at pin 32 and the *spare select*, or *SCS*, signal located at pin 36. The internal circuitry of the computer partially decodes two different groups of addresses to provide these two signals.

The cartridge select, or $\overline{CTS}$, signal is activated (low) whenever the CPU accesses any address within the range of 49152 through 65519. This block of 16,368 addresses is assigned to the external ROM cartridges within the Color Computer memory map (see Fig. 1-3). Thus, the $\overline{CTS}$ signal is used to select and enable the ROM devices within the

ROM cartridges during the execution of a ROM cartridge program. You can also use this select line to enable other external devices interfaced to the expansion connector.

The spare select, or $\overline{SCS}$, signal is activated (low) whenever the CPU accesses any address within the range of 65344 through 65375. This block of 32 addresses is *unused* by the computer. We will, therefore, use the $\overline{SCS}$ signal extensively to enable external devices when interfacing to the expansion connector. In many cases, you can use this signal alone to enable external devices and thus eliminate the need for external decoding circuitry.

Control

You are already familiar with the function of most of the expansion connector control signals as a result of learning about the 6809E I/O lines earlier. Notice from Table 4-1 that the 6809E $\overline{HALT}$, $\overline{NMI}$, and $\overline{RESET}$ control lines are provided on the expansion connector. In addition to these control lines, the expansion connector provides three system control lines. They are the *cartridge detection*, or *CART*, line located at pin 8, the *sound*, or *SND*, line located at pin 35, and the *internal device selection disable*, or *SLENB*, line located at pin 40.

The Color Computer uses the CART line to detect when a ROM cartridge is inserted into the expansion connector. The ROM cartridges manufactured by Radio Shack cause the Q clock signal at pin 7 to be shorted to the CART input line at pin 8. When this happens, an interrupt is generated to the CPU that causes the computer to jump to the program in the ROM cartridge which begins at address 49152. The 6809E interrupt used is the fast interrupt request, or $\overline{FIRQ}$. You can use this same technique to execute your own programs located in an external ROM that you have programmed and interfaced to the expansion connector.

The sound, or SND, line at pin 35 is an input line that is connected to the internal sound generating circuits of the computer. A digital signal input at this pin will cause a tone to be generated by your TV monitor. The exact tone depends on the frequency of the input signal. Radio Shack uses this input line to create sound effects for their ROM cartridge game programs.

The internal device selection disable, or SLENB, line at pin 40 is also an input line. When activated, or low, the internal device selection circuitry of the computer is disabled. This permits unused sections of the memory map to be used by external devices that are interfaced to the expansion connector. An external circuit might use this input line

to create a memory map for the computer that is different from the fixed map discussed in Chapter 1. Applications might include interfacing a direct memory access controller or a host computer to the computer.

THE PIA INTERFACE

In Chapter 2 you were acquainted with the functional capabilities of the 6821 PIA. You found that the PIA makes a convenient and flexible parallel interface since it performs the fundamental interfacing tasks of latching and 3-state buffering. Now, let's see how the PIA can be connected to the Color Computer expansion connector to provide parallel I/O operations.

The circuit in Fig. 4-1 shows how the 6821 PIA can be connected to the Color Computer expansion connector. First, notice that the PIA is powered by the +5-volt dc supply voltage available on the expansion connector. The CPU E clock signal on the expansion connector is connected directly to the E clock signal input on the PIA. This is the only clock signal required for the interface. Thus, the Q clock signal will not be used for the PIA interface.

For data transfer, the PIA has eight data lines labeled D0-D7. These eight PIA data lines are connected directly to the eight CPU data lines (D0-D7) available on the expansion connector. In addition, the PIA has an $R/\overline{W}$ line that must be connected to the expansion connector $R/\overline{W}$ line. Recall that this line is used to indicate the direction of data transfer on the data bus.

The PIA has three *chip-select* lines labeled CS0, CS1, and CS2 in Fig. 4-1. These three lines must be activated to enable the PIA for any data transfer operations. Normally, the address bus lines must be decoded to enable the PIA via these chip-select lines. This would be accomplished by placing an address decoder circuit between the address bus lines and the PIA chip-select lines. However, you can take advantage of the internal decoding provided by the Color Computer spare select, or $\overline{SCS}$, signal at pin 36 of the expansion connector. Recall that this output signal is active (low) whenever the computer accesses any of 32 addresses within the range of 65344 through 65375. We will, therefore, assign the PIA to this address range in the computer memory address map. To do this, you must connect the $\overline{SCS}$ line on the expansion connector to the $\overline{CS2}$ line on the PIA. The other two PIA chip-select lines (CS0 and CS1) must be permanently tied high to the +5-volt supply. The disadvantage of using the $\overline{SCS}$ line to enable the

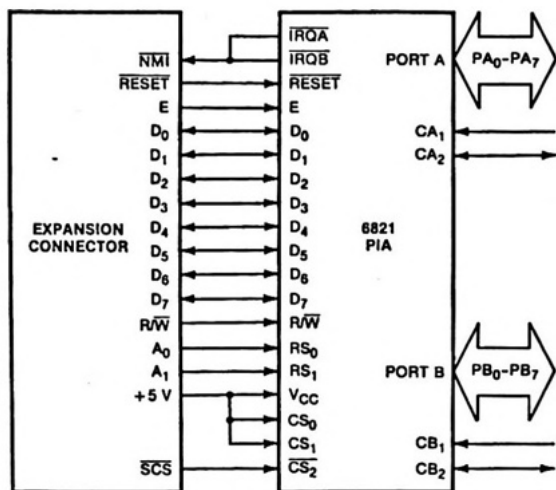


Figure 4-1
The PIA interface.

PIA is that you are assigning 32 addresses to the PIA. As you are aware, the PIA only requires four address assignments. However, this is a small price to pay to eliminate the extra decoding circuit that would be required without using the expansion connector $\overline{SCS}$ line.

In addition to the chip-select lines, the PIA has two register-select lines labeled RS0 and RS1 in Fig. 4-1. These two lines are used to select the registers within the PIA once it has been enabled via its chip-select lines. You must connect RS0 and RS1 to address lines A0 and A1, respectively, on the expansion connector. This will assign the PIA registers to four consecutive addresses as shown in Table 4-2. Notice that these are the first four addresses of the 32 addresses decoded by the Color Computer $\overline{SCS}$ line. Other addresses within this 32 address space can also be used to enable the PIA and access its internal registers. However, we will only use the first four (65344–65347). Notice also from Table 4-2 that address 65344 will select either data register A (DRA) or data direction register A (DDRA). In the same way, address 65346 will select either data register B (DRB) or data direction register B (DDRB). Recall that the exact register selected depends on the bit 2 status of the respective control register.

The PIA contains a $\overline{RESET}$ line that is connected directly to the $\overline{RESET}$ line available on the expansion connector. This allows the PIA to be reset, or reinitialized, whenever the reset button on the back of the computer case is depressed. Resetting the PIA will automatically *clear*

Table 4-2. The PIA Register Address Assignments Provided by the Expansion Connector Interface Circuit in Fig. 4-1

Address	PIA Register Selected
65344	DDRA or DRA*
65345	CRA
65346	DDRB or DRB*
65347	CRB

**Depends on the bit 2 status of the respective control register.*

all of its internal registers. This allows you to start from a "clean slate" when reinitializing the PIA.

Finally, there are two interrupt request lines available on the PIA, labeled $\overline{\text{IRQA}}$ and $\overline{\text{IRQB}}$. You will notice from Fig. 4-1 that we have wire-ORed these lines together and connected them to the nonmaskable interrupt, or NMI, input line on the expansion connector. This will allow an external peripheral device connected to the PIA ports to get the attention of the CPU. Recall that when the CPU receives a nonmaskable interrupt, it will automatically break away from what it is doing and service the interrupt by executing an interrupt service routine. The Color Computer is designed such that it will jump to RAM address 265 to be directed to the interrupt service routine whenever it receives a nonmaskable interrupt, or NMI, signal. This will be discussed in more detail shortly.

The remaining expansion connector lines are not required for the PIA interface. You will find that the PIA interface circuit in Fig. 4-1 will allow you to provide parallel I/O for several different computer interfacing applications. Devices that can be connected directly to the PIA interface in Fig. 4-1 include digital-to-analog converters, analog-to-digital converters, programmable timers, and parallel printers, among others.

Before we go any further, it will be helpful to discuss how the transfer of data is synchronized, or controlled, between the CPU and an I/O device. Again, you will see that the PIA is very useful for this purpose.

I/O DEVICE CONTROL

Data transfer between the CPU and an I/O device can be synchronized, or controlled, by one of two general control techniques—*programmed control* or *interrupt control*.

Programmed I/O Control

Program control is the simplest, but least efficient of the two techniques. With simple programmed I/O control, the computer is programmed to read data from input devices and write data to output devices at *scheduled* intervals. With the PIA interface discussed previously, this is done by simply reading and/or writing information to the PIA I/O ports. The data is read or written regardless of whether or not the I/O device needs to be serviced. The advantage here is simplicity; however, this method is not very efficient since many times the CPU is tied up performing unnecessary read/write operations.

A more efficient form of programmed I/O is called *polling*. Polling involves the use of a *status flag*. A status flag is usually a single read/write bit location that is activated by an I/O device. The flag is set or cleared when the I/O device needs to be serviced. The CPU must periodically *poll*, or read, the flag status of each I/O device to determine if service is needed. If the status flag of a given I/O device has been activated, the CPU will enter into a service routine for that device. If no service is needed, the CPU will poll the status flag of another device, and so on until each status flag in the system has been polled. With this method, each I/O device must be polled several times a second. This requires a small subroutine called a *polling loop*. Some efficiency is sacrificed since the CPU must periodically break away from what it is doing to execute the polling loop subroutine. Moreover, the I/O device must wait until the CPU gets around to checking its status flag before any data transfer can take place. This results in a relatively poor response time from the CPU when the I/O device must have instant service.

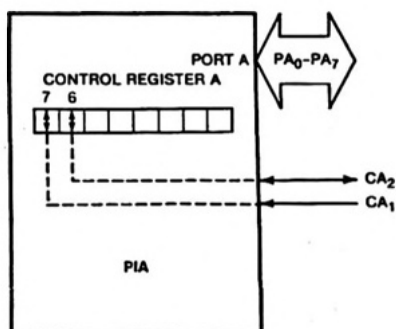
One advantage of simple programmed I/O and polling is that minimum interfacing hardware is required. In addition, you know exactly when the I/O device might be serviced since the I/O operation is scheduled into your program execution. You can schedule the I/O operations into your program and nothing will interfere with that schedule. Therefore, we say that programmed I/O, or polling, is *synchronous*, or synchronized with program execution.

The PIA interface discussed in the last section can be used to perform polling operations for I/O devices. Notice from Fig. 4-1 that each PIA port has two *control* lines. The control lines for port A are labeled CA1 and CA2. The port B control lines are labeled CB1 and CB2. These lines are used to provide both programmed and interrupt control of data transfers through their respective PIA port. We will discuss how I/O control is provided by the CA1 and CA2 control lines. The

port B control lines (CB1 and CB2) are used in the same way. Thus, anything we say about port A will also apply to port B, except where noted otherwise.

Recall that a programmed I/O polling operation requires that you check the status of a status flag. The PIA provides up to four status flags that can be used for polling operations. These flags are located in the two PIA control registers (CRA and CRB). Two status flags are located in the port A control register (CRA) and two are located in the port B control register (CRB). In a given control register, there is a status flag for each control line of that port. Thus, for port A, both the CA1 and CA2 control lines have an associated status flag in the port A control register. The CA1 status flag is bit 7 of the control register and the CA2 status flag is bit 6 of the control register as illustrated in Fig. 4-2. When control line CA1 is activated by an I/O device, bit 7 of the port A control register will set (logic 1). When CA2 is activated, bit 6 of the control register will set. The control lines can be activated by an I/O device which is connected to the PIA port lines and requires service from the CPU.

Figure 4-2 _____
The CA1 status flag is bit 7 of the port A control register and the CA2 status flag is bit 6 of the port A control register.



You must determine the flag status by polling the respective status flag. For example, the PIA interface in Fig. 4-1 assigns the port A control register to address 65345. To poll the CA1 status flag (bit 7) would require a simple BASIC polling subroutine as follows:

```

2050 A = PEEK (65345)
2060 IF A = 128 OR A > 128 THEN 3050
2070 RETURN
.
.
.
.
3050 (Beginning of I/O service routine)

```

This subroutine would be called periodically to check the CA1 status flag. Since bit 7 is the most significant bit of the control register, any value that is equal to or greater than 128_{10} indicates that bit 7 is set. If the flag is set as a result of an active CA1 control line, the CPU will jump to the beginning of the I/O service routine at line 3050. If the flag is cleared, the CPU will return to the main program.

The following subroutine can be used to check the CA2 flag status in bit 6 of the port A control register:

```
2050 A=PEEK (65345)
2060 IF A=64 THEN 3050
2070 IF A > 64 AND A < 128 THEN 3050
2080 RETURN
```

.

.

.

3050 (Beginning of I/O service routine)

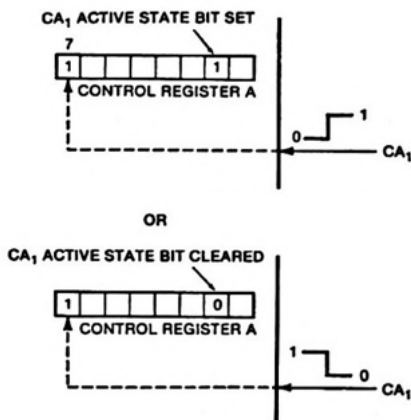
Here, an extra IF statement must be provided such that the CA2 status flag (bit 6) may be checked separate from the CA1 status flag (bit 7).

Now, you may be wondering how a status flag is cleared once it is set by an I/O device. The answer is simple—any read, or PEEK, operation on the port data register will clear the control register status flags. Thus, you must include a port read operation in your I/O service routine such that the status flag is cleared and made ready for a subsequent I/O operation.

The PIA provides an additional feature with respect to the activation of its control lines. You can define the *active state* of a given control line as being a low-to-high (0 to 1) transition or a high-to-low (1 to 0) transition. The active state is the control line activity that causes the status flag in the control register to set. The active state of the CA1 control line is defined by bit 1 of the port A control register. The active state of the CA2 control line is defined by bit 4 of the port A control register. If the *active state bit* is set (logic 1), a low-to-high transition on the respective control line will cause the status flag to set. If the active state bit is cleared (logic 0), a high-to-low transition on the respective control line will cause the status flag to set. This process is illustrated for the CA1 control line in Fig. 4-3. The active state for CA2 is defined in the same way, except that bit 3 is the CA2 active state bit. You must define the active states of the control lines when you initialize the PIA.

Sometimes it is desirable, or required, to generate an output signal to

Figure 4-3
The active state bit defines the active state transition (0 to 1 or 1 to 0) of the control line.



an I/O device which tells the device that the CPU has completed the requested I/O operation. The output signal allows the I/O device to prepare for another operation. The PIA can also be used to generate this output signal. You might have noticed from the PIA interface circuit in Fig. 4-1 that the CA2 and CB2 control lines can be either input or output control lines while CA1 and CB1 are *always* input control lines. You have already seen how the PIA control lines can be used for input. Now, let's see how CA2 and CB2 can be used as output control lines. We will confine our discussion to port A, keeping in mind that the operation of port B is the same, except where noted otherwise.

First, bit 5 of the port A control register defines the CA2 control line as being either an input or an output control line. When bit 5 of control register A is cleared (logic 0), the CA2 control line is defined as an input control line and can be activated by an I/O device for polling operations as discussed previously. When bit 5 of control register A is set (logic 1), the CA2 control line is defined as an output control line.

The CA2 and CB2 control lines are normally used as output lines when a *handshake* is required between the CPU and an I/O device. A handshake involves two-way control signal communication between the CPU and an I/O device as shown in Fig. 4-4. One communication line provides a signal from the I/O device to the CPU, which notifies the CPU that service is required. The second communication line provides a signal from the CPU to the I/O device, which notifies the I/O device that the CPU has completed the requested I/O operation. Consequently, you can use the CA1 line of the PIA to provide the input request signal and the CA2 line of the PIA to provide the output acknowledge signal.

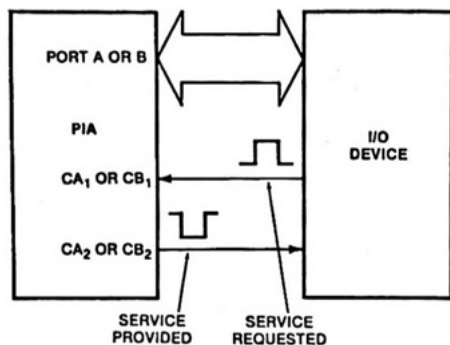


Figure 4-4
A handshake operation involves two-way control signal communication between the CPU and an I/O device.

We have shown an input device interfaced to the PIA in Fig. 4-5. The handshake operation illustrated here is called an *input handshake*. Here's how it works: The input device activates the CA₁ control line of the PIA, indicating that it has input data available at port A. The active CA₁ input line sets the bit 7 status flag in the port A control register. Thus, when the CPU polls the status flag, it will jump to the input device service routine. The service routine will read the port A data. When port A is read, the CA₂ output control line, which is *normally high*, will automatically go *low* for one CPU cycle (approximately 1 microsecond). This output signal completes the handshake and tells the input device that its data has been read by the CPU. The input device can then produce another byte of data and the foregoing process repeated until the entire input operation is complete. Notice

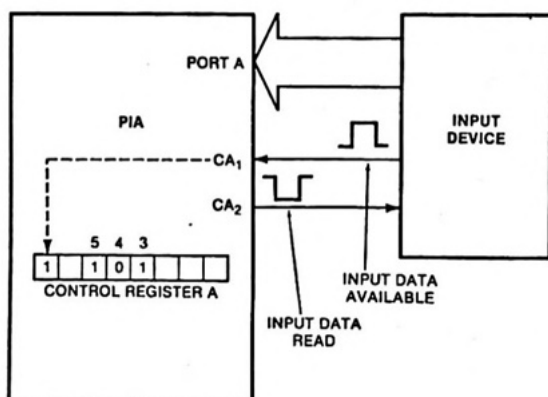


Figure 4-5
An input handshake via port A.

that one *complete* input handshake is provided for each byte of data transferred through port A.

To provide a handshake for an output device, you *must* use port B of the PIA as shown in Fig. 4-6. A handshake with an output device is called an *output handshake*. The handshaking process for port B is the same as that for port A except that the CB2 output signal is generated by a *write* operation. Consequently, port B must be used for output handshaking since the CB2 output line is only activated by a *write* to port B operation. In the same way, port A must only be used for input handshaking since the CA2 output line is only activated by a *read* port A operation. This is one of the major differences between ports A and B of the PIA.

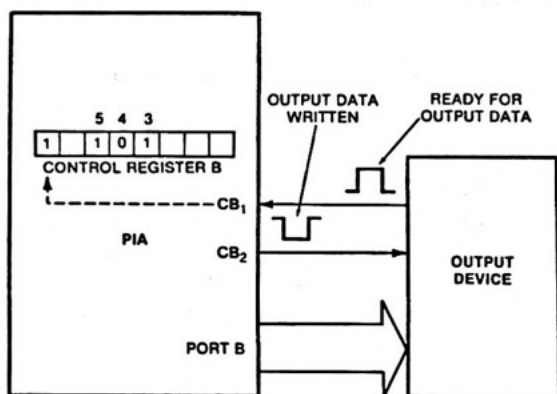


Figure 4-6
An output handshake via port B.

To provide the handshaking operations just discussed, the respective port control register must be configured as shown in Figs. 4-5 and 4-6. Notice that bits 5, 4, and 3 of the control register must contain 1 0 1, respectively. The respective control register(s) must be configured in this manner when you initialize the PIA.

Now, let's look at an example. Suppose you wish to configure port A of the PIA for input handshaking. The input device that is to be interfaced to port A will provide a high-to-low transition to get the attention of the CPU, and the computer will use a polling routine to determine when the input device requires service. The required control register bit configuration is shown in Fig. 4-7. Let's take a closer look at this configuration:

- Bit 0 is cleared to specify programmed I/O control of the data input operation via a polling routine. You will discover shortly that when bit 0 of the control register is set, interrupt control is specified.
- Bit 1 is cleared such that a *high-to-low* transition on the CA1 input line will set the bit 7 status flag.
- Bit 2 is set to access the data register, not the data direction register, during subsequent data input operations via port A.
- Bits 3, 4, and 5 are configured as 101 to specify input handshake control using CA2 as an output line.
- Bit 6 is cleared and not used since CA2 is being used as an output control line.
- Bit 7 is cleared since it is to be used as a status flag for the polling routine.

The decimal equivalent of the binary configuration shown in Fig. 4-7 is 44. Thus, the decimal value 44 must be written to the port A control register during the PIA initialization process. Now, if we assume that the PIA is interfaced to your computer as we discussed earlier (Fig. 4-1), the following simple one-line BASIC statement will configure the PIA for the input handshaking operation:

RESET

10 POKE 65345,44

How can this be? Easy—if you RESET the computer prior to the PIA initialization, all the internal PIA registers are cleared. Consequently, port A is automatically configured as an input port by the RESET op-

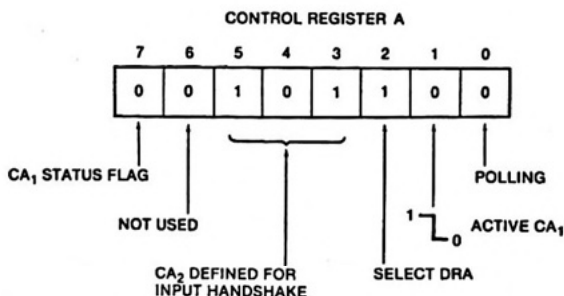


Figure 4-7

The port A control register bit configuration required for handshake control of an input operation.

eration. Then, all you have to do is write, or POKE, the port A control register value (44) to address 65345. Recall that this is the address assigned to the port A control register by the PIA interface circuit in Fig. 4-1. Once the PIA is initialized, the computer must periodically poll the CA1 status flag by executing a polling subroutine as we discussed previously.

We should mention that other configurations of the control register(s) are possible when using either CA2 or CB2 as output control lines. However, we will not discuss these configurations here, since they are not as widely used for handshake operations. You may wish to consult *How to Program and Interface the 6800*, available from Howard W. Sams, & Co., Inc., if you desire more information on the use of the PIA control lines.

Interrupt I/O Control

Interrupt I/O control is more efficient than programmed I/O control, or polling, since it provides fast response times for peripheral devices. However, one disadvantage to using interrupt control is that extra hardware and software are needed, especially if the interrupts are to be assigned priorities. Moreover, interrupts are *asynchronous* when compared to the execution of a program, meaning that they are not scheduled in your program. You, or the CPU, do not know exactly when they will take place or how much time the servicing might require.

Recall that an interrupt is used to get the attention of the CPU. If the interrupt is accepted (not masked), the CPU will break away from what it is doing and service the interrupt by executing an interrupt service routine. After the interrupt is serviced, the CPU will return to where it "left off" before the interrupt was generated, except during a RESET operation.

To use interrupt control, the interrupting device must activate one of the CPU interrupt input lines. This can be done with a direct connection to a CPU interrupt input line; or a peripheral interface device, like the PIA, can be used to indirectly generate the interrupt to the CPU. The latter method is used especially when handshaking is required between the CPU and peripheral device.

To generate an interrupt via the PIA during a handshake operation, the peripheral device must activate either the CA1 or CB1 input control line. Recall that activation of these lines will cause the bit 7 status flag to set in the respective control register for polling operations. However, you can also configure the PIA such that activation of either

CA1 or CB1 will generate an interrupt to the CPU. This is accomplished by setting bit 0 of the respective control register when you initialize the PIA. When this control register bit is set, the PIA will generate an interrupt to the CPU when the respective input control line is activated.

The PIA has two output lines, labeled $\overline{\text{IRQA}}$ and $\overline{\text{IRQB}}$, which must be connected to one of the CPU interrupt input lines as shown in Fig. 4-8. When a peripheral device activates CA1, the $\overline{\text{IRQA}}$ line will go low and generate an interrupt to the CPU. Likewise, when a peripheral device activates CB1, the $\overline{\text{IRQB}}$ line will go low and generate an interrupt to the CPU. The interrupt line will be "held" low until the CPU performs a read operation on the respective PIA port. The $\overline{\text{IRQA}}$ and $\overline{\text{IRQB}}$ lines of the PIA can be connected to separate CPU interrupt input lines or they can be wire-ORed together and connected to the same CPU interrupt input line. Of course, bit 0 of the respective control register must be set (logic 1) as illustrated in Fig. 4-8 for any interrupts to be "passed on" by the PIA. If bit 0 is cleared, a polling routine must be used to detect the active CA1 or CB1 input line status as we discussed earlier.

The advantage of using the PIA to generate interrupts to the CPU is that handshake control of the I/O operation can be implemented efficiently. For example, input handshaking can be provided by using CA1 as an input line and CA2 as an output line as illustrated in Fig. 4-9. The idea here is the same as before, except that an active CA1 input line will cause an interrupt to be generated to the CPU via the

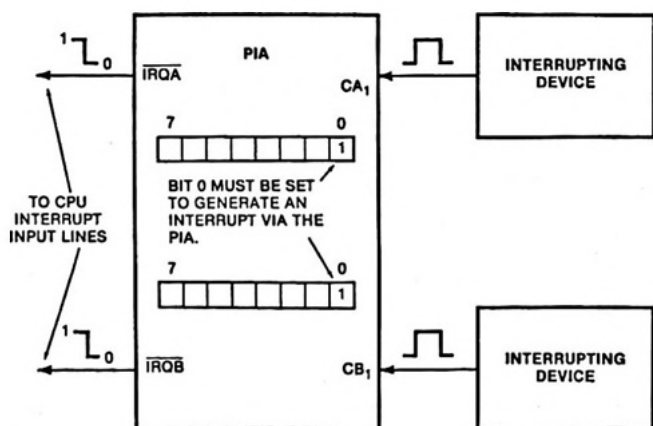


Figure 4-8 Interrupt generation via the PIA.

$\overline{\text{IRQA}}$ line of the PIA. When the CPU accepts the interrupt, it will go to the interrupt service routine which reads port A. Reading port A will *clear* the interrupt by returning the $\overline{\text{IRQA}}$ line high. In addition, the read port A operation will cause an active-low output pulse to be generated to the input device via the CA2 output control line. Thus, the handshake is complete.

Notice from Fig. 4-9 that the PIA control register configuration required for interrupt control differs only by the bit 0 status of that required for programmed control, or polling, of the input operation. Recall that programmed control of the input handshake required that the port A control register be configured with the decimal value 44. For interrupt control of the handshake operation, you must configure the control register with the decimal value 45.

Likewise, output handshaking can be provided with interrupt control via port B as illustrated in Fig. 4-10. Again, the port B control register must be configured with the decimal value 45 during PIA initialization. The only difference between this and an input handshake via port A is that the CB2 output signal is generated by a *write* port B operation.

Now, you are probably wondering how interrupt control of I/O operations can be accomplished with your Color Computer. Recall that there are two CPU interrupt input lines provided on the expansion connector— $\overline{\text{RESET}}$ and $\overline{\text{NMI}}$. Using the PIA interface circuit discussed previously (Fig. 4-1), you can connect the $\overline{\text{IRQA}}$ or $\overline{\text{IRQB}}$ interrupt lines of the PIA to either the $\overline{\text{RESET}}$ or $\overline{\text{NMI}}$ pins on the expansion

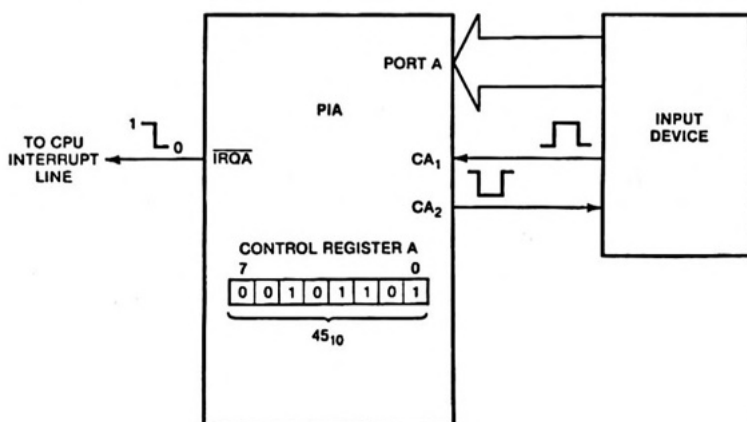


Figure 4-9 Input handshaking using interrupt control via port A of the PIA.

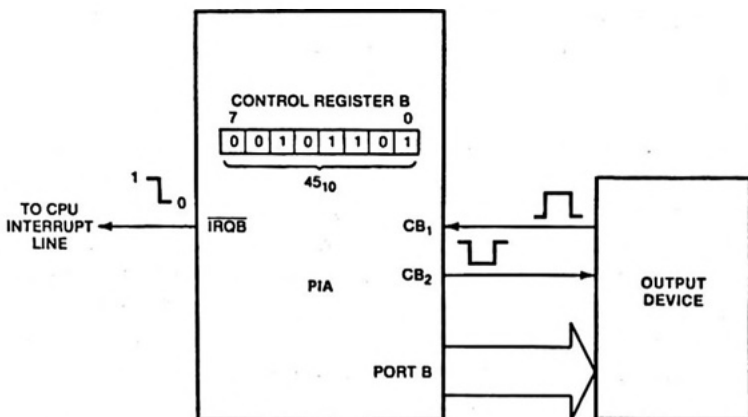


Figure 4-10
Output handshaking using interrupt control via port B of the PIA.

connector to generate an interrupt to the CPU. When connected to the $\overline{\text{RESET}}$ pin, an interrupt generated through the PIA will simply cause the computer to discontinue any processing and display the Color BASIC copyright notice. It's the same as if you were to push the RESET button on the back of the computer case. The $\overline{\text{RESET}}$ interrupt could be used to automatically reset, or reinitialize, the computer as the result of some external event.

You can provide interrupt control of I/O operations by connecting $\overline{\text{IRQA}}$ or $\overline{\text{IRQB}}$ to the $\overline{\text{NMI}}$ interrupt input line on the expansion connector. Recall that the $\overline{\text{NMI}}$ interrupt is a nonmaskable interrupt and is not used internally by the computer. When the $\overline{\text{NMI}}$ interrupt input line is activated (low), the computer will go to RAM address 265 to be directed to the beginning of your interrupt service routine. You must insert a 6809E assembly language JUMP instruction at this address to direct the CPU to the beginning of the interrupt service routine. The assembler code for the JUMP instruction is 126_{10} . This code must be followed in memory locations 266 and 267 by the beginning address of the interrupt service routine. For example, suppose you wish to locate your interrupt service routine at address 2000_{10} (hex address 07D0). The following series of POKE instructions must be executed prior to the interrupt to direct the CPU to address 2000_{10} when the interrupt occurs:

```
POKE 265,126
POKE 266,7
POKE 267,208
```

Note that $7_{10} = 7_{16}$ and $208_{10} = D0_{16}$. Consequently, 7,208 is equivalent to $07D0_{16} = 2000_{10}$. Obviously, the interrupt service routine must be a 6809 assembly language program since you cannot locate BASIC programs at specific addresses. After the interrupt service routine has been executed, the computer will automatically return to where it left off before the interrupt was generated provided that a return command (code 59_{10}) is the last instruction in the service routine.

SUMMARY

In this chapter, the Color Computer expansion connector signals were described in detail. You learned how to provide a parallel interface for your computer by using a PIA. You were also acquainted with two methods of I/O control—programmed control, or polling, and interrupt control. In addition, you saw how the PIA can be used to perform input and output handshaking operations under both programmed and interrupt I/O control. We suggest at this time that you go to Chapter 6 and perform the PIA interfacing experiments (Experiments 1 and 2). This will help to reinforce the ideas and concepts discussed in this chapter. After you have completed the experiments, return to Chapter 5 where we will discuss analog interfacing to the Color Computer.

5 Analog Conversion

Our entire discussion up to this point has evolved around digital signals since most computers today are digital computers. As you are aware, digital computers, like the Color Computer, represent information using a two-state, or binary, system. A digital signal, or binary bit, is either on or off, 1 or 0. Digital computers use collections of binary bits to form binary words that are used to represent arithmetic, logic, and control information. Once a quantity is "digitized," the digital computer can readily interpret and operate on the information.

The world outside the digital computer, sometimes called the "real world," is not a digital world. Rather, it is an *analog* world. The sun *gradually* comes up in the morning and *gradually* sets in the evening. It does not suddenly appear in the morning and disappear in the evening. If it did, we could represent its rising and setting with a digital signal. If a voltage or current were used to represent sunrise and sunset, it would be a continuous waveform that gradually increased in the morning and decreased toward evening. Such a waveform is called an analog signal. Other physical quantities that are analog in nature include temperature, light, pressure, and force, among others. Since the computer is a digital device, conversions must be made between analog and digital signals if the computer is to monitor and interact with such real-world events. For example, an *analog-to-digital* conversion must be made for the computer to monitor temperature. Conversely, a *digital-to-analog* conversion must be made for the computer to control a thermostat.

In this chapter, we will show you how digital-to-analog and

analog-to-digital conversions can be made using devices called digital-to-analog converters, or *DACs*, and analog-to-digital converters, or *ADCs*. In addition, you will learn how these devices can be connected to your computer to monitor and control real-world events. It is not our intent to provide you with a detailed discussion of all the inner workings of *DACs* and *ADCs*. Rather, we will concentrate on the use of these devices and treat them as "black boxes."

OBJECTIVES

At the end of this chapter you should be able to:

- Describe the functional parts of a digital-to-analog converter.
- Describe the operation of a simple 8-bit *DAC*.
- Explain how to use the internal digital-to-analog converter features of the Color Computer.
- Write software programs to control digital-to-analog converters.
- Explain how to interface the MC1408, NE5018, and MC3410 *DACs* to your computer.
- Explain the successive approximation method of analog-to-digital conversion.
- Explain the dual-slope integration method of analog-to-digital conversion.
- Explain how to use the internal analog-to-digital converter features of the Color Computer.
- Use the ADC0804LCN and ADC0809LCN analog-to-digital converters.
- Design an interface for the ADC0804LCN and ADC0809LCN devices.

DIGITAL-TO-ANALOG CONVERTERS

Digital-to-analog converters, frequently called *DACs* or *D/A* converters, are circuits that have digital inputs and a single analog output. The digital inputs are combinations of logic 1s and logic 0s that form a binary number. In the applications and circuits described in this book, these inputs will be compatible with the logic levels that are generated by TTL circuits. Thus, they are TTL-compatible inputs. The analog output provided by a *D/A* converter is a voltage or current that is continuously variable between an upper and a lower limit. So that some basic principles can be illustrated, we will assume that a *D/A*

converter is available with four binary inputs, and with an analog output voltage that has a range from 0 to +15 volts. The 16 possible binary inputs range from 0 (0000) to 15 (1111).

Each of the four binary inputs to the converter is weighted, in much the same way that the bits themselves are weighted. In the 4-bit D/A converter, the four inputs would have weights of 1, 2, 4, and 8, each being an integer power of 2. Since each bit is weighted differently, each bit will "contribute" a different amount to the output voltage of the D/A converter. In this case, the most-significant bit (MSB) has a value of either 0 or 8, so it can contribute either 0 or 8 volts to the output of the converter. The least-significant bit (LSB) will also make a contribution to the total output, this being either 0 or 1 volt. We have assumed that the D/A converter contains the necessary circuitry to accurately add these contributed voltages. In this converter, then, the binary bits have the ability to add either 0 or their particular weight, expressed in volts; that is, 8 volts, 4 volts, 2 volts, and 1 volt. Any voltage between 0 and 15 volts (0, 1, 2, 3, . . . 13, 14, or 15 volts), can be generated by the 4-bit D/A converter through the application of the proper 4-bit binary code to the four inputs of the converter. The relationship between the binary inputs and the voltage output is shown in Table 5-1.

Table 5-1. Four-Bit D/A Converter with Voltage Range from 0 to 15 Volts

Binary Code Input				Output Voltage
D = 8	C = 4	B = 2	A = 1	(Volts)
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15

While the output of the D/A converter was described as being "continuous," you should note that the output of this 4-bit DAC is a series of discrete voltages that are easily measured in steps, whether the steps are a few millivolts, or 1 volt. The voltage level of a single step is called the *step size* of the DAC. An output for our hypothetical 4-bit DAC is shown in Fig. 5-1. For illustration purposes, we have used a 4-bit free-running counter to supply the digital input to the DAC. The counter will count from 0000_2 to 1111_2 then "roll over" and repeat the count. The minimum output voltage is 0 volts. Notice that the output is *staircased* above zero. Each step in the staircase contributes 1 volt to the output. Therefore, we say that the step size of the DAC is 1 volt. The output in Fig. 5-1 increases in steps of 1 volt to a maximum level of 15 volts, which represents a binary input of 1111_2 . There are 15 discrete steps and, therefore, the maximum output voltage is $15 \times 1 \text{ volt/step}$, or 15 volts.

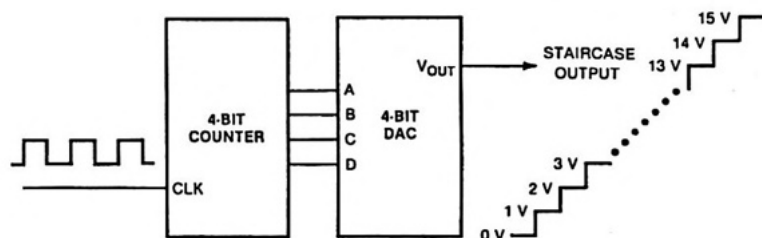


Figure 5-1
Typical staircase output of a D/A converter.

The block diagram for a typical D/A converter is shown in Fig. 5-2. In most commercially available DACs, the binary inputs control switches that either add, or do not add, to the total output a current that is proportional to their weight. It is much easier to switch currents in these devices than it is to switch voltages. Consequently, most commercial DACs are current output devices. A precision resistor ladder is used to control the individual currents, and you may see the notation "R/2R resistor ladder network" associated with DACs.

In computer interfacing, you will probably find little use for DACs with fewer than eight bits. In fact, 10- and 12-bit DACs are quite common in computer systems. As the number of binary inputs is increased, so is the resolution of the converter. This simply means that the voltage steps that are output by the converter are smaller. Thus, an 8-bit D/A converter would have a resolution of one part in 256,

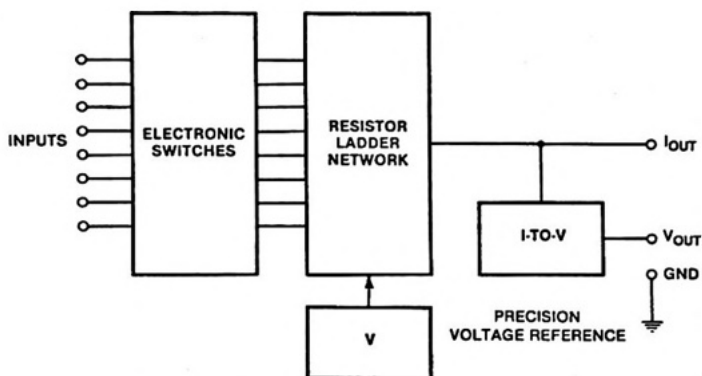


Figure 5-2
Block diagram of a typical D/A converter.

and a 12-bit converter would have a one-part-in-4096 resolution. You will probably find that most interfacing tasks that call for an analog control voltage are easily handled with 8- or 10-bit DACs.

Digital-to-analog converters are very fast, converting the applied digital value into the corresponding voltage output within a micro-second. There is no other control required; you simply apply the binary bits, and the voltage output of the converter changes accordingly. Typically, output ports, or latches, are used in computer interfaces to store binary values for use by DACs. You should be familiar with these devices from Chapter 3.

Before we start discussing the interfacing of DACs, we need to spend a bit more time discussing the characteristics of the output of the converter. For the purposes of this discussion, we will assume that there is an 8-bit DAC available, with its output specified as being within the range of 0 to +10 volts. In this converter, there are 256 different voltages, each of which consists of contributions from the eight weighted binary inputs. The MSB would contribute 5 or 0 volts, while the bit next to the MSB would contribute 2.5 or 0 volts, and so on for each of the eight bits, as shown in Table 5-2. If you add all of these eight weighted voltages, as the converter would for a binary input of 11111111_2 , or 255_{10} , you would find that the total is 9.9609375 volts, and not 10 volts as was specified. How can this be? Just remember that the MSB contributes one-half of the full-scale voltage, the next to MSB contributes one-quarter, and so on. Only an infinite number of bits would be able to have their corresponding voltages added to reach the 10-volt full-scale specification. Actually,

Table 5-2. Voltage Weights for an Eight-Bit D/A Converter

Bit Position	Voltage*
Most-significant bit	5.0000000 = $\frac{1}{2}$ of full scale
	2.5000000 = $\frac{1}{4}$ of full scale
	1.2500000 = $\frac{1}{8}$ of full scale
	0.6250000 = $\frac{1}{16}$ of full scale
	0.3125000 = $\frac{1}{32}$ of full scale
	0.1562500 = $\frac{1}{64}$ of full scale
	0.0781250 = $\frac{1}{128}$ of full scale
Least-significant bit	0.0390625 = $\frac{1}{256}$ of full scale

*10-volt full-scale converter

the 9.9609375-volt full-scale output is close enough to 10 volts for most interfacing tasks. In fact, the difference between the maximum output from the DAC and the 10-volt maximum that was specified is 0.0390625 volt, the contribution from the LSB. Ten-bit and 12-bit DACs with the same full-scale range have smaller voltage steps, or greater resolution, so that they can more closely approximate the 10-volt full-scale voltage, as noted in Table 5-3. In this discussion, we have provided all of the decimal digits for the calculations of the voltages for each step. In practice, this is not really practical, since most measurements are useful to three significant digits. Thus, the 9.9609375-volt maximum would really be measured at about 9.96 volts.

Not all converters have the same full-scale voltage output. Some converters are available with a variety of outputs; for example, ± 2.5 volts, ± 5 volts, 0–10 volts, and 0–1 volt. Current-output DACs are also available, having outputs of ± 1 mA or 0–10 mA. The converters that have both positive and negative output voltages (currents) are called *bipolar* DACs, while those that have either positive or negative outputs are called *unipolar* DACs. There are applications for both types. If you need a DAC that has a voltage range that is not readily available among the standard converters offered by manufacturers, you can use operational amplifiers (op-amps) to change the range, to add an offset voltage, to invert the voltage, and perform other “con-

Table 5-3. Voltage Resolution of Various D/A Converters

Converter	Max. Full Scale*	Resolution	%Resolution
8-Bit	9.9609375 volts	0.0390625 volt	0.39
10-Bit	9.9902344 volts	0.0097656 volt	0.1
12-Bit	9.9975588 volts	0.0024412 volt	0.024

*10-volt full-scale converter

version" functions on the analog output from the converter. Op-amps are also used to convert the current output by some DACs into a voltage.

Just as the converters have different types of outputs, there are devices that have different types of digital inputs, too. Some of the commonly available input codes include binary, binary-coded decimal (BCD), and offset binary. In most cases, the binary input models will be the most useful in computer interfacing applications. Converters are also available with inputs that are compatible with the various types of logic families; for example, CMOS, ECL, and TTL. The wide variety of DACs makes them easy to use in computer interface circuits.

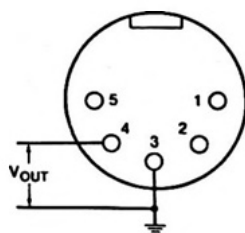
The Internal Color Computer DAC

The Color Computer has an internal 6-bit DAC that is used for a variety of purposes. The analog output signals provided by the internal DAC are used for the audio cassette output, sound to the TV monitor, and joystick inputs. Actually, an analog-to-digital conversion must be provided for the joystick input signals. However, the internal DAC is used as part of an A/D converter to provide analog-to-digital conversions for the computer.

In addition to its internal uses, the DAC output can also be controlled by a user program and obtained from pin 4 of the audio cassette port, as shown in Fig. 5-3. Analog voltage output values from 0 volts to approximately 1.0 volt can be obtained. However, the output resolution is not very high since the converter is only a 6-bit DAC. The output resolution is one part in 64, which translates to a 0.0156-volt step size within a 0 to 1.0-volt output range.

You can provide an analog output via the internal DAC by writing a value to address 65312 using a POKE instruction. The six most-significant bits contained at address 65312 will be converted to an analog output by the internal DAC, as shown in Fig. 5-4. The two least-significant bits contained at address 65312 are not used by the internal DAC. Consequently, data values between 0 and 252 will pro-

Figure 5-3
The analog output of the internal 6-bit DAC can be obtained at pin 4 of the audio cassette port.



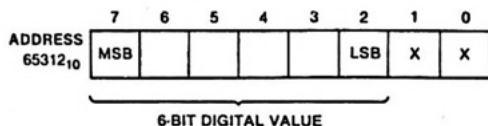


Figure 5-4

The internal DAC will convert a 6-bit digital value contained in the six most-significant bit locations of address 65312.

vide an output voltage value between 0 volts and approximately 1.0 volt. However, only 64 data values within this range will produce discrete output voltage steps, since four given data values may produce the same output level. For example, data values 0, 1, 2, and 3 will all produce the same output voltage level (0 volts). Likewise, data values 4, 5, 6, and 7 all produce a 0.0156-volt output level, and so on.

You can write BASIC programs to generate an output waveform using the internal DAC. The following program will generate the positive voltage *ramp* shown in Fig. 5-5:

```

100 FOR X = 0 TO 252 STEP 4
110 POKE 65312, X
120 NEXT X
130 GOTO 100

```

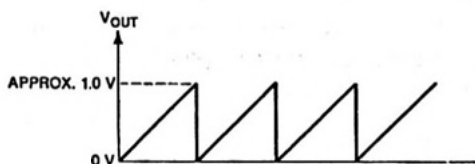


Figure 5-5

A positive voltage ramp generated by a BASIC program using the internal DAC.

Notice that the voltage ramp increases from 0 volts to approximately 1.0 volt as X is incremented in steps of 4 from the value 0 to 252. The GOTO statement in line 130 causes the ramp to fall back to 0 volts and repeat itself such that a continuous output voltage waveform is produced.

The triangular waveform shown in Fig. 5-6 can be produced by executing the following program:

```

100 FOR X = 0 TO 252 STEP 4
110 POKE 65312, X
120 NEXT X
130 FOR X = 252 TO 0 STEP -4

```

```

140 POKE 65312, X
150 NEXT X
160 GOTO 100

```

Here, the output voltage increases from 0 volts to 1.0 volt as X is incremented, and decreases from 1.0 volt to 0 volts as X is decremented. The output cycle repeats itself due to the GOTO statement in line 160.

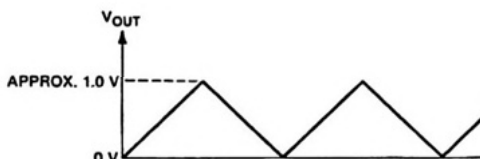


Figure 5-6 A triangular waveform voltage generated by a BASIC program using the internal DAC.

To generate one cycle of the square waveform shown in Fig. 5-7, you must output a maximum voltage value and provide a delay, then output the minimum voltage value and delay. You then repeat the process with a GOTO command to provide a repeating square waveform. The following program will accomplish this task:

```

100 POKE 65312, 0
110 FOR I = 1 TO 10
120 NEXT I
130 POKE 65312, 252
140 FOR I = 1 TO 10
150 NEXT I
160 GOTO 100

```

Finally, you can generate the sine wave shown in Fig. 5-8 with the following BASIC program:

```

100 FOR X = 0 TO 360
110 Y = 128 + INT(127*(SIN(X*0.017453)))
120 POKE 65312, Y
130 NEXT X
140 GOTO 100

```

The SIN function in line 110 of this program computes the sine of angles that are expressed in radians rather than degrees. Consequently, a conversion is required so that degrees are converted to radians. This is the purpose of the 0.017453 factor in line 110.

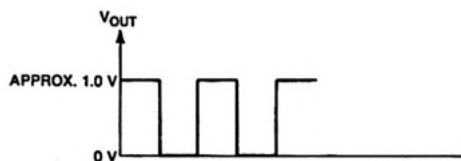


Figure 5-7 _____
A square waveform voltage generated by a BASIC program using the internal DAC.

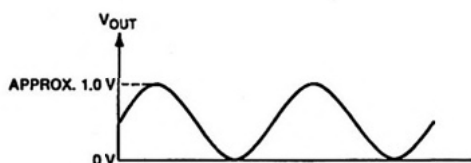


Figure 5-8 _____
A sine waveform voltage generated by a BASIC program using the internal DAC.

The sine program also illustrates some other things that must be considered when using a BASIC program to generate data for a simple digital device like a DAC. The values for the sines of different angles range between -1 and $+1$, so how can a *unipolar* DAC be used to provide the sine function voltage values? The full-scale range of sine values (-1 to $+1$) must be converted to the full-scale range of the DAC, and the *bipolar* sine values must be converted to *unipolar* values for use by the DAC. These conversions are performed by line 110 of the sine program. Notice that if the sine of an angle is zero, the resulting value for the variable Y is 128, which when transferred to the DAC will provide an output voltage at the middle of the DAC output range. The sine of the unknown angle is multiplied by 127 so that the sines of the angles between 0° and 360° will be converted to values between -127 and $+127$. When these values are added to $+128$, the resulting range for Y is $+1$ to 255 , which can be readily used by the DAC. The INT, or integer, command is used to generate integer values of the sine function.

You will find many applications for DACs in which a particular function like a positive ramp, or a set of data values (sines of angles) are required. High-level languages, like BASIC, allow you to write programs to generate the required data much easier than assembly language programming. However, there is one big problem. High-level languages cannot output the data values fast enough for many appli-

cations. Recall that BASIC is relatively slow. The maximum output frequency of the waveforms discussed previously is on the order of 6 Hz, using the internal Color Computer DAC. This relatively low output frequency might be useful for driving a plotter, but it is too slow to drive an oscilloscope display. Other factors control this maximum frequency output; however, the primary limiting factor is the relatively slow execution of the BASIC commands.

Now, recall that assembly language programs execute up to 1000 times faster than equivalent BASIC programs. Thus, the optimum situation is to use the power of BASIC to generate the data values, and the speed of the assembly language to output the data values to the DAC. This is accomplished by writing three programs:

1. A BASIC program to generate the data values and store them in a memory *look-up* table.
2. An assembly language program to obtain the data values from the memory look-up table and output those values to the DAC.
3. A BASIC program to load the assembly language program in memory.

As an example, let's see how you might generate a sine waveform using this procedure. First, the following is a BASIC program that will generate the data values and store them in a memory look-up table:

```
100 N = 2000
110 FOR X = 0 TO 360 STEP 2
120 Y = 128 + INT(127*(SIN(X*0.017453)))
130 POKE N,Y
140 N = N + 1
150 NEXT X
160 END
```

There are 180 data values generated by the sine function in line 110. We have chosen to store these data values in a look-up table that begins at RAM address 2000. The variable "N" determines where the data value is to be stored in the data table. Notice that N is incremented from 2000 to 2179.

Next, you must write a 6809 assembly language program that will obtain the sine data values from the look-up table and output those values to the Color Computer DAC located at address 65312. The following 6809 assembly language program will do the trick:

Assembly Code	Hex Op-Code	Decimal Op-Code
LDX #	8E	142
07	07	7
D0	D0	208
LDA, X+	A6	166
80	80	128
STA \$\$	B7	183
FF	FF	255
20	20	32
CMPX #	8C	140
09	08	8
33	83	131
BNE	26	38
F6	F6	246
BRA	20	32
F1	F1	241

We don't expect you to know how this assembly language program works. It is only important that you know what it does.

Finally, you must write a BASIC program that loads the assembly language program into memory. The following BASIC program will do the job:

```

10 CLEAR 25, 8000
20 DATA 142, 7, 208, 166, 128, 183, 255, 32, 140, 8, 131, 38,
  246, 32, 241
30 FOR I = 0 TO 14
40 READ B: POKE 8000 + I, B
50 NEXT I
60 END

```

Notice the use of the DATA statement in this program to load the 6809 assembly language program into memory. The assembly language program begins at address 8000. Thus, you must enter the command *EXEC 8000* to execute the assembly language program and output the sine waveform data to the DAC. The DAC output is available at pin 4 of the Color Computer audio cassette port. In Chapter 6, you will generate voltage waveforms using this procedure.

Recall that the Color Computer uses a 6-bit internal DAC. However, many analog control applications require greater than 6-bit resolution. You must, therefore, interface an 8-, 10-, or 12-bit DAC to the com-

puter to provide this increased resolution. This is the topic of our next discussion.

DAC Interfacing

When a digital-to-analog conversion resolution of greater than six bits is required, you must interface a higher resolution DAC to your computer. We will discuss general DAC interfacing concepts first, then you will be acquainted with several real DAC products.

Interfacing an 8-bit DAC to your computer is relatively easy. You will use the PIA interface that was developed in Chapter 4. An 8-bit DAC is connected to one of the PIA ports as shown in Fig. 5-9. It doesn't matter which port is used; however, the port that is used must naturally be configured as an output port. Digital data to be converted is simply written to the respective port data register using the same procedure that we discussed for the 6-bit internal DAC. Since many DACs are low-level current-output devices, the addition of a current-to-voltage transducer and amplifier may be required. This can be a standard operational amplifier, or op-amp, circuit as shown inside the dotted box in Fig. 5-9. The load resistor (R_L) converts the current output of the DAC to a voltage output by Ohm's law. The low-level voltage output is then amplified by the op-amp to a usable level. Thus, the final voltage output becomes:

$$V_{OUT} = (I_{OUT} R_L) \left(\frac{R_2}{R_1} + 1 \right)$$

Finally, the addition of a low-pass filter will smooth out the steps of the DAC output. The filter cutoff frequency should be equal to the highest frequency component of the output waveform to avoid attenuation. The filter circuit is optional, and the application for the output waveform will dictate its use.

When interfacing to DACs that have a resolution greater than eight bits, you must use both PIA ports as shown for the 10-bit DAC in Fig. 5-10. Here, port A supplies the eight least-significant bits and port B supplies the two most-significant bits. However, a latching circuit must be placed between the PIA port output lines and the 10-bit DAC. We have used a 74LS373 8-bit latch and a 74LS75 2-bit latch. Without this latching circuit, "glitches" would appear in the analog output waveform. Here's how it works:

Suppose the 10 DAC input lines were connected directly to the PIA port lines. Now, suppose the DAC is providing an analog output level that corresponds to the 10-bit digital value 00 1000 0000. You now

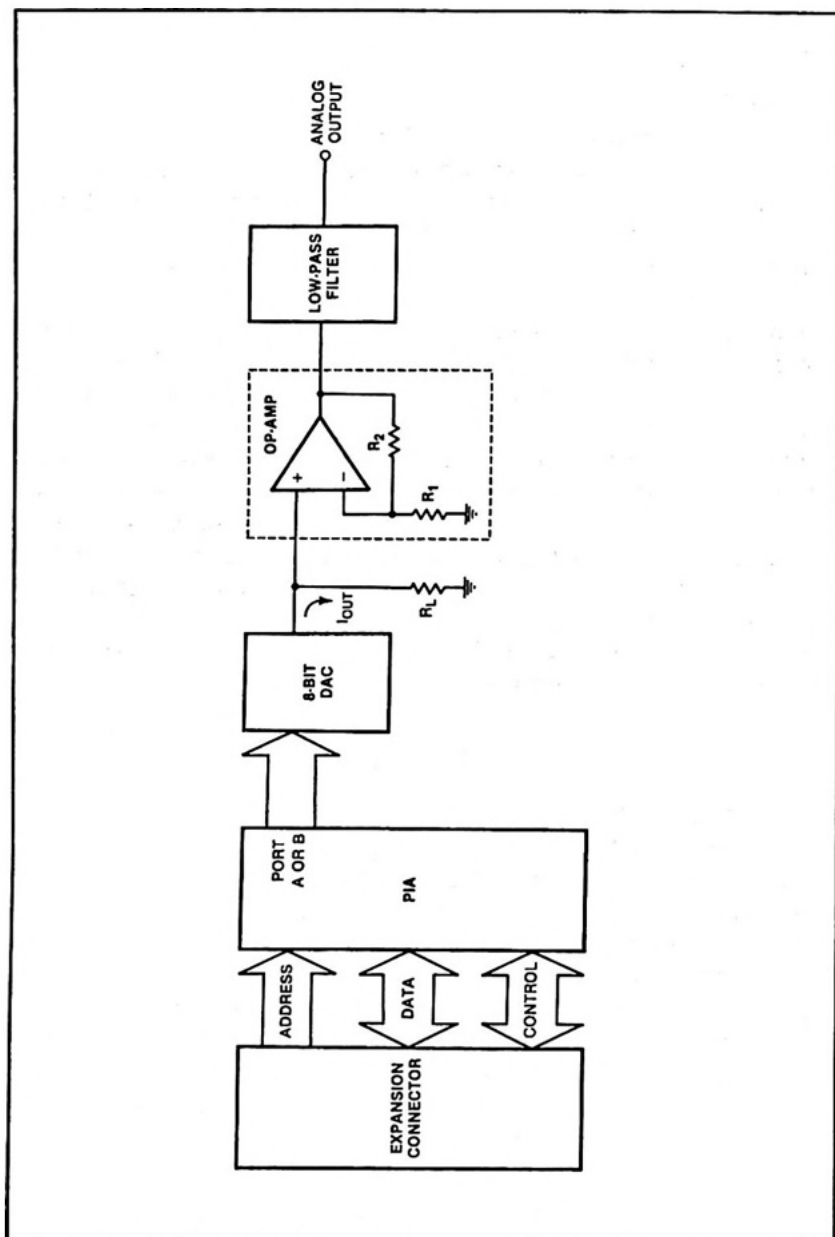


Figure 5-9 _____
An 8-bit DAC connected to the PIA interface.

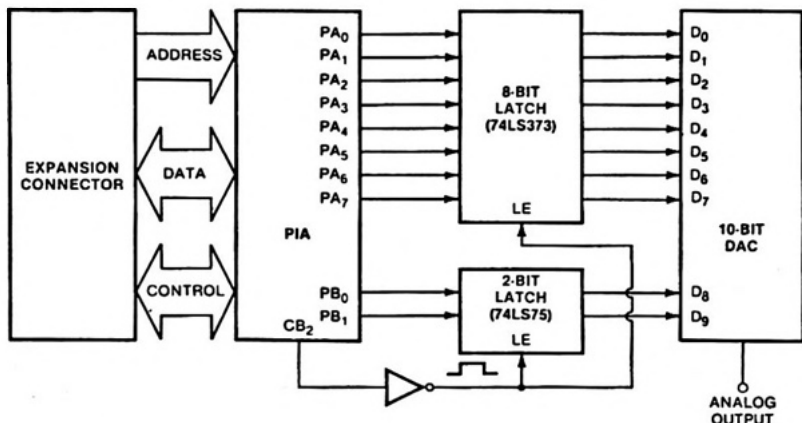


Figure 5-10
A 10-bit DAC connected to the PIA interface.

wish to convert a second digital value, say 01 0000 0000. The second value is clearly a higher value than the first and, thus, should create a higher analog output level. However, when you write the new digital value to the PIA, you must write the eight least-significant bits (0000 0000) to port A first, then the two most-significant bits (01) must be written to port B. Here's where the problem arises. Immediately after you write the eight least-significant bits to port A, the DAC "sees" an intermediate input value of 00 0000 0000, which consists of the eight least-significant bits of the new data value and the two most-significant bits of the old data value. Consequently, an intermediate analog output value of 0 volts will be generated by the DAC. Obviously, this output "glitch" is not desirable.

To eliminate the glitch problem, you must *double buffer* the interface with latches as shown in Fig. 5-10. The latches are used to capture the entire 10-bit data value before passing it on to the DAC. You must configure the PIA for output handshaking, as we discussed in Chapter 4 so that the CB2 output control line can be used to enable the latches. Then, to output a 10-bit value to the DAC, you must write the eight least-significant data bits to the port A data register and the two most-significant data bits to the port B data register. When the two most-significant bits are written to port B, the CB2 output line will go low for one CPU cycle and enable the latches. The entire 10-bit data value will then be passed on to the DAC.

You might also need to provide a current-to-voltage transducer and amplifier for the DAC output as we discussed earlier. In addition, a

low-pass output filter might be required. However, the resolution of a 10-bit DAC is one part in 1024, resulting in a very small step size. Consequently, most applications will not require filtering.

You can connect 12- and 14-bit DACs to your computer in the same way as a 10-bit DAC. The only difference is that you must use more of the PIA port lines and increase the size of the latching circuit. Now, let's look at some real DAC products.

The MC1408-8 DAC Interface

The Motorola MC1408-8 DAC is an 8-bit device that has been quite popular for use in small computer systems. An MC1408-8 interface circuit is shown in Fig. 5-11. The eight data input lines to the DAC are connected directly to either port A or port B of the PIA. An operational amplifier has been used to provide the necessary current-to-voltage conversion, so that the circuit will provide a 10-volt full-scale output. An external reference voltage is also supplied to the MC1408-8, 2.5 volts in this case, which is applied through a 1250-ohm resistor to provide a known current of 2 mA at full scale. This means that the output of the MC1408-8 DAC will be 2 mA at full scale. A 5000-ohm feedback resistor is used in the op-amp circuit to provide a 10-volt full-scale output. An LM318 op-amp, or equivalent, has been selected for its high-performance characteristics.

The NE5018 DAC Interface

Another popular 8-bit DAC interface is shown in Fig. 5-12. This circuit uses the Signetics NE5018 DAC. Here, additional components have been added so that the output of the converter can be adjusted for the maximum voltage output desired. The full-scale adjustment allows you to adjust the output circuit so that its maximum is very close to 10 volts for an input value of $1111\ 1111_2$. You can also configure the NE5018 to provide a bipolar output of ± 5 volts. In addition, an offset adjustment allows you to set the output of the DAC to 0 volts for a digital input value of $0000\ 0000_2$. You can use the NE5018 without the full-scale and offset controls by simply removing the four control resistors from the circuit. Most commercial DACs can be adapted to include these optional controls.

The MC3410 DAC Interface

The MC3410 is a 10-bit DAC manufactured by Motorola, and is very similar to the MC1408-8 DAC discussed earlier. An interface circuit for the MC3410 is shown in Fig. 5-13. Notice the use of the 74LS373 and

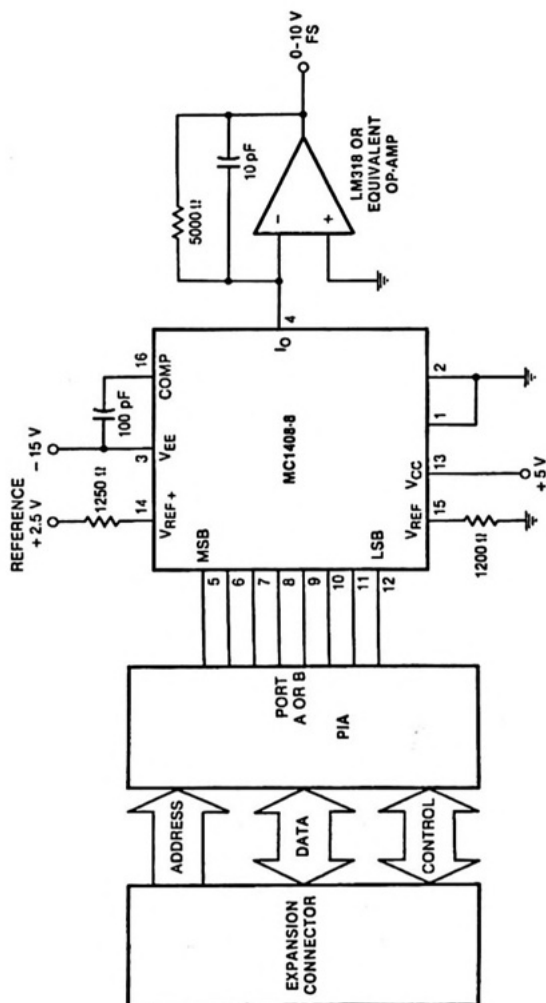


Figure 5-11 The Motorola MC1408-8 8-bit DAC connected to the PIA interface.

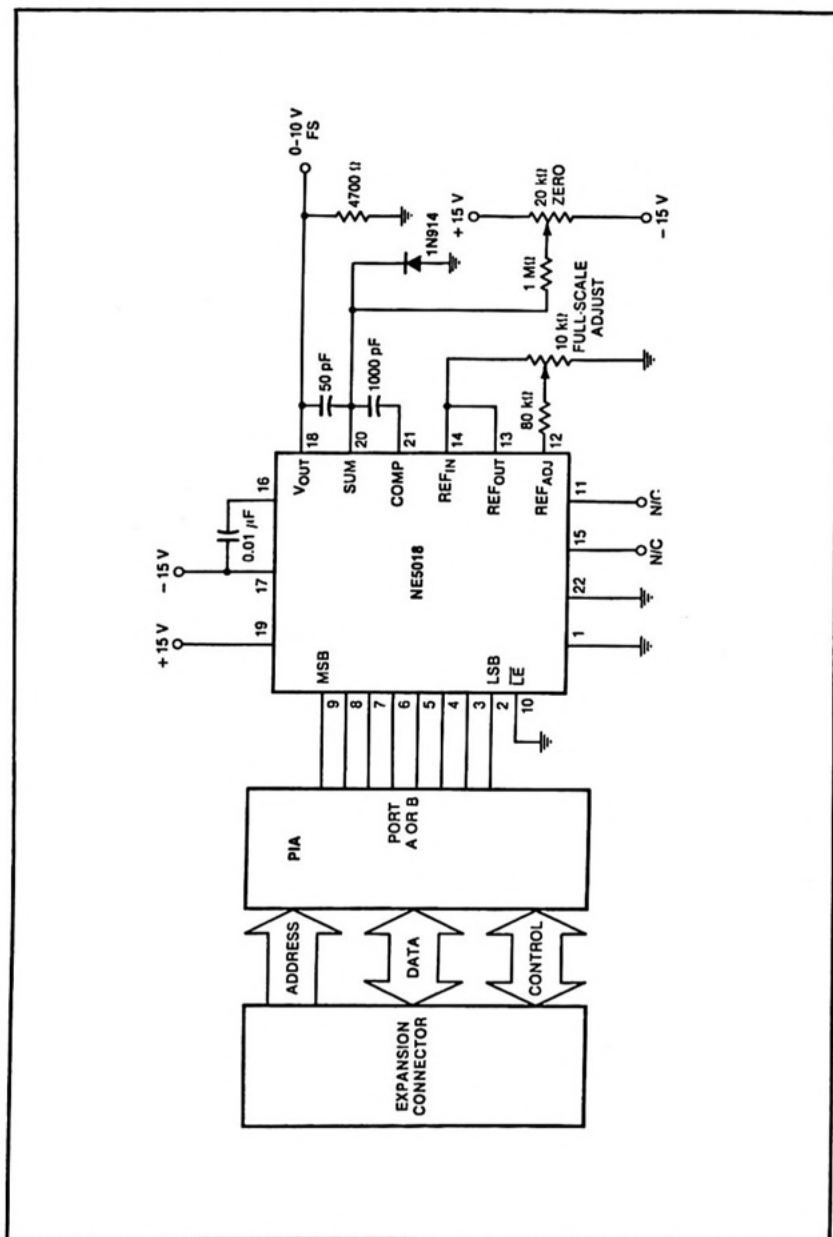


Figure 5-12 The Signetics NE5018 8-bit DAC connected to the PIA interface.

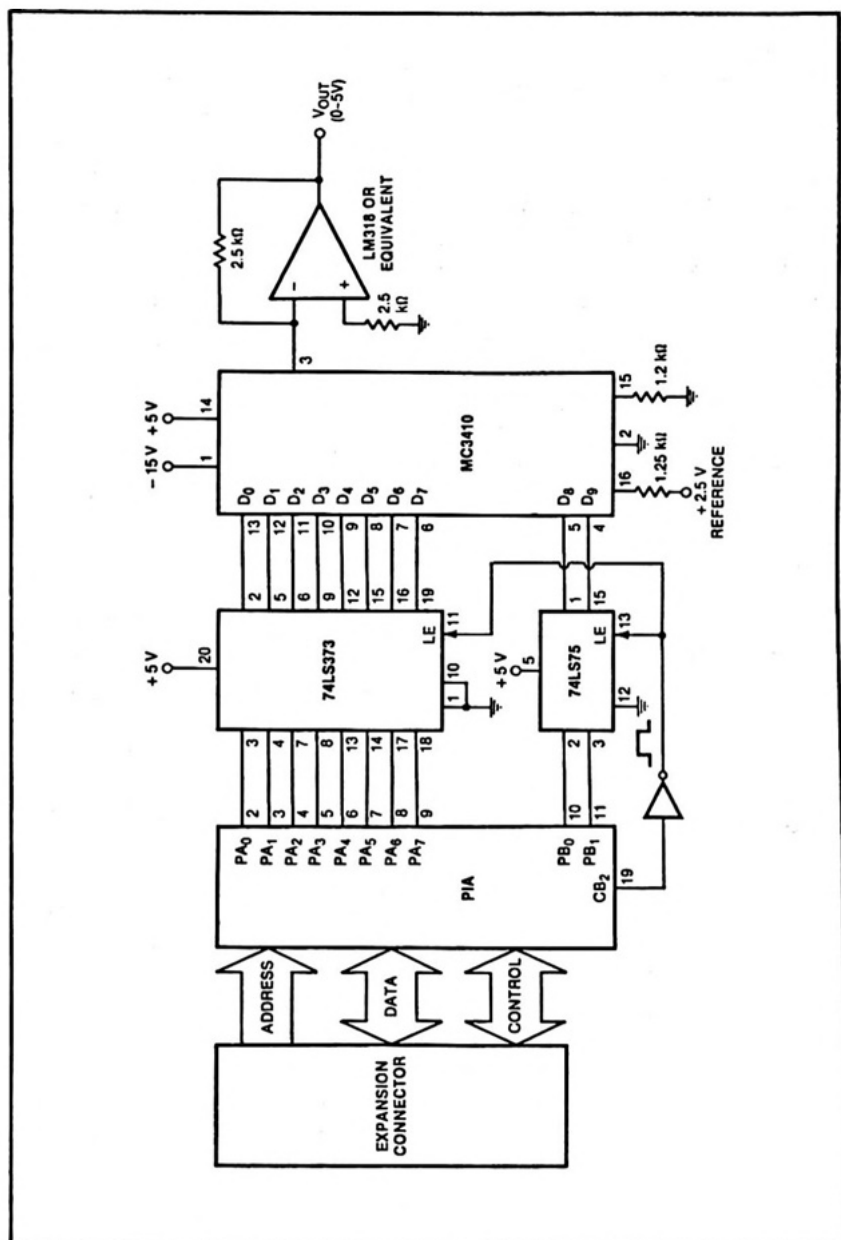


Figure 5-13 The Motorola MC3410 10-bit DAC connected to the PIA interface.

74LS75 latches to provide buffering between the MC3410 and the PIA. Again, an op-amp has been provided as a current-to-voltage transducer for the DAC output. The voltage output range provided by this circuit is 0 to 5 volts.

There are many other digital-to-analog converters available from a number of different manufacturers. A representative list of these has been included at the end of this chapter for your reference purposes.

At this time, we suggest that you go to Chapter 6 and perform the digital-to-analog conversion experiments. Then, return to this chapter and read the next section on analog-to-digital conversion.

ANALOG-TO-DIGITAL CONVERTERS

In this section, we will be describing the use of analog-to-digital converters, also called A/D converters or ADCs. These devices give the computer the ability to measure an analog voltage so that physical measurements of pressure, temperature, light intensity, and others may be made available to the computer for processing with BASIC language programs. In some cases it will be easy for you to sit down in front of your computer and enter various measurements by using the keyboard, while in other cases it will be easier to use a sensor and an analog-to-digital converter that is controlled by the computer. If you are simply converting a single temperature measurement from degrees Fahrenheit to degrees Celsius, then it is easy to enter the value into a short BASIC program that will perform the conversion. However, if you are trying to measure the indoor and outdoor temperatures of a building every 10 minutes, over a 24-hour period, it would certainly be convenient if a small computer could do this without any actions from the operator. At other times, the measurements that are required will be acquired at such short time intervals that it will be impossible to record the information fast enough by hand. Again, a small computer is ideally suited to quickly acquire such measurements.

Analog-to-digital converters are not difficult to use with small computer systems. In this section, we will show you how to use the limited internal analog-to-digital converter features of the Color Computer. In addition, we will show you how to interface several ADC devices to your computer using the PIA interface developed in Chapter 4.

ADC Fundamentals

Most analog-to-digital converters operate by comparing an unknown applied voltage, which is to be "converted," to a known volt-

age, or a series of known voltages. Since ADCs are specified as having a particular range of input voltages, and a certain number of output bits, their resolution, or resolving power, is readily determined. For example, an ADC with a 0 to 10-volt input and a 10-bit output can resolve any unknown voltage that is within its input range into a 10-bit binary word between 0 and 1023. Therefore, the resolution of the converter is one part in 1024, or 9.76 millivolts. In most cases, you will use ADCs that have binary outputs, although other output coding formats are readily available. When ADCs are used with binary computers, the models with binary outputs are the best, since no code conversions in either software or hardware are required.

Several different techniques are used to perform analog-to-digital conversions. Two of the most common conversion techniques are *successive approximation* and *dual-slope integration*. Most of the analog-to-digital converters in use today, including the ones discussed in this section, operate using one of these two conversion techniques. We will, therefore, limit our discussion to the successive approximation and dual-slope integration methods of analog-to-digital conversion.

Successive Approximation — This method of analog-to-digital conversion is the fastest and most widely used conversion technique. It uses two main hardware ingredients—a digital-to-analog converter and a comparator. Suppose you wish to construct a 6-bit successive approximation ADC around the PIA interface developed in Chapter 4. To do this, you must connect a 6-bit DAC and a comparator to one of the PIA ports as shown in Fig. 5-14. We have chosen to use port A of the PIA for reasons that will be obvious in the next section. Notice that the six least-significant port A data lines (PA0–PA5) are connected to the DAC input lines. The DAC output voltage (V_{OUT}) and the unknown analog voltage ($V_{UNKNOWN}$) are connected to the two comparator input lines. The comparator output is connected to the most-significant port A data line (PA7). The PIA must be initialized to configure PA0 through PA5 as data output lines, and PA7 as a data input line. Here's how it works:

The unknown analog voltage is applied to the noninverting input of the comparator, as shown in Fig. 5-14. At the start of the conversion process, the most-significant input line to the DAC (PA5) is set. This produces a DAC output equal to one-half of its maximum rating (2.5 volts in Fig. 5-14). The comparator is used to compare this DAC output to the unknown analog voltage level ($V_{UNKNOWN}$). The comparator will provide a low (logic 0) output if the DAC output is greater than the unknown analog voltage, and a high (logic 1) output if the DAC output is less than the unknown analog voltage. The greater-than/

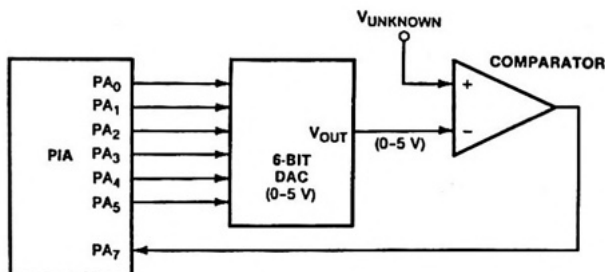


Figure 5-14
A 6-bit successive approximation A/D converter.

less-than condition is then determined by polling the comparator output provided on the PA7 input line of the PIA. If a greater-than condition is detected (PA7 low), the PA5 output line must be cleared. If a less-than condition is detected (PA7 high), the PA5 output line is left set. After this, the PA4 output line is set and the compare procedure is repeated to determine the greater-than/less-than condition between the corresponding DAC output and the unknown analog voltage level. The process is then repeated until all six of the PIA output lines (PA5—PA0) have been "tested." The digital equivalent of the unknown analog voltage is then obtained by reading bits 0 through 5 of the port A data register. The successive approximation of the unknown analog voltage level is illustrated in Fig. 5-15. As you can see, the DAC "homes-in" on the unknown analog voltage level using a bit-by-bit trial-and-error method.

The successive approximation technique is relatively fast since it only requires n comparison tests to be made for an n -bit converter. However, as you can imagine, the converter circuit in Fig. 5-14 requires several software steps. An assembly language program must be written to provide the required software since a BASIC program would be too slow. A flowchart for a 6-bit successive approximation conversion routine is shown in Fig. 5-16.

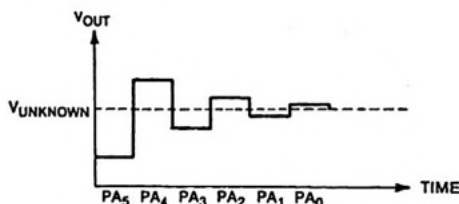
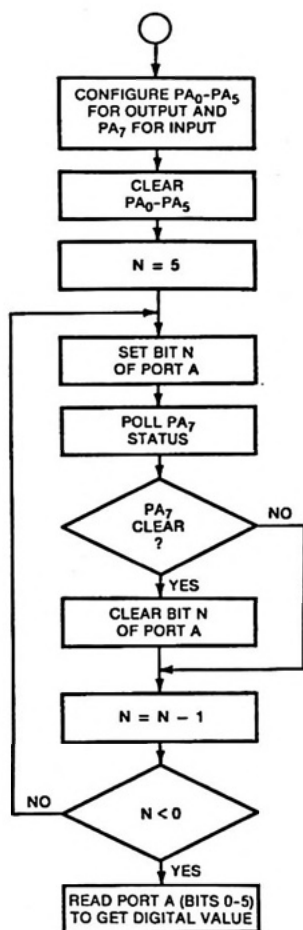


Figure 5-15
Diagram illustrating the successive approximation method of A/D conversion.

Figure 5-16 _____
**Flowchart for a 6-bit successive ap-
proximation A/D conversion routine.**



As you are about to discover, your Color Computer provides analog-to-digital conversions for the joystick controls in the manner just described. There are also 8-, 10-, and 12-bit single-chip successive approximation ADC devices that can be interfaced easily to your computer. These devices provide higher resolution, are relatively fast, inexpensive, and require a minimum amount of software to control the conversion process. We will discuss two representative ADC devices later on in this chapter.

Dual-Slope Integration—This method of analog-to-digital conversion is slower than the successive approximation method, but has an

important advantage of being relatively *immune to noise*. The dual-slope conversion technique uses a voltage integration process that sums an unknown voltage over a fixed period of time, called T_1 . At the end of this period, higher unknown voltages will have higher sums, while lower unknown voltages will have lower sums. The actual conversion process takes place when the integrated value of the applied voltage is reduced at a known rate. Since it will take longer to decrease a high value of integrated voltage than it will to decrease a low value, the period required to cause the integrated voltage to be decreased to zero is a representation of its value and, thus, the unknown analog voltage.

The timing diagram for the dual-slope conversion process is shown in Fig. 5-17. The T_1 period for two unknown voltages is equal, so that at the end of this integration period the higher voltage, A, will have summed to a higher total voltage than will have voltage B. This is analogous to water flowing out of two pipes at different rates. After a fixed period, the water flowing out of the pipe with the higher flow rate will have filled more of a container than the flow from the other pipe will have. Once the integration period has ended, the integrated voltage is discharged at a known rate. Since the rate is fixed, the slopes of all of the discharge plots will be equal, as shown in Fig. 5-17. Only the discharge period will change, being longer for the higher initial unknown voltage input. Thus, the discharge period is proportional to the unknown input voltage that is to be digitized.

All of the functions of integrating, discharging, and measuring the discharge period are performed by the integrating, or dual-slope ADC.

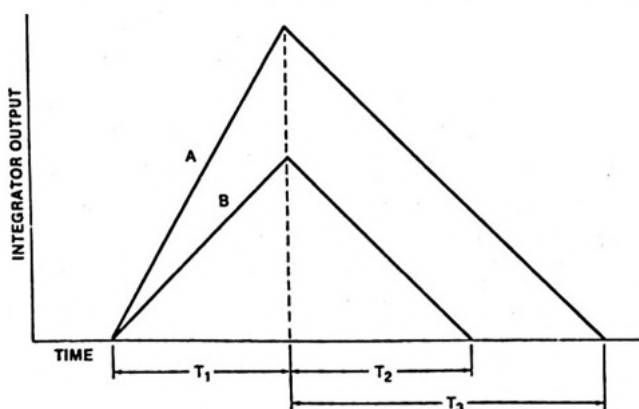


Figure 5-17 Timing diagram for dual-slope conversion of voltages A and B.

The discharge periods, T2 and T3 (Fig. 5-17), are generally used internally by the converter to gate a counter that accumulates a binary count that is proportional to the voltage. You do not have to be concerned with this, though, since the converter manufacturers take care of it.

You are probably wondering why anyone would choose a converter that uses such a slow process. The answer is that the dual-slope A/D converter integrates, or averages, the unknown voltage for later measurement, rather than measuring it quickly and directly, as is the case with a successive approximation converter. Why is the averaging so important? The voltage integration technique also averages some noise, since we assume that the noise will add to the signal as many times as it subtracts from it, as shown in Fig. 5-18. The noise has been greatly exaggerated in this figure so that the actual voltage may be easily seen. The signal is also shown with some periodic noise, such as 60-Hz noise, since it may be easier to see how this adds and subtracts from the signal on a regular basis.

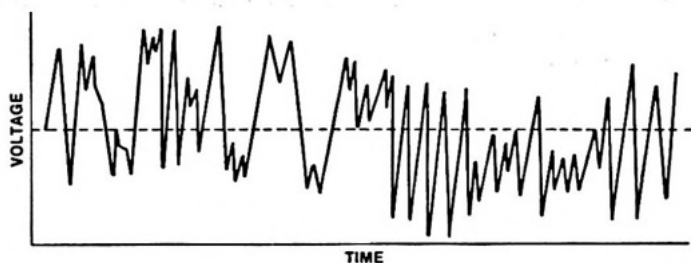


Figure 5-18
Noise is "averaged out" with an integrating ADC.

The timing diagram in Fig. 5-19 shows how a dual-slope A/D converter would integrate this noisy signal. The resulting integrated value of the noisy signal would be very close to the voltage that would have resulted from the integration of the noise-free signal. Actually, we are most interested in the area under the time-voltage line, as shown in the shaded portion of Fig. 5-19. Once the integration period has ended, the discharging is started, and the period that is required for the integrated voltage to reach zero volts is measured. This represents the actual average value of the unknown noisy signal. In this example, the averaging took place during the integration period. In most dual-slope ADCs, the integration period is set at an integer multiple of a 60-Hz sine-wave period, so that line-frequency noise is averaged out.

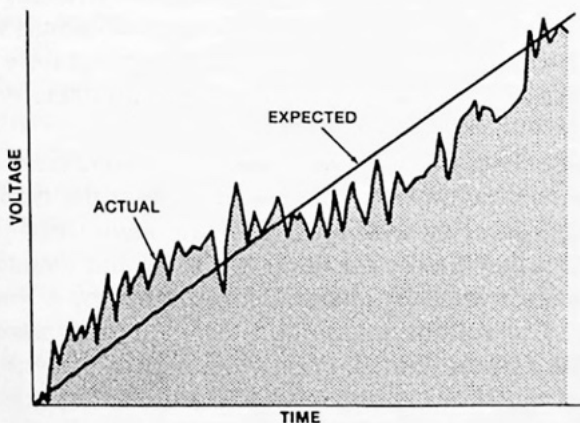


Figure 5-19 Timing diagram showing the integration of a noisy analog signal, versus what is expected for a noise-free signal.

Of course, converters are not perfect, so this is a noise reduction technique, and not all of the noise is eliminated. We will discuss a typical dual-slope ADC later on in this chapter. Now, let's see what internal analog-to-digital conversion capabilities are available in the Color Computer. Then, we will show you how to interface external ADC devices to your computer to provide increased resolution, speed, and noise immunity.

The Internal Color Computer ADC

Whether you realize it or not, you have already been acquainted with the internal ADC of the Color Computer. The example circuit we used in Fig. 5-14 to discuss successive approximation is the actual analog-to-digital converter circuit provided in your computer. The entire circuit is shown in Fig. 5-20. An internal PIA supplies the 6-bit digital compare values to the internal 6-bit DAC that we discussed in the digital-to-analog converter section of this chapter. The DAC output is compared to an unknown analog input voltage by an LM339 comparator device. The unknown analog input voltage is supplied to the comparator by an MC14529 4-to-1 *multiplexer*, or *data selector*, device. The multiplexer simply selects one of four analog input voltages and passes the selected voltage to the comparator. The selection process is controlled by the logic present on the CA2 and CB2 output control lines of the PIA. The four analog input signals are provided via the two joystick input connectors on the back of the Color Computer

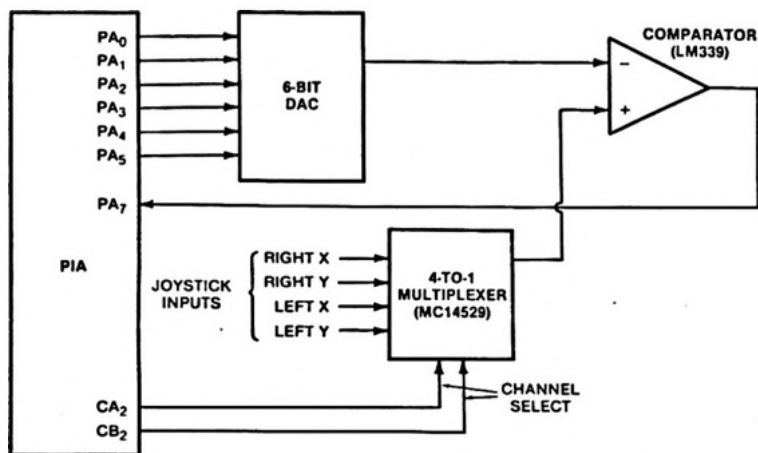


Figure 5-20
The internal ADC circuit of the Color Computer.

case, as shown in Fig. 5-21. Notice that each joystick input connector provides an X input (horizontal) and Y input (vertical) such that the computer can determine the X-Y coordinate position of the joystick. However, you can also use any of the four X/Y connector lines to input a 0–5-volt analog signal to the computer. To do this, you apply the analog signal to the desired X/Y input pin on one of the joystick connectors. Then, you can use the BASIC *JOYSTK* command to convert the 0–5-volt analog input to a 6-bit digital value. For example, the following program will convert an analog signal that is input via the X pin (pin 1) of the right joystick connector:

```
10 X = JOYSTK (0)
20 PRINT X
30 GOTO 10
```

This program will repeatedly display the decimal equivalent of the analog signal.

If you want to convert an analog signal being input on the Y pin (pin 2) of the right joystick connector, you must use the command *JOYSTK (1)* in the program. The *argument* in the *JOYSTK* command can range from 0 to 3 and, thus, select any one of the four analog inputs provided by the joystick connector pins. The *JOYSTK* command select options are summarized in Table 5-4. Actually, the *JOYSTK* command argument selects one of the four analog inputs by provid-

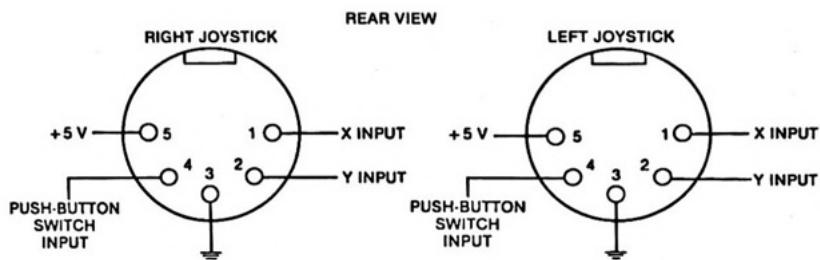


Figure 5-21
The joystick connectors on the back of the Color Computer case.

Table 5-4. Analog Input Select Options Provided by the JOYSTK Command

JOYSTK (n)	Analog Input Selected
n = 0	Right joystick X input (pin 1)
n = 1	Right joystick Y input (pin 2)
n = 2	Left joystick X input (pin 1)
n = 3	Left joystick Y input (pin 2)

ing the proper select logic to the MC14529 4-to-1 multiplexer in the ADC circuit of Fig. 5-20.

There are two problems associated with using the internal 6-bit ADC and the preceding BASIC program. First, you can only provide 6-bit resolution for the analog input signal. This means that you can only resolve to the nearest $5/64$, or 0.08 volt, when converting a 0 to 5-volt signal. To solve this resolution problem, you must interface an external 8-, 10-, or 12-bit ADC to your computer. You will see how this is done shortly.

The second problem with using the internal ADC is a result of the relatively slow BASIC program being used to read the analog voltage level. The BASIC program can only provide approximately 46 conversions per second. This is too slow to keep up with a rapidly changing analog signal, since it only "samples" the analog signal 46 times per second. We have found that by using the BASIC program you can effectively reproduce, or follow, analog signals up to 10 Hz. The diagrams in Fig. 5-22 illustrate the effect of *sampling rate* on analog signal reproduction. By increasing the ADC sampling rate, the computer can reproduce, or follow, the analog signal with greater precision. Of course, higher analog signal frequencies require higher ADC sampling rates.

You can increase the sampling rate of the internal Color Computer

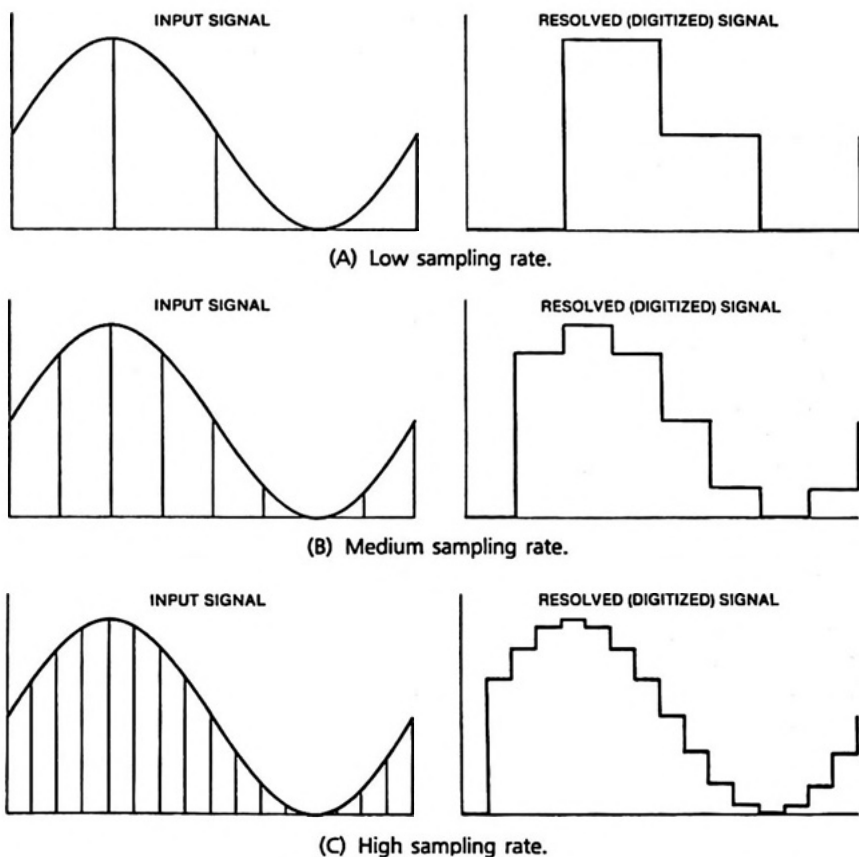


Figure 5-22 _____
 The ADC sampling rate determines how well the computer can follow an analog signal.

ADC by calling an assembly language program to perform the conversions. The Color Computer BASIC ROM contains an assembly language routine that can be called to perform analog conversions via the internal ADC. This ROM routine begins at address 43493₁₆. You can use this routine, or write one of your own. We have found that by using the internal ROM assembly language conversion routine, you can reproduce, or follow, analog signals up to 1 kHz with the internal computer ADC. This technique will be demonstrated as part of the analog-to-digital conversion experiments in Chapter 6.

We stated earlier that higher analog-to-digital converter resolution must be obtained by interfacing an external 8-, 10-, or 12-bit ADC to

your computer. In addition, the external converter will provide higher sampling rates since very little conversion software is required. With this in mind, let's see how external ADCs can be interfaced to your computer.

Interfacing to ADCs

When an increased sampling rate and an analog-to-digital conversion resolution of greater than six bits is required, you must interface an external ADC to your computer. We will discuss some general ADC interfacing concepts first, then show you how to connect several ADC devices to your computer.

A typical 8-bit ADC device is shown in Fig. 5-23. The analog input voltage must be within a specified range to produce a meaningful digital output. Most ADCs provide for input range adjustment through application of reference voltage levels and/or offset adjustments. Both unipolar (+ or -) and bipolar (+ and -) analog input ranges are possible.

Unlike DACs, ADCs require two additional control lines—one called *START* and the other called *End of Conversion*, or *EOC*. A *START* pulse must be sent to the converter whenever a conversion is required. Once the ADC receives the *START* signal, it takes from 20 microseconds to several milliseconds to perform the conversion, depending on the technique used for the conversion process. This is relatively slow, compared to the operation of the system CPU. Consequently, the CPU cannot simply supply a *START* pulse and then read the converted value from the converter outputs. Some type of handshake

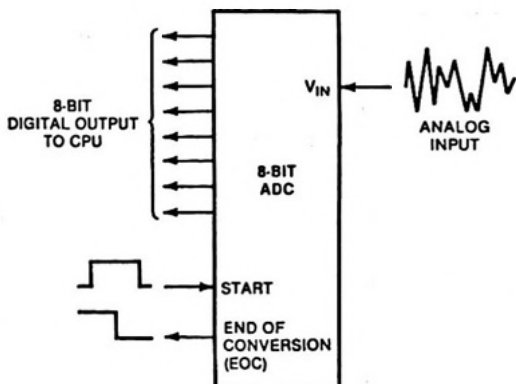


Figure 5-23
A typical 8-bit ADC device.

must be provided to tell the CPU that the ADC has completed the conversion. This is the function of the End-of-Conversion, or EOC, control line on the ADC. The EOC line is an output line from the ADC that indicates when the converter has completed a conversion. Once the CPU receives this signal, it can read the converted value from the ADC outputs and start a new conversion if desired.

The two control lines just discussed may have other names (depending on the manufacturer), but their function is the same for all converters. The START line is sometimes called *Initiate Conversion (IC)*, *CONVERT*, and *RUN*. The End-of-Conversion line is sometimes called *Data Valid (DV)*, *DONE*, *STATUS*, and *BUSY*.

Fig. 5-24 shows an 8-bit ADC connected to the PIA interface developed in Chapter 4. We have chosen to connect the ADC to port A of the PIA. Thus, you must configure port A for input handshaking as discussed in Chapter 4. To start the conversion, you simply read port A. Recall that reading port A will cause the CA2 output line of the PIA to go low for one CPU cycle. This is sufficient to start the conversion. When the ADC completes the conversion process, it will activate the CA1 input line of the PIA to indicate that the conversion has been completed. Recall that the active CA1 input line can be detected by polling the bit 7 status of the port A control register, or an interrupt can be generated via the $\overline{\text{IRQA}}$ line of the PIA. In any event, once the active CA1 input line is detected, the converted value can be obtained at any time by reading the port A data register of the PIA. Furthermore, by reading port A, another conversion is started via the CA2 output line.

Some ADCs provide an additional feature of *analog multiplexing*.

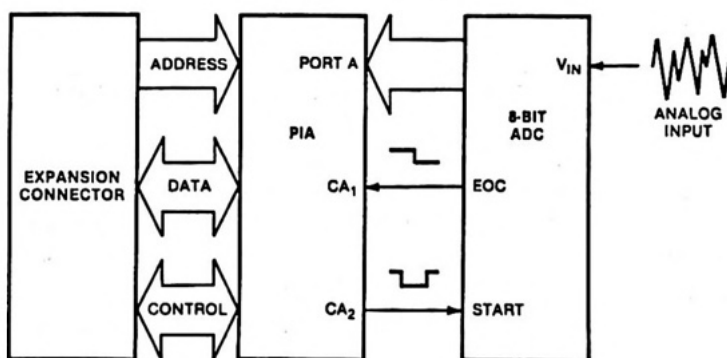


Figure 5-24
An 8-bit ADC connected to the PIA interface.

Analog multiplexing allows more than one analog signal to be sampled, or converted, by the same ADC. An ADC with an internal 4-to-1 analog multiplexer is shown connected to the PIA interface in Fig. 5-25. The internal multiplexer selects one of four analog inputs and passes the selected input to the converter portion of the ADC. In this case, two *channel-select* input lines are provided on the ADC to select one of the four analog inputs. You can provide channel selection by connecting the ADC channel-select lines to any two unused PIA port lines. We have chosen to use PB0 and PB1 in Fig. 5-25. You must configure these lines for output and then select any one of the four analog inputs by writing the proper 1s and 0s to the PB0 and PB1 data register bits. Once the desired channel is selected, the conversion process is controlled via the ADC START and EOC lines as discussed previously. Many existing ADCs provide this internal multiplexing capability. If more analog signals must be monitored, larger internal multiplexers are required. Single-chip converters with internal 8-to-1 multiplexers are now common.

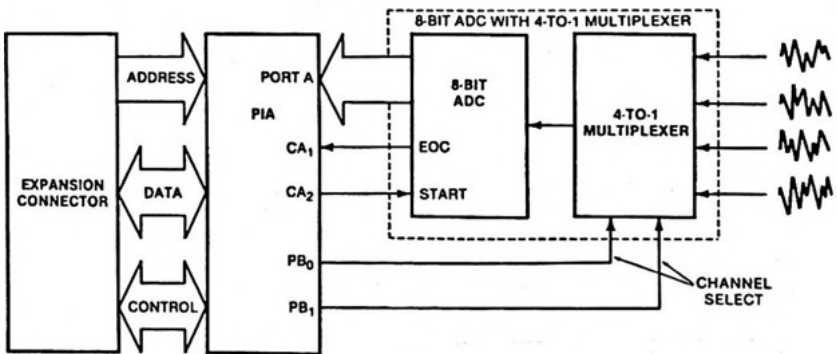


Figure 5-25 An 8-bit ADC with a 4-to-1 internal multiplexer connected to the PIA interface.

There are applications in which resolution of greater than eight bits is required for measurement of unknown voltages. Converters with greater resolutions have more than eight output bits, so both PIA ports must be used for the interface. A 12-bit ADC is shown connected to the PIA interface in Fig. 5-26. Here, we have connected the eight least-significant data lines (D0–D7) to port A, and the four most-significant data lines (D8–D11) to port B. The START and EOC control lines are connected to the CB1 and CB2 control lines of the PIA and perform the same functions as they did in the 8-bit ADC

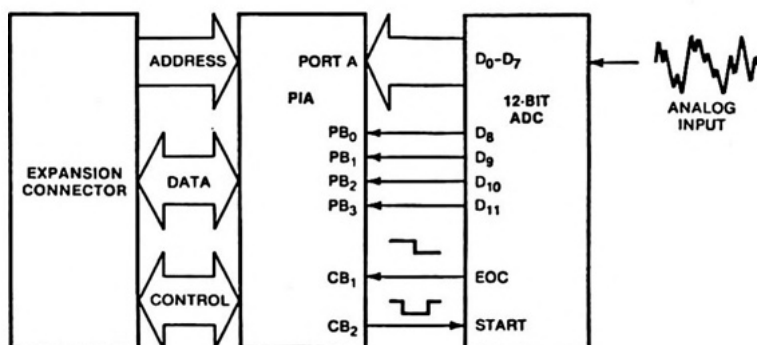


Figure 5-26
A 12-bit ADC connected to the PIA interface.

interface circuit. Since some of the 12 bits will be input from one port, while the remaining bits are input from another port, the resulting values must be combined so that the data values are between 0 and 4096, the range possible with a 12-bit ADC. The computer treats the four most-significant data bits obtained from port B as if they represent a separate data value between 0 and 15. To *scale* these bits to their actual positional weight within the 12-bit data word, you must multiply them by 256, and then add them to the 8-bit data value obtained from port A. For example, suppose the PIA has been configured properly for a data input operation from the 12-bit ADC shown in Fig. 5-26. As before, we will assume that the port A data register is located at address 65344 and the port B data register is located at address 65346.

The following BASIC *scaling* program can be used to scale the 12-bit data input value obtained from ports A and B:

```
100 A = PEEK (65344)
110 B = PEEK (65346) AND 15
120 X = (B * 256) + A
```

Line 100 in this program reads the lower eight bits of the 12-bit data value from port A, and line 110 reads port B. However, you are only interested in the four least-significant data bits of port B. You must, therefore, *mask-out*, or zero-out, the four most-significant port B data bits such that they are not used by the computer in interpreting the data input value. This is accomplished by performing a logical AND operation between the port B data and a *mask value*. The mask value is an 8-bit constant that has 0s in all the unwanted bit locations and

1s in the desired bit locations. Since we are only interested in the four least-significant data bit locations of port B, the mask value must be 0000 1111₂, or 15₁₀. The last line in the scaling program (line 120) multiplies the masked port B data value by 256 and adds the result to the 8-bit data value obtained from port A. The final result, X, will be the decimal equivalent of the 12-bit data input provided by the ADC circuit shown in Fig. 5-26.

Now, you have all the hardware and software ingredients required to interface an external ADC to your computer. Let's look at several actual ADC devices before performing the ADC interfacing experiments in Chapter 6.

The ADC0804LCN ADC Interface

The National Semiconductor ADC0804LCN is an 8-bit successive approximation ADC. The circuit in Fig. 5-27 shows how to connect the ADC0804LCN to the Color Computer via the PIA interface developed in Chapter 4. We have chosen to use port A of the PIA. A read-port-A operation will start a conversion by activating the CA2 output line of the PIA. When the conversion is completed, the ADC0804LCN will activate (low) the CA1 input control line of the PIA. The active CA1 input can be detected by polling bit 7 of the port A control register;

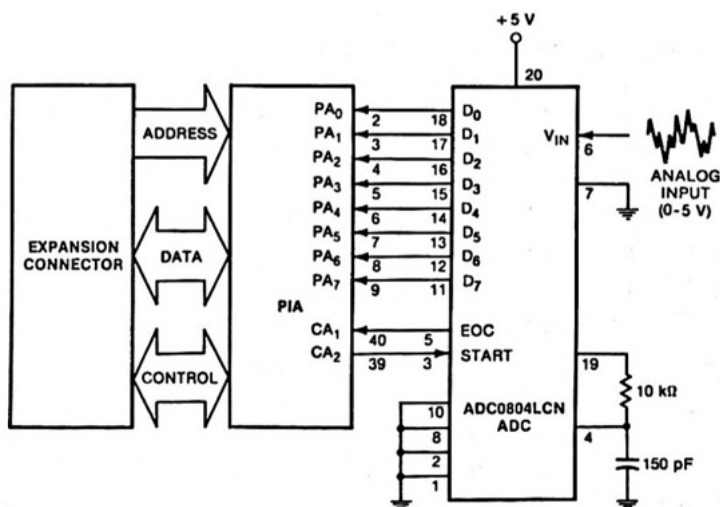


Figure 5-27 The National Semiconductor ADC0804LCN 8-bit ADC connected to the PIA interface.

or interrupt control can be used by connecting the $\overline{\text{IRQA}}$ output line of the PIA to the $\overline{\text{NMI}}$ interrupt input line available on the Color Computer expansion connector. Programmed I/O, or polling, control of the conversion process will be demonstrated when you interface this ADC to the Color Computer in Chapter 6. After the End of Conversion, or EOC, signal is detected, you simply read the port A data register to obtain the converted value. The circuit in Fig. 5-27 will convert analog input voltage values within a range of 0 to 5 volts. The resulting digital value will range from 0000 0000₂ to 1111 1111₂, or 0₁₀ to 255₁₀. Consequently, the ADC0804LCN has a resolution of $(1/256) \times 5$ volts, or 19.5 millivolts. For higher resolution, an ADC of greater than eight bits must be used.

You could measure higher voltage ranges, but you must use external attenuation on the analog input signal to bring it within the 0 to 5-volt input range of the ADC. Lower voltage ranges can be measured by applying an external reference voltage to the ADC0804LCN. Consult the ADC0804LCN data sheet provided in Appendix C for details on this option.

The ADC0809LCN ADC Interface

The ADC0809LCN is also manufactured by National Semiconductor. It is an 8-bit successive approximation ADC very similar to the ADC0804LCN just discussed. However, the ADC0809LCN includes an internal 8-to-1 analog multiplexer. The ADC0809LCN is shown connected to the PIA interface in Fig. 5-28. The conversion process is controlled via port A of the PIA as before. However, an analog input channel must be selected before a conversion is started. Three of the port B lines are configured for output and used to select one of eight analog inputs. The logic written to the PB0–PB2 data register bits will determine which channel is selected, as shown in Table 5-5. The remaining features of the ADC0809LCN are similar to those of the ADC0804LCN.

The ICL7109 ADC Interface

The ICL7109 is a 12-bit dual-slope ADC manufactured by Intersil. Consequently, the ICL7109 has a higher resolution and greater noise immunity than the two 8-bit successive approximation ADCs just discussed. However, it is much slower than the two previous devices. The ICL7109 is shown connected to the PIA interface in Fig. 5-29.

Given an analog input signal, the converter will provide 12 bits of output data to the PIA. We have chosen to connect the eight least-

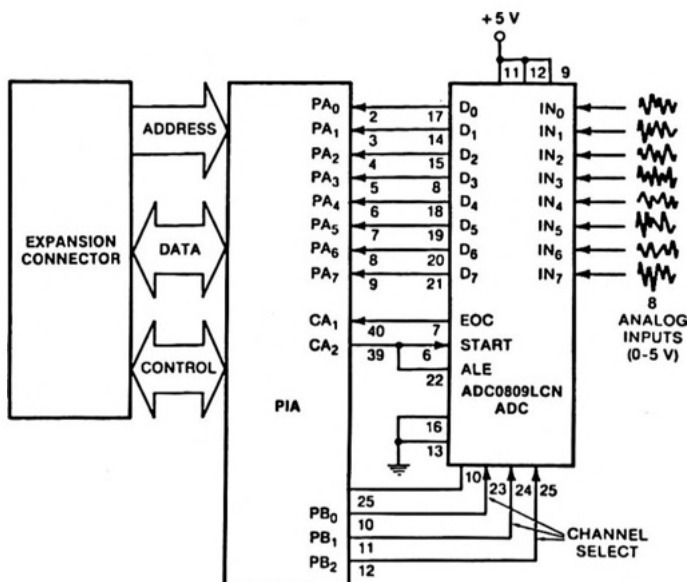


Figure 5-28 The National Semiconductor ADC0809LCN 8-bit ADC with internal 8-to-1 multiplexing connected to the PIA interface.

Table 5-5. Analog Input Channel Selection Provided by the PIA Interface Circuit in Fig. 5-28

Port B Logic			Analog Channel Selected
PB2	PB1	PB0	
0	0	0	IN0
0	0	1	IN1
0	1	0	IN2
0	1	1	IN3
1	0	0	IN4
1	0	1	IN5
1	1	0	IN6
1	1	1	IN7

significant data lines (D0–D7) to port A of the PIA, and the four most-significant data lines (D8–D11) to port B of the PIA. The ICL7109 also provides a *polarity* output line (*POL*) to indicate input voltage polarity. We have connected this to the PB4 line of the PIA. The circuit in Fig. 5-29 has been designed so that the analog input range of the converter will be ± 5 volts. An *overrange* output line (*OR*)

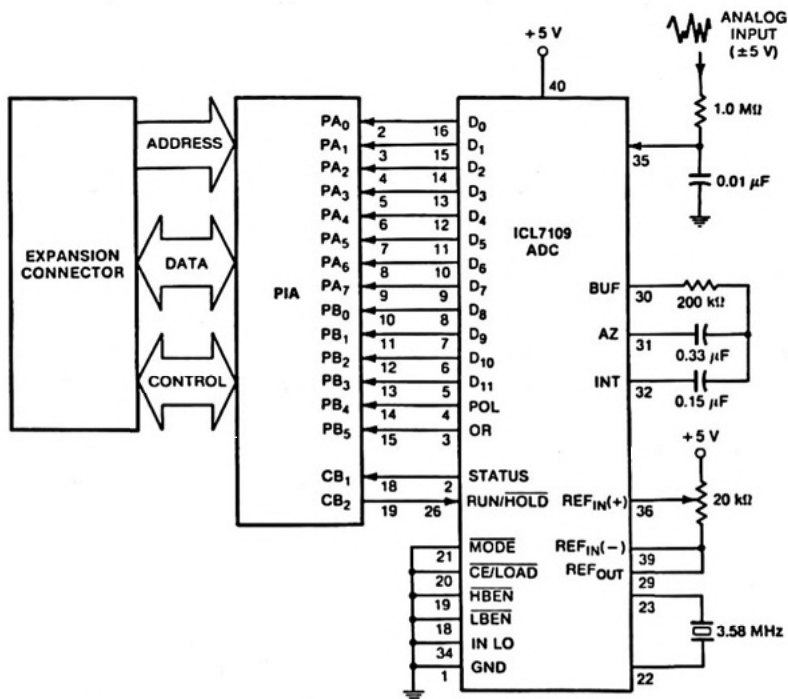


Figure 5-29 The Intersil ICL7109 8-bit dual-slope ADC connected to the PIA interface.

is also provided by the ICL7109 to indicate when the analog input voltage is beyond the ± 5 -volt input range. We have chosen to connect this line to PB5 of the PIA.

The circuit in Fig. 5-29 uses the CB1 and CB2 control lines of the PIA to control the conversion process. If you write the value 61₁₀ to the port B control register when the PIA is initialized, CB2 will be designated as an output line and held in the logic high state. This allows the converter to "RUN" and provide continuous conversions. This is called the *continuous conversion mode* of the ICL7109. The circuit in Fig. 5-29 will provide approximately 7.5 conversions per second using the 3.58-MHz crystal shown. At the end of each conversion, the STATUS output line of the converter is activated (low). Consequently, the CB1 input line of the PIA is activated and the end-of-conversion signal can be detected. The converted value is then obtained by reading the port A and port B data registers of the PIA. The

12-bit value must then be scaled by software as we discussed earlier in this section.

Now, writing the value 53_{10} to the port B control register causes the CB2 output line of the PIA to be held *low* and, thus, "HALT" the continuous conversion process. A conversion can then be started when desired by writing the value 61_{10} to the port B control register. This allows for conversions to be initiated under software control and is called the *one-shot conversion mode*. The one-shot conversion mode is frequently used when a single data value is to be obtained, and when it is necessary to obtain the most recently converted value. Furthermore, the one-shot mode is useful when assembly language programs are going to be used to control the converter. You might want to consult the ICL7109 data sheet available from Intersil for other methods of controlling the conversion process.

That completes our discussion of analog-to-digital converters. We suggest that you go to Chapter 6 at this time and perform the analog-to-digital converter experiments.

A LIST OF REPRESENTATIVE DAC AND ADC MANUFACTURERS

Advanced Micro Devices
901 Thompson Place
Sunnyvale, CA 94086

Data Translation, Inc.
100 Locke Drive
Marlboro, MA 01752

Analog Devices, Inc.
P.O. Box 280
Norwood, MA 02062

Datel-Intersil, Inc.
11 Cabot Boulevard
Mansfield, MA 02048

Analogic Corp.
Audubon Road
Wakefield, MA 01880

Hybrid Systems Corp.
22 Linnell Circle
Billerica, MA 01821

Beckman Instruments, Inc.
Advanced Electro-Products Div.
2500 Harbor Boulevard
Fullerton, CA 92634

Intech, Inc.
282 Brokaw Road
Santa Clara, CA 95050

Burr-Brown
P.O. Box 11400
Tucson, AZ 85734

Motorola Semiconductor Products,
Inc.
P.O. Box 20912
Phoenix, AZ 85036

National Semiconductor Corp.
2900 Semiconductor Drive
Santa Clara, CA 95051

Signetics Corp.
811 E. Arques Avenue
Sunnyvale, CA 94086



6 Experiments

Before you can interface your computer expansion port to the "outside world," you must extend the expansion connector signals outside of the computer case and provide a convenient means of connecting external devices to the expansion connector signals. We have developed a simple solderless breadboard that is inserted into the Color Computer expansion connector to perform the expansion connector interfacing experiments. The breadboard provides an easy means of connecting external devices, like the PIA, to the expansion port of the computer.

In the first part of this chapter, we will show you how to construct your own expansion connector breadboard. Once the breadboard is completed, you will use it to construct the PIA interface we developed in Chapter 4. Then, you will provide parallel data I/O for external devices such as switches, LEDs, DACs, and ADCs as we discussed in the previous chapters of this book.

Note: If you prefer *not* to construct the expansion breadboard yourself, you can obtain a completely assembled and tested version by requesting ordering information at the following address:

BREADBOARD
P.O. Box 265
Randolph, NY 14772

All of the expansion connector experiments in this chapter require the use of the expansion connector breadboard. Two experiments (Experiment 3 and Experiment 5) demonstrate the use of the internal

DAC and ADC capabilities of the Color Computer and, therefore, do not require the use of the expansion connector breadboard. These two experiments can be performed without any additional components.

OBJECTIVES

In this chapter, you will:

- Construct a solderless breadboard for the Color Computer expansion connector.
- Construct the PIA expansion connector interface circuit developed in Chapter 4.
- Input binary data and control information to the computer via the PIA interface.
- Output binary data from the computer via the PIA interface.
- Output analog signals from the internal Color Computer DAC.
- Interface an external 8-bit DAC to the Color Computer.
- Input analog signals to the internal Color Computer ADC.
- Interface an external 8-bit ADC to the Color Computer.
- Construct a digital voltmeter using an ADC interface.

THE EXPANSION CONNECTOR BREADBOARD

The expansion connector breadboard is pictured before its assembly in Fig. 6-1 and after its assembly in Fig. 6-2. Notice that both top (labeled component side) and bottom views are provided in each picture. You will need the following parts to construct your breadboard:

- 1 No. 4609 Vector peripheral interface board.
- 11 8-pin solderless connector blocks.
- Connecting wire.

You can obtain the 4609 Vector board from almost any electronic component distributor. We obtained our board from Jameco Electronics, 1355 Shoreway Rd., Belmont, CA 94002. The 8-pin solderless connector blocks are a bit harder to find. We purchased ours from the Heath/Zenith Co., Benton Harbor, MI 49022 as part No. 432-973.

Once you have obtained the breadboard and solderless connector blocks, all you need is connecting wire (No. 20-22), a 25-watt pencil-type soldering iron, a little super-glue, and a lot of patience. We suggest that you lay out the solderless connector blocks on the

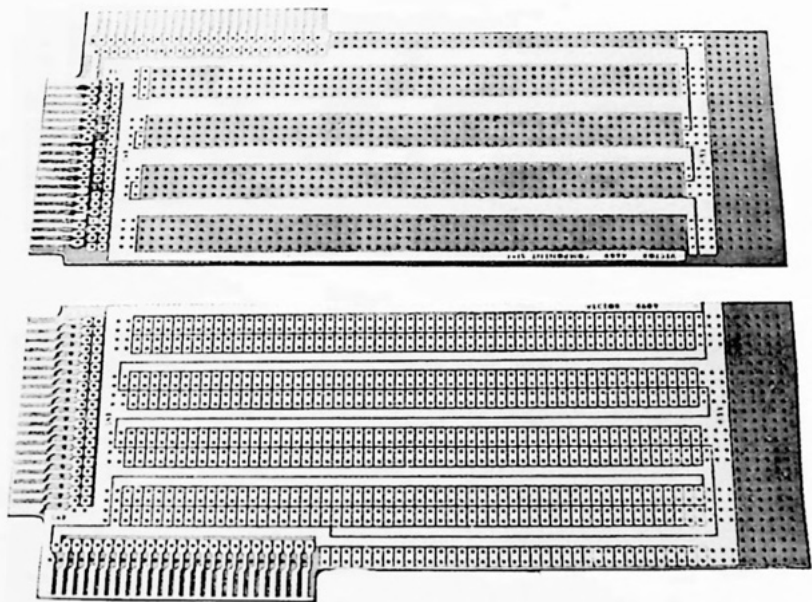


Figure 6-1
The expansion connector breadboard (Vector No. 4609) before assembly.

breadboard to correspond to the Color Computer expansion connector signals as shown in Fig. 6-3. Then, construct the breadboard as follows:

Step 1 — Cut off the plastic feet on the bottom of the 8-pin solderless connector blocks using a sharp pocket knife or small pair of cutting dikes. Trim, or file, the feet flush to the bottom of the connector block.

Step 2 — Insert each solderless connector block into its relative position on the blue *nonconductive* portion on the top (component side) of the breadboard (see Fig. 6-3). Notice that the solderless connector blocks are placed on the same side, but between the ground (GND) conductive paths of the breadboard. The connector block pins must be inserted into the breadboard holes and extend through to the other side for soldering. You might have to trim or realign the metal pins so that they fit through the breadboard holes properly.

Step 3 — Remove any one of the connector blocks and place a small drop of super-glue on the bottom of the block. Reinsert the block and hold it a few seconds until the glue is dry. Repeat the above gluing procedure for each connector block on the breadboard.

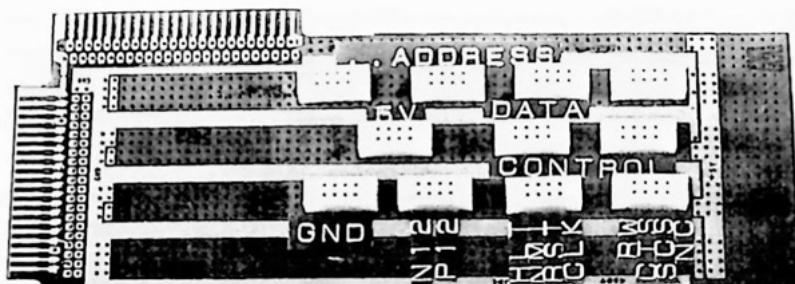


Figure 6-2
The expansion connector breadboard after assembly.

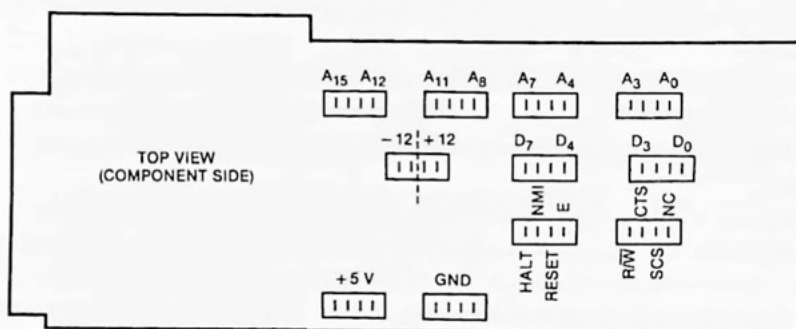


Figure 6-3
Suggested expansion connector breadboard layout.

Step 4 — The internal expansion connector pins of the Color Computer are laid out as shown in Fig. 6-4. Notice that the odd pins are on the top of the expansion connector and the even pins are on the bottom of the connector. Consequently, the edge connector pins on the top of the breadboard will make contact with the odd pins of the

Pin No.	Signal Name	Description
1	- 12 V	- 12 volts (100 mA)
2	+ 12 V	+ 12 volts (300 mA)
3	HALT	Halt Input to the CPU
4	NMI	Nonmaskable Interrupt to the CPU
5	RESET	Main Reset and Power-Up Clear Signal to the System
6	E	Main CPU Clock (0.89 MHz)
7	Q	Quadrature Clock Signal which Leads E
8	CART	Interrupt Input for Cartridge Detection
9	+ 5 V	+ 5 volts (300 mA)
10	D ₀	Data Bit 0
11	D ₁	Data Bit 1
12	D ₂	Data Bit 2
13	D ₃	Data Bit 3
14	D ₄	Data Bit 4
15	D ₅	Data Bit 5
16	D ₆	Data Bit 6
17	D ₇	Data Bit 7
18	R/W	Read/Write Signal
19	A ₀	Address Bit 0
20	A ₁	Address Bit 1
21	A ₂	Address Bit 2
22	A ₃	Address Bit 3
23	A ₄	Address Bit 4
24	A ₅	Address Bit 5
25	A ₆	Address Bit 6
26	A ₇	Address Bit 7
27	A ₈	Address Bit 8
28	A ₉	Address Bit 9
29	A ₁₀	Address Bit 10
30	A ₁₁	Address Bit 11
31	A ₁₂	Address Bit 12
32	CTS	Cartridge Select Signal
33	GND	Signal Ground
34	GND	Signal Ground
35	SND	Sound Input
36	SCS	Spare Select Signal
37	A ₁₃	Address Bit 13
38	A ₁₄	Address Bit 14
39	A ₁₅	Address Bit 15
40	SLENB	Input to Disable Device Selection

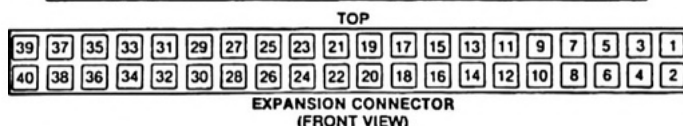


Figure 6-4 Pin assignments for the expansion connector.

expansion connector, and the edge connector pins on the bottom of the breadboard will make contact with the even pins of the expansion connector. Therefore, the edge connector pins of the breadboard must be assigned as shown in Fig. 6-5 to connect to the proper signals in the Color Computer.

Step 5 — Using a sharp pocket knife or razor blade, break the conductive path between pin 40 on the edge connector (see Fig. 6-5) and the ground bus of the breadboard. This will protect against possible breadboard shorts when performing the experiments.

Step 6 — Solder the edge connector pins of the breadboard to the solderless connector block pins by following the breadboard layout

diagram in Fig. 6-3 and the edge connector pin assignment diagram in Fig. 6-5. We suggest you proceed as follows:

1. Cut each wire to length *before* you solder it. Then, trim approximately $\frac{1}{16}$ to $\frac{1}{8}$ inch of insulation from both ends of the wire.
2. Run all connecting wires along the bottom of the breadboard.
3. *Solder all of the even lines first.* When you are finished, the even lines will be located below the odd lines. Consequently, solder-

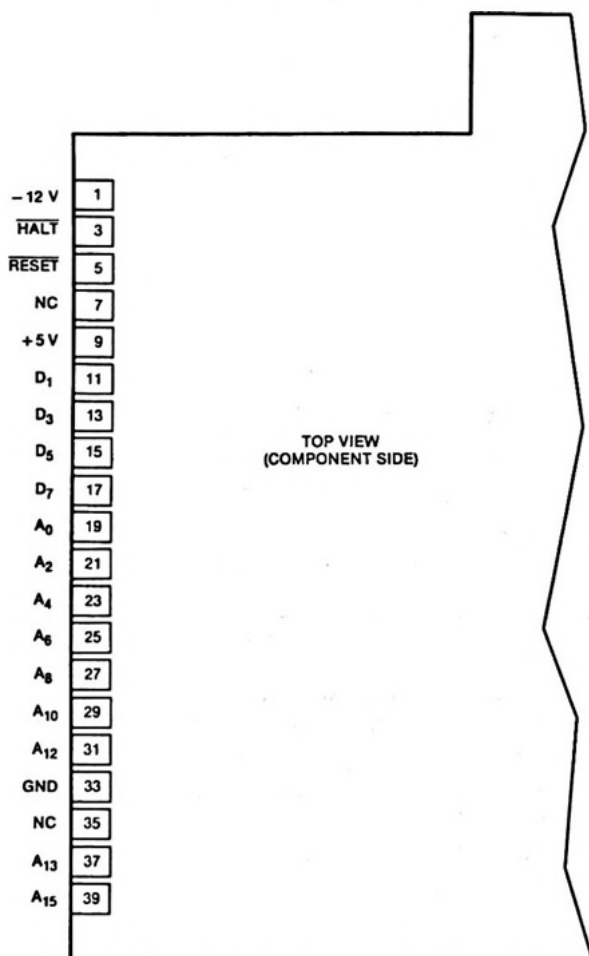
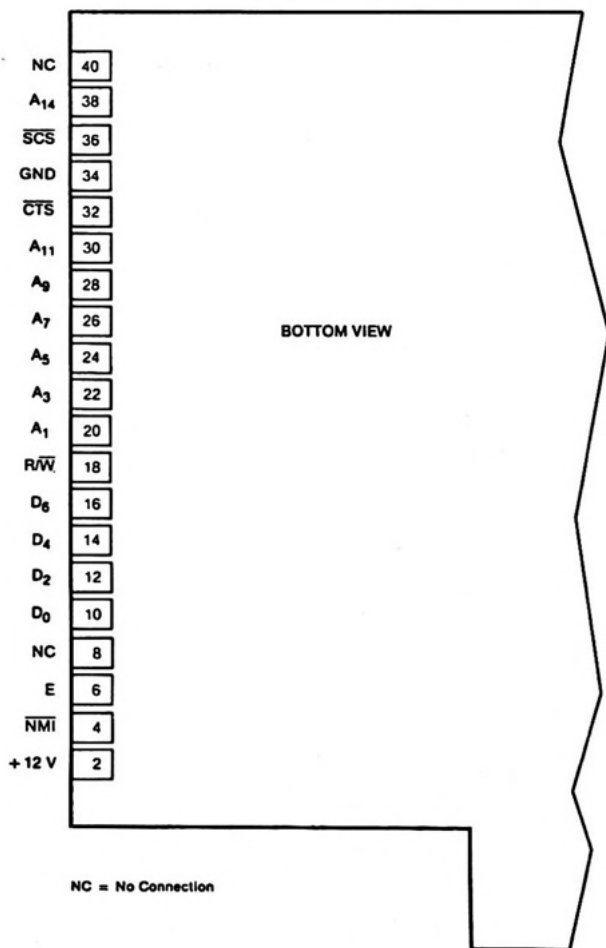


Figure 6-5
Pin assignments for the breadboard edge connector.

ing the even lines first will prevent you from accidentally burning any of the wire insulation when you solder the odd lines.

4. When you solder to an *even edge-connector pin*, the solder connection will be made on the bottom of the breadboard.
5. When you solder to an *odd edge-connector pin*, you must insert the wire through the bottom of the breadboard and make the solder connection on the top of the breadboard.



6. After you solder a line, use a VOM to check for good continuity between the edge connector pin and its respective connector block socket. You can insert a small wire into the socket to provide a connection point for the VOM.
7. After you have completed all the soldering connections, use a VOM to check for possible shorts between the lines. If a short is found due to excessive solder, you can use a pocket knife to trim away the excessive solder or cut through the shorted path. Otherwise, you must resolder the line(s).

Now that you have completed your breadboard, you are ready to perform the expansion connector interfacing experiments.

EXPERIMENT 1: DATA INPUT AND THE PIA INTERFACE

Purpose

To demonstrate binary data input via the PIA interface circuit developed in Chapter 4.

Equipment

- 1 Color Computer
- 1 Expansion Connector Interface Breadboard
- 1 Solderless Breadboard
- 1 6820 or 6821 PIA
- 1 8-Bit Binary Switch Block
- 8 1000-Ohm Resistors
- Hook-Up Wire

Schematic Diagram (Fig. 6-6)

Program

```
10 POKE 65344,0
20 POKE 65345,4
30 A = PEEK (65344)
40 PRINT A
50 GOTO 30
```

Procedure

Step 1 — Construct the PIA interface circuit shown in Fig. 6-6 using the expansion connector interface breadboard and an extra solderless breadboard. Locate the PIA and binary input switches on the solder-

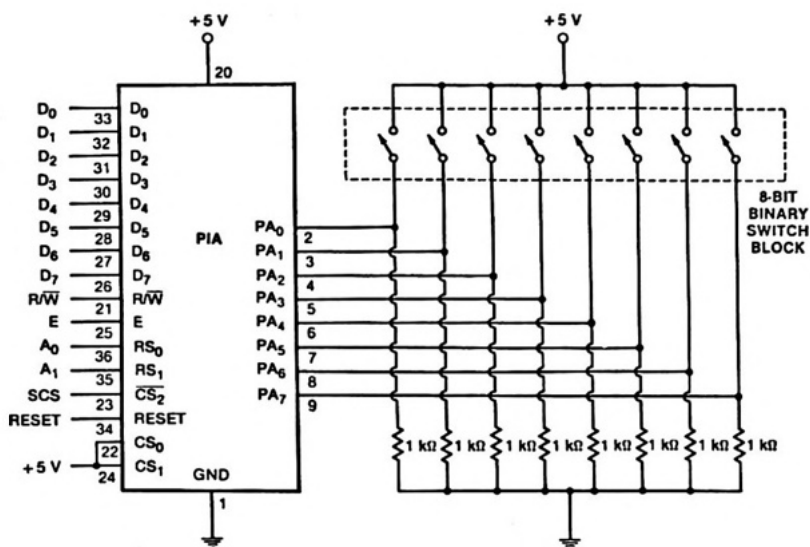


Figure 6-6
Schematic diagram for Experiment 1.

less breadboard and connect hook-up wire to the expansion connector interface breadboard as shown in Fig. 6-7. We suggest that you remove the expansion connector breadboard from the computer while making connections to it. This will help to prevent damage to the breadboard and the internal expansion connector of the computer. Furthermore, you must use caution when handling the PIA since it is a MOS device and sensitive to static electricity. Make sure you are grounded with a ground strap, and avoid touching the PIA pins. Also, when not using the PIA, be sure to place it in conductive foam or an antistatic tube.

Step 2 — Make sure the computer power is turned off, and carefully insert the expansion connector interface breadboard into the expansion connector on the right side of the computer (see Fig. 6-7).

Step 3 — Turn the computer power on and enter the given data input program.

Step 4 — Run the program and observe the TV monitor. You should observe a decimal value between 0 and 255 being displayed.

Step 5 — Change the binary input value to the PIA by changing any of the binary switch positions. You should observe a change in the decimal value being displayed by the computer. If no change is ob-

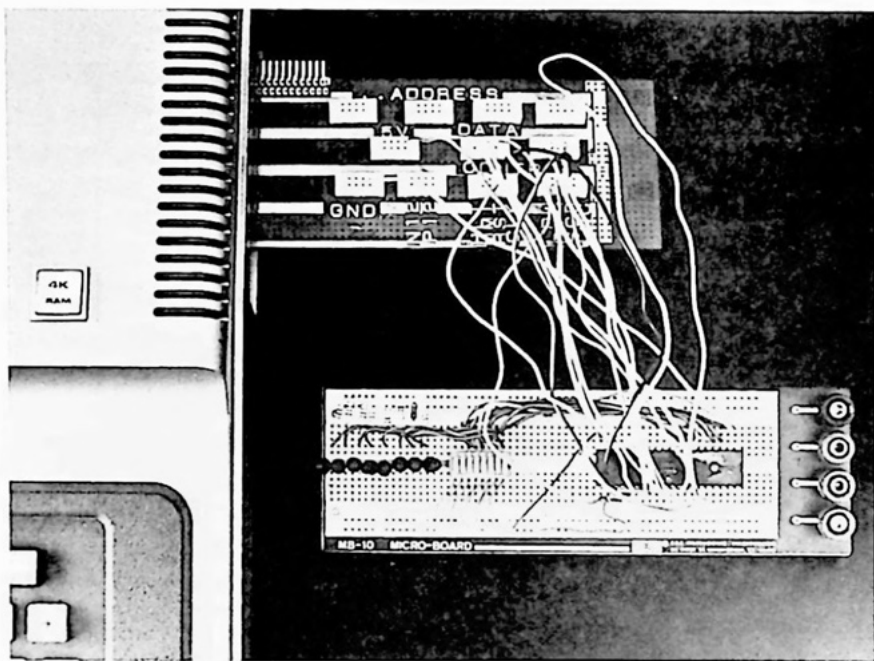


Figure 6-7
The PIA interface connected to the expansion connector breadboard for binary data input.

served, the circuit is not working properly and you must check your circuit connections and examine your program.

Step 6 — Put all eight binary switches in their logic 0, or open position. You should observe the value 0 being displayed by the computer. Now, put all the binary switches in their logic 1, or closed position. You should observe the value 255 being displayed by the computer.

Step 7 — Set the binary input switches for any random input value. Determine the input value by observing the switch positions. You should observe the decimal equivalent of the binary input value being displayed by the computer.

Step 8 — Repeat Step 7 for several different switch settings. You can practice converting from binary to decimal by observing the binary switch positions and observing the decimal equivalent value displayed by the computer.

Discussion

You have constructed the PIA interface circuit that we developed for the Color Computer expansion connector in Chapter 4. Consequently,

the PIA registers are assigned to addresses 65344 through 65347 as shown in Table 6-1. The eight binary logic switches are connected to port A of the PIA via eight pull-down resistors. Thus, an open switch will cause a logic 0 to be present on its respective port A input line, and a closed switch will cause a logic 1 to be present on its respective port A input line.

The data input program first configures port A of the PIA for input as discussed in Chapter 4. The variable *A* is then assigned to the data input value present on the port A data lines due to the binary switch settings. The port A data register is read by the PEEK instruction and displayed on the TV monitor by the PRINT instruction. The PEEK and PRINT operations are then repeated such that any change in the binary switch settings will be detected and displayed by the computer.

Procedure (continued)

Step 9 — Change the data input program by *adding* the following lines:

```

45 IF A = 128 THEN 100
100 X = RND (8)
110 Y = RND (255)
120 CLS(X)
130 SOUND Y,1
140 GOTO 30

```

Step 10 — Run the program and observe the display value. It should be the decimal equivalent of the binary switch setting as before.

Step 11 — Set all the switches to their logic 0, or off position. Now, set the most significant switch, which represents data line D7, to its logic 1, or on position. You should observe and hear the results. The input value 1000 0000₂, or 128₁₀, causes the computer to generate a series of random colors and sounds.

Step 12 — Change the input to any value except 1000 0000₂, or

Table 6-1. The PIA Register Assignments Provided by the Circuit in Fig. 6-6

Address	PIA Register Selected
65344	DRA or DDRA*
65345	CRA
65346	DRB or DDRB*
65347	CRB

*Depends on bit 4 of the respective control register.

128₁₀. The computer should now go back to displaying the corresponding decimal equivalent.

Step 13 — Change line 45 of the program so that the color/sound routine is executed for a different data input value (A). Run the program and you should find that the color/sound routine is only executed when the correct data input value is input to the computer via the binary switches and PIA interface. You should also find that the computer will not recognize any data value above 255₁₀, since the maximum input value is only eight bits.

Step 14 — Save the PIA interface connections for the next experiment.

Discussion

The lines that you added to the program in Step 9 cause the computer to “look” for a specific input value. When this value is detected, the color/sound routine is executed until a different input value is detected. This simple routine demonstrates how you can control the computer with an external event. For example, suppose the computer is being used to control an automatic garage-door opener, and the eight binary input switches provide the input code required to open the garage door. There are 2⁸, or 256, different possible switch settings, or input codes, provided by the switches. The computer can be programmed to look for a specific code and, when detected, respond by opening the garage door.

Now that you are familiar with data input, let's see how data can be output via the PIA interface (Experiment 2).

EXPERIMENT 2: DATA OUTPUT

Purpose

To demonstrate binary data output via the PIA interface circuit.

Equipment

- 1 Color Computer
- 1 Expansion Connector Interface Breadboard
- 1 Solderless Connector Breadboard
- 1 6820 or 6821 PIA
- 2 7405 Open-Collector Inverter ICs
- 8 Single-Element LEDs
- 8 1000-Ohm Resistors
- Hook-Up Wire

Schematic Diagram (Fig. 6-8)

Procedure

Step 1 — Turn the computer power off and construct the circuit shown in Fig. 6-8. (Note: If you have saved the PIA circuit from Experiment 1, you need only to disconnect the binary input switches from port A and connect the data output LEDs to port B as shown in Fig. 6-8. All other circuit connections remain the same.)

Step 2 — Insert the expansion connector interface board and turn the computer power on.

Step 3 — Enter and run the following program:

```
10 POKE 65346,255
20 POKE 65347,4
30 FOR X = 0 TO 255
40 POKE 65346,X
50 FOR I = 1 TO 200
60 NEXT I
70 NEXT X
80 GOTO 30
```

You should observe that the output LEDs provide a binary count from 0000 0000₂ through 1111 1111₂, or 0₁₀ through 255₁₀.

Discussion

The PIA interface circuit is the same as the one you used in Experiment 1 and we developed in Chapter 4. The LEDs are connected to port B of the PIA via open-collector inverters. The 7405 open-collector inverters are required to *drive* the LEDs. Damage could result to the PIA if you were to drive the LEDs directly with the PIA port lines, since the PIA is not capable of supplying the current required by the LEDs.

Open-collector devices are commonly used in computer systems to drive low-voltage, low-current devices like LEDs and reed relays. Actually, the 7405 open-collector devices “sink” the LED current to ground, but the term *driver* is more commonly used for devices of this type. Notice that the LED anodes are all connected through current-limiting resistors to the +5-volt supply. This is called a *common-anode* connection. Consequently, a 0 volt, or logic 0, level is required to illuminate a given LED. A logic 1 output on a given port B output line is inverted by the 7405 to produce the required logic 0 level to illuminate the respective LED. A logic 0 output on a port B data line is inverted to a logic 1 and will *not* cause the respective LED to illuminate.

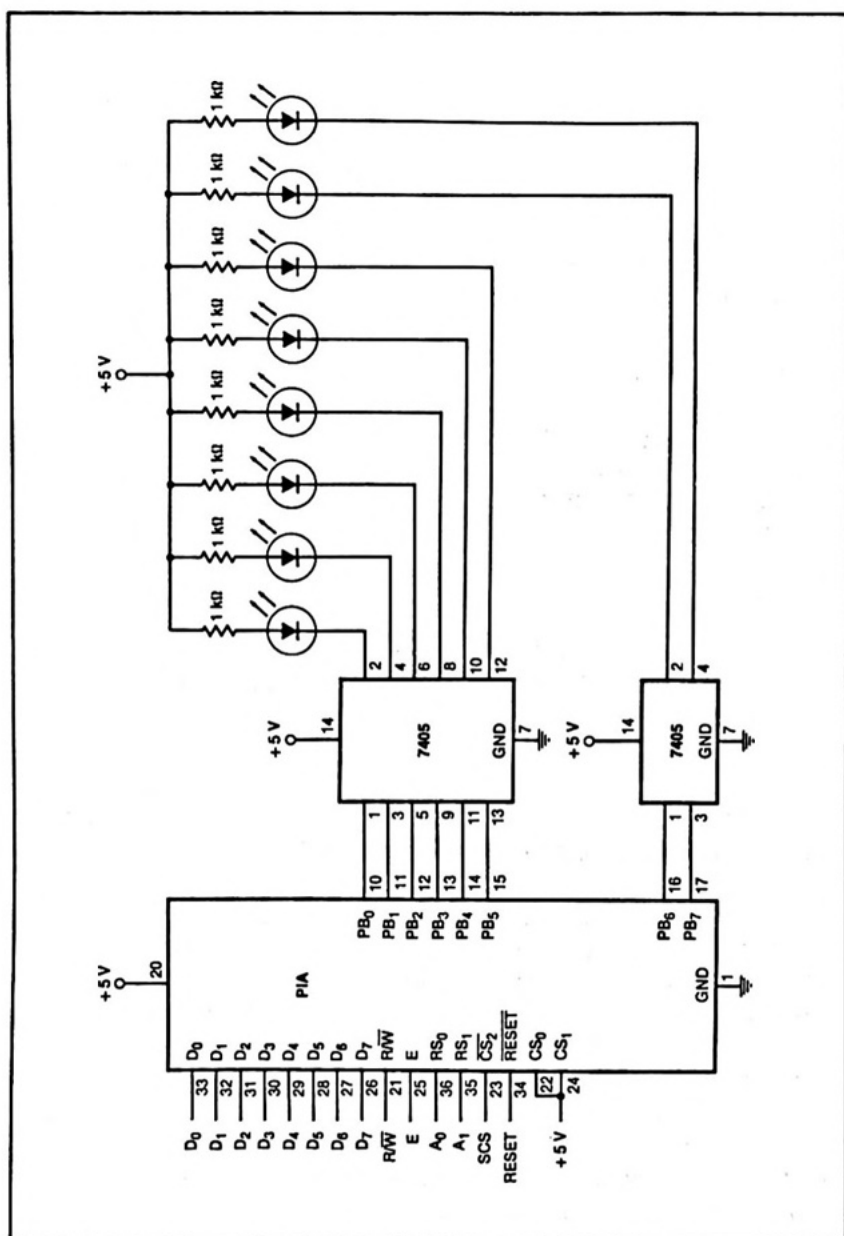


Figure 6-8
Schematic diagram for Experiment 2.

The LED display program first configures port B for output in lines 10 and 20. The variable *X* is then incremented from 0 to 255 and POKED to the port B data register each time it is incremented (lines 30–70). Each time *X* is POKED to port B, a delay is created in lines 50 and 60 such that you can observe the count. Without this delay, the count would be too fast to observe. The GOTO statement in line 80 simply repeats the count.

Procedure (continued)

Step 4 — Change line 30 as follows:

30 FOR X = 255 TO 0 STEP -1

Step 5 — Run the program and observe the LED display. You should observe that the LEDs count down from 1111 1111₂ to 0000 0000₂. Why?

Step 6 — Delete lines 50 and 60 in the program.

Step 7 — Run the program and observe the LED display. You should observe a very rapid down-count since the delay between counts has been eliminated. In fact, you should observe that the least significant data bit (D0) is counting so fast that it looks as if it is on constantly.

Step 8 — Save the PIA interface connections for Experiment 4.

EXPERIMENT 3: INTERNAL DIGITAL-TO-ANALOG CONVERSION

Purpose

To demonstrate digital-to-analog conversion using the internal 6-bit DAC of the Color Computer.

Equipment

- 1 Color Computer
- 1 Voltmeter or VOM
- 1 Oscilloscope

Diagrams (Figs. 6-9 and 6-10)

Procedure

Step 1 — Turn your computer off and connect a voltmeter or VOM to the cassette output connector pins on the back of the computer as shown in Fig. 6-9. To make contact with a connector pin, you can insert a small pin or wire lead into each respective connector pin. A doubled-over ½-watt resistor lead wire will do the job. Then, attach

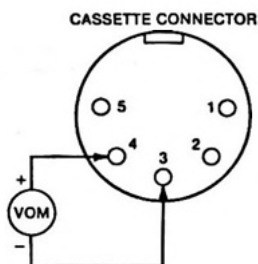


Figure 6-9
The internal DAC output is obtained by connecting a VOM or oscilloscope across pins 4 and 3 of the cassette output connector.

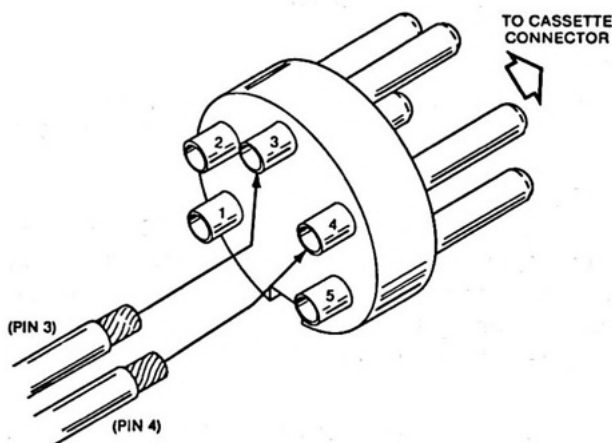


Figure 6-10
A standard 5-pin male DIN connector plug can be constructed to obtain the DAC output from the Color Computer's cassette output port.

the voltmeter leads to the inserted connector wire leads. An alternative method is to construct a 5-pin male DIN connector plug as shown in Fig. 6-10. Once constructed, the plug can be easily inserted into the cassette port of the Color Computer to provide a convenient means of connecting a voltmeter or oscilloscope to the DAC output. The 5-pin male DIN connector can be purchased at most Radio Shack stores.

Step 2 — Set the voltmeter to a DC range which will permit you to measure a 0 to 2-volt level.

Step 3 — Turn your computer on and enter the following instruction:

POKE 65312,255 **ENTER**

Step 4 — Observe the voltmeter reading. This is the maximum output voltage level available from the internal DAC. We observed a volt-

meter reading of approximately 0.8 volt. Different voltmeters might provide different readings due to their respective loading effects on the computer. We used a Simpson Model 260 VOM set on a 2.5-volt DC range.

Step 5 — Enter the following program into your computer:

```
10 FOR X = 0 TO 252 STEP 4
20 POKE 65312,X
30 FOR I = 0 TO 50
40 NEXT I
50 NEXT X
60 GOTO 10
```

Step 6 — Run the program and observe the DAC output on the voltmeter. You should observe that the DAC output gradually increases from 0 volts to its maximum level then drops back to 0 volts. The cycle is then repeated.

Step 7 — Enter the following program into your computer:

```
10 FOR X = 0 TO 252 STEP 4
20 POKE 65312,X
30 FOR I = 0 TO 50
40 NEXT I
50 NEXT X
60 FOR X = 252 TO 0 STEP -4
70 POKE 65312,X
80 FOR I = 0 TO 50
90 NEXT I
100 NEXT X
110 GOTO 10
```

Step 8 — Run the program and observe the DAC output on the voltmeter. You should observe that the DAC output gradually increases to its maximum level then gradually decreases to 0 volts. The cycle is then repeated.

Discussion

The first program that you ran was the BASIC ramp generation program discussed in Chapter 5. Increasing values from 0 and 252 are continuously stored to the internal DAC, which is located at address 65312. The DAC output, therefore, begins at 0 volts and increases to its maximum output level then falls back to 0 volts. The cycle is then repeated.

The second program is the triangular waveform generation program discussed in Chapter 5. The DAC output voltage increases from 0 volts to its maximum output level as X is incremented, and decreases from its maximum output level back to 0 volts as X is decremented. The output cycle then repeats itself.

In both of these programs, a short software delay is created each time the data value is written to the DAC. This is done so that the voltmeter meter movement can "follow" the output of the DAC. By shortening the delay loop, you can increase the output frequency. Conversely, by lengthening the delay loop, you can lower the output frequency. To do this, just change the range of values for the variable "I" in either program. Try it!

Procedure (continued)

Step 9 — Disconnect the voltmeter and connect an oscilloscope to the DAC output.

Step 10 — Run the triangular waveform generation program again and observe the oscilloscope display. You will find that you can observe a gradual increase, then decrease on the display. However, you cannot "see" the triangular waveform cycles because of the extremely low output frequency obtained from the DAC. This is due to the relatively slow execution of the BASIC program. How can the output frequency be increased so that the waveform cycles are visible on the oscilloscope? You guessed it! The output frequency can be increased by using an assembly language program to supply the data values to the DAC.

Step 11 — Type in the following command to erase the previous program from the computer memory:

NEW **ENTER**

Step 12 — Enter the following programs into the computer:

```
10 REM "ASSY. LANG. LOADER"  
*15 PCLEAR 1  
*20 CLEAR 25, 8000  
30 DATA 142,7,208,166,128,183,255,32,140,8,15,38,  
    246,32,241  
40 FOR I = 0 TO 14  
*50 READ B: POKE 8000 + I, B  
60 NEXT I  
70 END
```

***Notes:**

Do not enter line 15 unless you have an *Extended Color BASIC* system.

If you have a 4K system, you must use the value 4000 in place of 8000 in lines 20 and 50.

```
100 REM "RAMP DATA GEN."  
110 N = 2000  
120 FOR X = 0 TO 252 STEP 4  
130 POKE N,X  
140 N = N + 1  
150 NEXT X  
160 END
```

Step 13 — Run the assembly language loader program which begins at line 10.

Step 14 — Run the data generation program which begins at line 100.

Step 15 — Execute the assembly language display program by entering the following command:

***EXEC 8000** **ENTER**

**Use the value 4000 for 4K systems.*

Step 16 — Observe the oscilloscope display. You should observe the ramp (sawtooth) waveform pictured in Fig. 6-11 when the oscilloscope is adjusted properly. Measure the output waveform amplitude

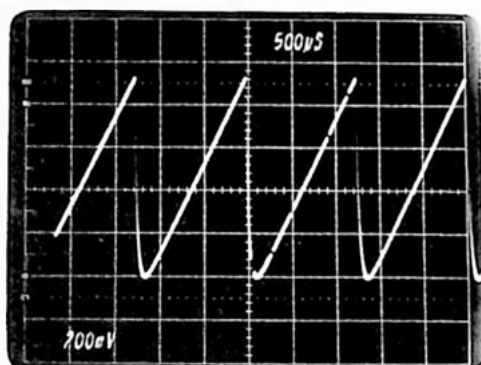


Figure 6-11 — A ramp (sawtooth) waveform generated by the internal DAC using an assembly language program.

and frequency. We measured its peak amplitude to be 0.9 volt and its frequency to be 800 Hz.

Step 17 — Reset the computer and enter the following triangular waveform data generation program beginning at line 100:

```
100 REM "TRIANGLE DATA GEN."  
110 N = 2000  
120 FOR X = 0 TO 128 STEP 4  
130 POKE N,X  
140 N = N + 1  
150 NEXT X  
160 FOR X = 128 TO 0 STEP -4  
170 POKE N,X  
180 N = N + 1  
190 NEXT X  
200 END
```

Step 18 — Run the data generation program which begins at line 100.

Step 19 — Execute the assembly language program by entering the following command:

*EXEC 8000 **ENTER**

**Use the value 4000 for 4K systems.*

Step 20 — Observe the oscilloscope display. You should observe the triangular waveform pictured in Fig. 6-12.

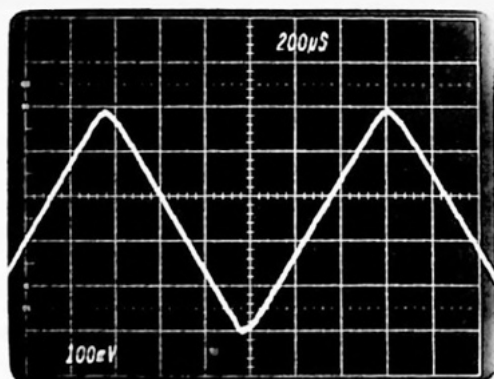


Figure 6-12
A triangular waveform generated by the internal DAC using an assembly language program.

Step 21 — Reset the computer. Then, *enter* and *run* the following square waveform data generation program which begins at line 100:

```
100 REM "SQUARE DATA GEN."  
110 FOR N = 2000 TO 2031  
120 POKE N,0  
130 NEXT N  
140 FOR N= 2032 TO 2063  
150 POKE N,255  
160 NEXT N  
170 END
```

Step 22 — Execute the assembly language display program as before by entering the command, EXEC 8000 (or 4000). Observe the oscilloscope display. You should observe the square waveform pictured in Fig. 6-13.

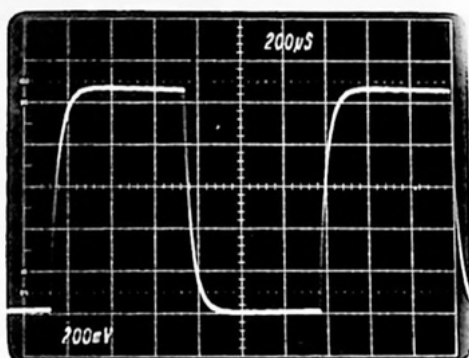


Figure 6-13
A square waveform generated by the internal DAC using an assembly language program.

Step 23 — Reset the computer. Then, change the DATA statement in line 30 of the assembly language loader program as follows:

```
30 DATA 142,7,208,166,128,183,255,32,140,8,131,38,  
246,32,241
```

Step 24 — Run the assembly language loader program which begins in line 10.

Step 25 — Enter and run the following sine waveform data generation program which begins at line 100:

```
100 REM "SINE DATA GEN."  
110 N = 2000
```

```

120 FOR X = 0 TO 360 STEP 2
130 Y = 128 + INT(127*(SIN(X*0.017453)))
140 POKE N,Y
150 N = N + 1
160 NEXT X
170 END

```

Step 26 — Execute the assembly language program as before by entering the command, EXEC 8000 (or 4000). Observe the oscilloscope display. You should observe the sine waveform pictured in Fig. 6-14.

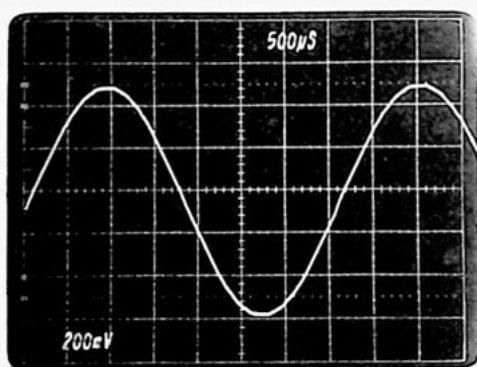


Figure 6-14
A sine waveform generated by the internal DAC using an assembly language program.

Discussion

You have used an assembly language routine to provide data values to the internal DAC to generate the preceding waveforms. The BASIC loader routine which begins at line 10 loads the assembly language display routine in memory locations 8000 through 8014 (4000 through 4014 for 4K systems). The BASIC data generation program which begins at line 100 generates the data values for the desired waveform. The data values are stored in memory locations 2000 through 2063 for the ramp, triangular, and square waveforms. The data values for the sine waveform are stored in memory locations 2000 through 2179, since the sine program generates 180 data values. When you execute the assembly language routine, the previously generated data values are simply written to the internal DAC located at address 65312 to provide the corresponding analog output waveform. You might want to refer back to the text material in Chap-

ter 5 for a more detailed discussion of these waveform generating programs.

Procedure (continued)

Step 27 — Now, just for fun, reset the computer and enter the following data generation program:

```
100 REM "RANDOM DATA GEN."  
110 FOR N = 2000 TO 2179  
120 X = RND (255)  
130 POKE N,X  
140 NEXT N  
150 END
```

Step 28 — Run the data generation program which begins at line 100.

Step 29 — Execute the assembly language display routine as before by entering the command, EXEC 8000 (or 4000). Observe the oscilloscope display. You should observe a display of the Color Computer's random number generator similar to that pictured in Fig. 6-15.

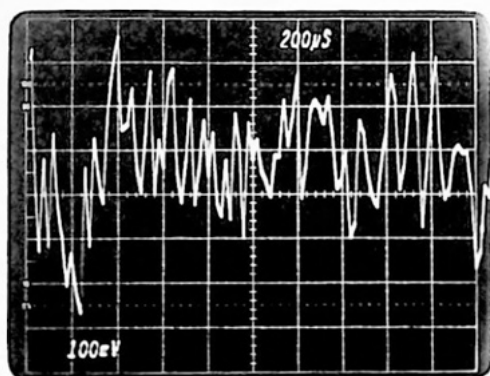


Figure 6-15
A random waveform generated by the internal DAC using the random number generator of the computer.

EXPERIMENT 4: DAC INTERFACING

Purpose

To connect an external 8-bit DAC to the expansion connector interface and demonstrate digital-to-analog conversion using the external DAC.

Equipment

- 1 Color Computer
- 1 Voltmeter or VOM
- 1 Oscilloscope
- 1 Expansion Connector Breadboard
- 1 Solderless Connector Block
- 1 Signetics NE5018 8-Bit Digital-to-Analog Converter
- 1 PIA Interface Circuit from Experiment 2
- 1 0.01-Microfarad Capacitor
- 1 50-Picofarad Capacitor
- 1 1000-Picofarad Capacitor
- 1 1N914 Signal Diode
- 1 4700-Ohm Resistor
- Hook-Up Wire

Schematic Diagram (Fig. 6-16)

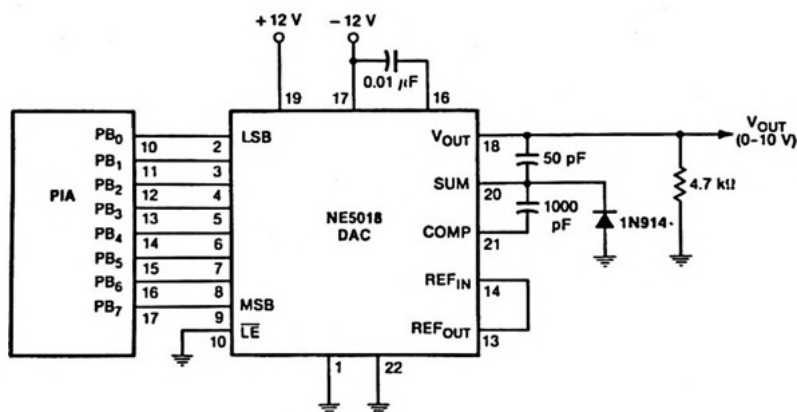


Figure 6-16
Schematic diagram for Experiment 4.

Procedure

Step 1 — Turn the computer power off and connect the NE5018 DAC to the PIA interface as shown in Fig. 6-16. (*Caution:* The NE5018 DAC is a MOS device and must be protected from static electricity. Make sure you are grounded and avoid touching the pins of the NE5018. When you are not using this device, store it in conductive foam or an antistatic tube.)

Step 2 — Connect a voltmeter between the DAC output (pin 18)

and ground. Set the voltmeter to a range which will allow for a 10-volt measurement.

Step 3 — Turn the computer power on and enter the following program:

```
10 POKE 65346,255
20 POKE 65347,4
30 POKE 65346,255
```

Step 4 — Run the program and observe the voltmeter reading. You should observe a constant voltage level of approximately 10 volts. This is the maximum output voltage level provided by the NE5018 DAC circuit in Fig. 6-16. If you did not obtain any output, check your circuit and program for possible errors.

Step 5 — Enter and run the following program:

```
10 POKE 65346,255
20 POKE 65347,4
30 FOR X = 0 TO 255
40 POKE 65346,X
50 NEXT X
60 GOTO 30
```

Step 6 — Observe the DAC output indication on the voltmeter. If the circuit is working properly, the output voltage level should increase from approximately 0 to 10 volts then fall back to 0 volts. The cycle then repeats itself.

Discussion

The NE5018 in Fig. 6-16 is connected to port B of the PIA. Consequently, the associated software must first configure port B of the PIA for output. This is accomplished by lines 10 and 20 in the preceding programs. The DAC will then output a voltage level which corresponds to the digital value written to the port B data register located at address 65346. The first program simply writes the value 255 to the port B data register. Since the NE5018 is an 8-bit DAC, the value 255 will provide a maximum output voltage level from the DAC. The second program is a ramp generation program that sequentially writes values from 0 to 255 to the DAC. Consequently, the DAC output increases from 0 volts to its maximum (approximately 10 volts) then drops back to 0 volts. The ramp cycle is then repeated.

As before, the BASIC program is too slow to generate higher frequencies. Higher frequencies must be generated using an assembly language program.

Procedure (continued)

Step 7 — Remove the voltmeter and connect an oscilloscope to the DAC output. Type in the following command to erase the previous program from the computer memory:

NEW **ENTER**

Step 8 — Enter the following programs into the computer:

```
10 REM "ASSY. LANG. LOADER"  
*15 PCLEAR 1  
20 POKE 65346,255  
30 POKE 65347,4  
*40 CLEAR 25,8000  
50 DATA 142,7,208,166,128,183,255,66,140,8,207,38,  
246,32,241  
60 FOR I = 0 TO 14  
*70 READ B: POKE 8000 + I,B  
80 NEXT I  
90 END
```

***Notes:**

*Do not enter line 15 unless you have an Extended Color BASIC system.
If you have a 4K system, you must use the value 4000 in place of 8000 in lines 40 and 70.*

```
100 REM "RAMP DATA GEN."  
110 N = 2000  
120 FOR X = 0 TO 255  
130 POKE N,X  
140 N = N + 1  
150 NEXT X  
160 END
```

Step 9 — Run the assembly language loader program which begins at line 10.

Step 10 — Run the data generation program which begins at line 100.

Step 11 — Execute the assembly language display program by entering the following command:

***EXEC 8000** **ENTER**

**Use the value 4000 for 4K systems.*

Step 12 — Observe the oscilloscope display. You should observe the ramp (sawtooth) waveform pictured in Fig. 6-17. Measure the output waveform amplitude and frequency. Our waveform had a peak amplitude of 10 volts and a frequency of approximately 200 Hz.

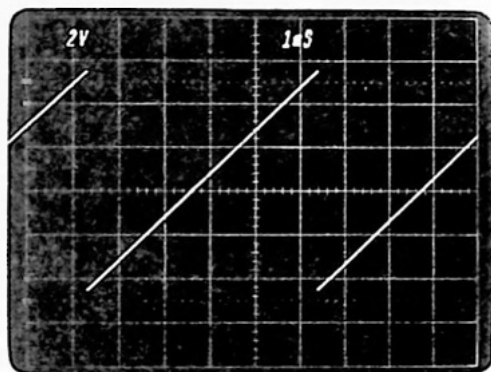


Figure 6-17 _____
A ramp (sawtooth) waveform generated by the NE5018 external DAC using an assembly language program.

Step 13 — Expand the oscilloscope time base (seconds/division) and volts/division settings such that you can observe the output steps of the DAC as pictured in Fig. 6-18. Carefully measure the DAC step size. We found the step size of our DAC output to be 0.04 volt. This value makes sense since the maximum output level of the 8-bit DAC is 10 volts. Consequently, the step size of the DAC is:

$$\text{Step Size} = \frac{1}{256} \times 10 = 0.039 \text{ volt}$$

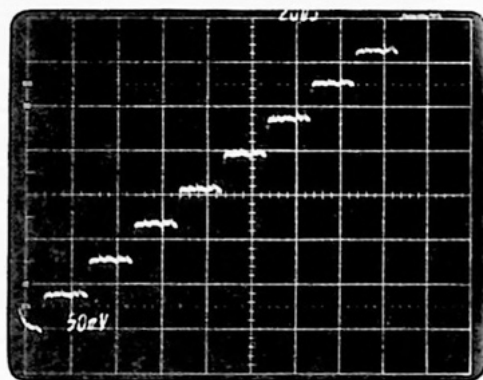


Figure 6-18 _____
The output steps of the NE5018 external DAC.

Step 14 — Reset the computer by pressing the RESET button on the back right side of the computer case.

Step 15 — Change the data generation program beginning at line 100 to first generate a triangular and then a square waveform. The data generation programs required to produce these waveforms are similar to those used with the internal DAC in Experiment 3. However, you must generate 256 data values instead of 64 data values since the external NE5018 is an 8-bit DAC. When the proper changes have been made, you should observe the waveforms pictured in Figs. 6-19 and 6-20 when each respective waveform is generated.

Step 16 — After you have successfully generated the triangular and square waveforms, reset the computer and change the DATA statement in line 50 of the assembly language loader program as follows:

```
50 DATA 142,7,208,166,128,183,255,66,140,8,131,38,  
246,32,241
```

Step 17 — Run the assembly language loader program which begins at line 10.

Step 18 — Enter and run the following sine waveform data generation program:

```
100 REM "SINE DATA GEN."  
110 N = 2000  
120 FOR X = 0 TO 360 STEP 2  
130 Y = 128 + INT(127*(SIN(X*0.017453)))  
140 POKE N,Y  
150 N = N + 1  
160 NEXT X  
170 END
```

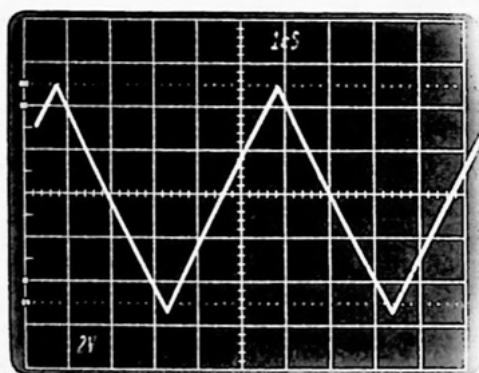


Figure 6-19
A triangular waveform generated by the NE5018 external DAC.

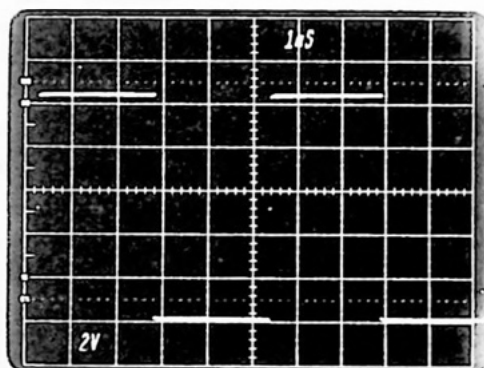


Figure 6-20
A square waveform generated by the NE5018 external DAC.

Step 19 — Execute the assembly language program as before by entering the command, EXEC 8000 (or 4000). Observe the oscilloscope display. You should observe the sine waveform pictured in Fig. 6-21.

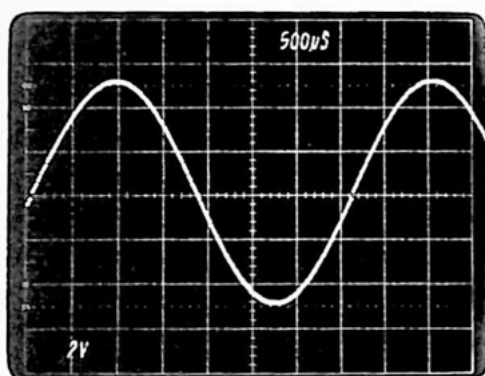


Figure 6-21
A sine waveform generated by the NE5018 external DAC.

Discussion

To generate higher output frequencies from the NE5018 DAC you must use an assembly language program. The procedure is the same as for the internal DAC in Experiment 3. The assembly language display program is loaded into memory with a BASIC program. Another BASIC program is executed to generate the data values for the waveform and store them in memory. The assembly language display

program is then executed to read the waveform data values from memory and write them to the external DAC. Since the external DAC is an 8-bit device, 256 data values are generated to provide increased resolution.

You will use the NE5018 DAC to perform Experiment 6. Therefore, *do not* disconnect any of the PIA/DAC circuit connections.

EXPERIMENT 5: INTERNAL ANALOG-TO-DIGITAL CONVERSION

Purpose

To demonstrate analog-to-digital conversion using the internal 6-bit ADC of the Color Computer.

Equipment

- 1 Color Computer
- 1 0–10,000-Ohm Potentiometer
- 1 Oscilloscope
- 1 Sine-Wave Generator

Diagrams (Figs. 6-22 and 6-23)

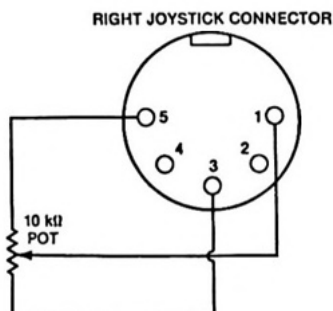


Figure 6-22

An analog signal applied to pin 1 of the right joystick connector can be converted to a digital input value by the internal A/D converter.

Procedure

Step 1 — Turn your computer off and connect a 10,000-ohm potentiometer to the *right* joystick input connector as shown in Fig. 6-22. To make contact with the connector pins, you can insert a small pin or wire into each respective connector pin. A doubled-over $\frac{1}{2}$ -watt resistor lead wire will do the job. Then, attach the potentiometer leads to the inserted wire leads. An alternative method is to construct a 5-pin male DIN connector plug as shown in Fig. 6-23. Once constructed, the plug can be easily inserted into the right joystick port of the Color Computer to provide a convenient means of

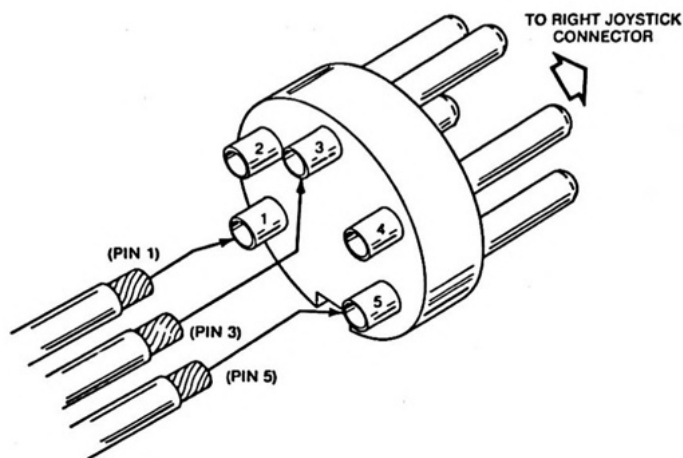


Figure 6-23

A 5-pin male DIN connector plug can be constructed to input external analog signals to the Color Computer's joystick port(s).

inputting an analog signal into the Color Computer. The 5-pin male DIN connector can be purchased at most Radio Shack stores. You will note that the required joystick connector plug is different from the cassette connector plug used in Experiment 3.

Step 2 — Turn the computer on and enter the following program:

```
10 X = JOYSTK (0)
20 Y = .23 + (X*.072)
30 PRINT X,Y
40 GOTO 10
```

Step 3 — Run the program and observe the display on the TV monitor. The display should indicate the digital equivalent (from 0 to 63) of the analog voltage input level. To the right of the digital value, the display will show the actual voltage level being input to the computer.

Step 4 — Change the potentiometer setting and note the change in the display. You might want to connect a voltmeter or oscilloscope between the potentiometer wiper and ground to check the accuracy of the displayed voltage values. You have just constructed a microcomputer-controlled digital voltmeter, or DVM.

Discussion

The JOYSTK (0) command in the program simply reads and converts the analog voltage level produced by the potentiometer and input to the right joystick X connector pin. The actual input voltage level is then calculated from the converted digital value in line 20 of the program. The computer displays both the digital equivalent of the analog input voltage level and the actual voltage level being input to the computer.

Procedure (continued)

Step 5 — Connect an oscilloscope to the internal DAC output available on the cassette connector, pins 3 and 4 (Reference Experiment 3).

Step 6 — Enter the following "echo" program into the computer:

```
10 REM "ECHO"  
20 X = JOYSTK (0)  
30 Y = .23 + (X*.072)  
40 POKE 65312,X  
50 GOTO 20
```

Step 7 — Run the echo program and vary the potentiometer setting. The internal DAC output observed on the oscilloscope should "echo" a voltage level proportional to the voltage level obtained from the potentiometer. The echoed voltage will have a range from 0 to 1 volt corresponding to a 0 to 5-volt input range. In addition, if you look close you will observe a "glitch" in the displayed waveform. Recall that the internal DAC is also used to perform A/D conversions. The glitch is generated when the Color Computer reads the input signal and performs the A/D conversion.

Step 8 — Turn the computer power off and disconnect the potentiometer from the joystick connector. Connect a sine-wave generator and a 1.5-volt battery, or DC offset, to the right joystick connector as shown in Fig. 6-24. Set the generator frequency between 10 Hz and 100 Hz. Set the generator for a 3-volt peak-to-peak sine-wave output.

RIGHT JOYSTICK CONNECTOR

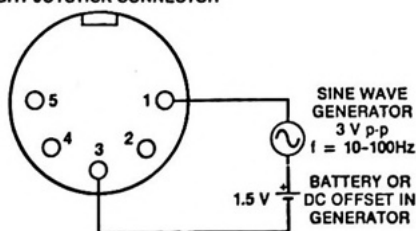


Figure 6-24
A sine-wave generator connected to the right joystick analog input port.

Step 9 — Turn the computer power on and enter the following echo program again:

```
10 REM "ECHO"  
20 X = JOYSTK (0)  
30 Y = .23 + (X* .072)  
40 PRINT X,Y  
50 POKE 65312,X  
60 GOTO 20
```

Step 10 — Run the program and observe the TV and oscilloscope displays. The TV should display the converted digital voltage values obtained from the analog sine-wave input. The sine-wave input is being "echoed" by the internal DAC. Consequently, the DAC is attempting to reproduce the sine wave from the converted digital voltage values. However, you will note from the oscilloscope display that the reproduction is not very good. This is because the BASIC program provides a sampling rate that is too slow to adequately "follow" the input waveform frequency.

Step 11 — Hold the shift key down and press the @ key. This will stop the program execution and you should observe a series of converted sine-wave values on the TV display. Notice that only 3 or 4 values are provided for each input cycle because of the slow sampling rate. Press any key to continue the program execution.

Discussion

The program reads and displays the analog input level as before. In addition, this program writes the converted digital value to the internal DAC located at address 65312. Consequently, the DAC is attempting to echo the input waveform. However, the BASIC program is too slow to adequately sample the input waveform.

Procedure (continued)

Step 12 — Enter the following assembly language loader program into the computer:

```
10 REM "ASSY. LANG. LOADER"  
*15 PCLEAR 1  
*20 CLEAR 25,8000  
30 DATA 198,0,142,1,90,189,169,229,182,1,90,183,  
255,32,32,240  
40 FOR I = 0 TO 15
```

```

*50 READ B: POKE 8000 + I,B
60 NEXT I
70 END

```

***Notes:**

Do not enter line 15 unless you have an *Extended Color BASIC* system.

If you have a 4K system, you must use the value 4000 in place of 8000 in lines 20 and 50.

Step 13 — Run the BASIC program in Step 12, then execute the assembly language program by entering the command, EXEC 8000 (4000 for 4K systems).

Step 14 — Observe the oscilloscope display. You should observe a waveform similar to that pictured in Fig. 6-25.

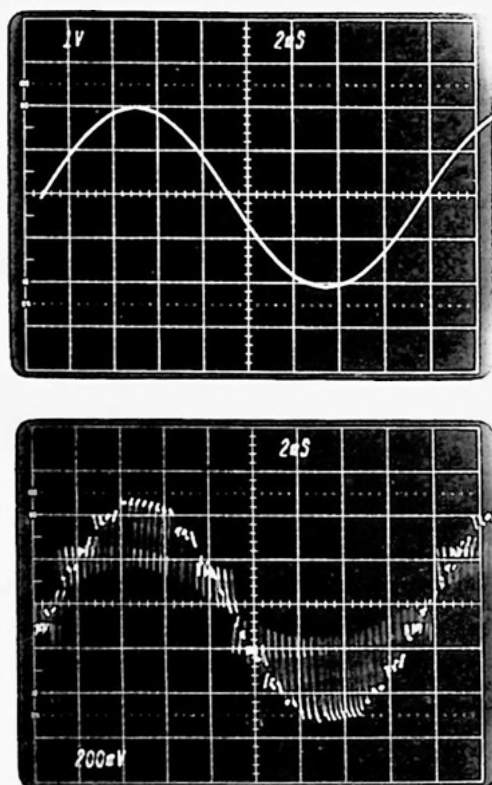


Figure 6-25 _____
A sine-wave input to the internal ADC (top) and its "echoed" output from the internal DAC (bottom).

Discussion

An assembly language program is now being used to sample and convert the sine-wave input. The converted value is then stored to the internal DAC which echoes the input. The DAC output display now resembles a sine wave since the sampling rate has been increased by the assembly language program. However, you will note that the DAC output is full of "glitches." Recall from Chapter 5 that the internal Color Computer DAC is also used to perform the internal analog-to-digital conversions. Thus, you are observing the successive approximation iterations between display points on the echoed waveform being displayed by the oscilloscope. To remove the glitches, an external ADC, which is separate from the internal DAC, must be used to perform the analog-to-digital conversions on the input waveform. Even though the echoed waveform is full of glitches, you will notice that the assembly language program provides a sampling rate which is fast enough to follow the input waveform frequency.

EXPERIMENT 6: ADC INTERFACING

Purpose

To connect an external 8-bit ADC to the expansion connector interface and demonstrate analog-to-digital conversion using the external ADC.

Equipment

- 1 Color Computer
- 1 Expansion Connector Breadboard
- 1 Solderless Connector Block
- 1 NE5018/PIA Interface Circuit from Experiment 4
- 1 National Semiconductor ADC0804LCN 8-Bit Analog-to-Digital Converter
- 1 150-Picofarad Capacitor
- 1 10,000-Ohm Resistor
- 1 0–10,000-Ohm Potentiometer
- 1 Sine-Wave Generator
- Hook-Up Wire

Schematic Diagram (Fig. 6-26)

Procedure

Step 1 — Turn the computer power off and connect the ADC0804LCN to the PIA interface as shown in Fig. 6-26. (*Caution:*

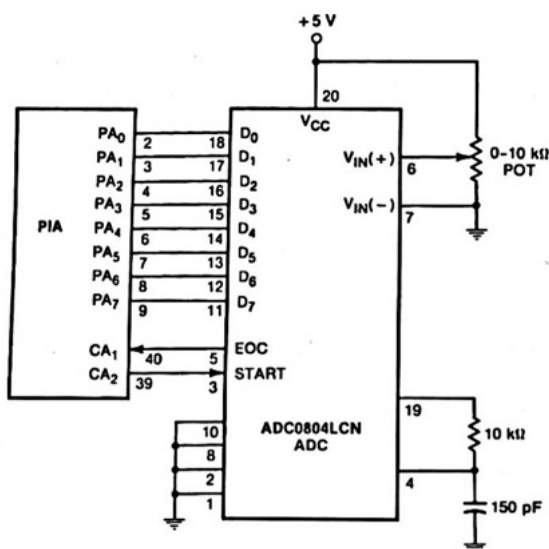


Figure 6-26
Schematic diagram for Experiment 6.

The ADC0804LCN ADC is a MOS device and must be protected from static electricity. Make sure you are grounded and avoid touching the pins of the ADC0804LCN. When you are not using this device, store it in conductive foam or an antistatic tube.)

Step 2—Turn the computer power on and enter the following program:

```
10 POKE 65345,44
20 A = PEEK (65344)
30 PRINT A, A*0.0196
40 B = PEEK (65345)
50 IF B > 128 THEN 20 ELSE 40
```

Step 3—Run the program and observe the display on the TV monitor. The display should indicate the digital equivalent (from 0 to 255) of the analog voltage input level. To the right of the digital value, the display will show the actual voltage level being input to the external ADC.

Step 4—Change the potentiometer setting and note the change in the display. You might want to connect a voltmeter or oscilloscope between the potentiometer wiper and ground to check the accuracy

of the displayed voltage values. You have just constructed a microcomputer-controlled digital voltmeter, or DVM.

Discussion

You have connected the ADC to port A of the PIA interface. The first line of the program configures port A for the input operation by writing the value 44_{10} to the port A control register at address 65345. The program then starts the conversion process by reading the port A data register located at address 65344. Recall that with this configuration a read operation on port A will cause a pulse to be generated on the CA2 output control line. Consequently, the ADC conversion process is started since the CA2 output line is connected to the ADC START line (see Fig. 6-26). The program then polls the bit 7 status of the port A control register located at address 65345 (lines 40 and 50). Since the End-of-Conversion line of the ADC is connected to the CA1 input control line of the PIA, bit 7 of the port A control register will set when the conversion is complete. Once the conversion is complete, the converted value is read from port A (line 20), and displayed on the TV monitor (line 30).

Step 5 — Turn the computer power off and connect an oscilloscope to the NE5018 DAC output line (pin 18). The NE5018 DAC is connected to port B of the PIA from Experiment 4.

Step 6 — Turn the computer power on and enter the following "echo" program:

```
10 POKE 65345,44
20 POKE 65346,255
30 POKE 65347,4
40 A = PEEK (65344)
50 PRINT A, 0.0196*A
60 POKE 65346,A
70 B = PEEK (65345)
80 IF B > 128 THEN 40 ELSE 70
```

Step 7 — Run the echo program and vary the potentiometer setting. The NE5018 DAC output observed on the oscilloscope should "echo" the input voltage level obtained from the potentiometer. However, the NE5018 DAC output observed on the oscilloscope will be twice the input voltage value since the DAC has a 0–10-volt output range and the ADC has a 0–5-volt input range.

Step 8 — Turn the computer power off and disconnect the potentiometer from the ADC input. Connect a sine-wave generator to the

ADC input as shown in Fig. 6-27. Notice that you must provide a DC offset voltage of approximately 1.5 volts in series with the sine-wave input so that the input voltage values are all above zero. Recall that the ADC0804LCN has an operating range of 0–5 volts. You can provide the required DC offset by using a 1.5-volt battery in series with the generator as shown in Fig. 6-27. The battery is not needed if your generator is equipped with a DC offset control.

Step 9—Set the sine-wave generator output frequency for 100 Hz with a peak-to-peak amplitude of approximately 3 volts.

Step 10—Turn the computer power on and enter the following echo program again:

```
10 POKE 65345,44
20 POKE 65346,255
30 POKE 65347,4
40 A = PEEK (65344)
50 PRINT A, 0.0196*A
60 POKE 65346,A
70 B = PEEK (65345)
80 IF B > 128 THEN 40 ELSE 70
```

Step 11—Run the echo program and observe the TV and oscilloscope displays. The TV should display the digital voltage values obtained from the analog sine-wave input. The sine-wave input is being echoed by the NE5018 DAC. Consequently, the DAC is attempting to reproduce the sine wave from the converted digital voltage values. Again, the reproduction is not very good since the BASIC program is not fast enough to follow the input waveform.

Step 12—Hold the shift key down and press the @ key on the computer keyboard. Notice that the program is only providing 3 or 4 conversions for each input cycle.

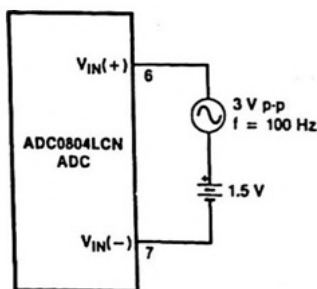


Figure 6-27
A sine-wave generator with a 1.5-volt DC offset connected to the ADC0804LCN input.

Discussion

You now have an ADC connected to port A of the PIA interface and a DAC connected to port B of the PIA interface. Therefore, the program first configures port A for input as in the first part of this experiment, and port B for output as in Experiment 4. The computer then reads the converted value from the ADC at port A and writes the value to the DAC at port B to provide the echo. However, you must use an assembly language program to provide a higher sampling rate so that the waveform can be accurately digitized and reproduced by the DAC.

Procedure (continued)

Step 13 — Enter the following assembly language loader program into the computer:

```
10 REM "ASSY. LANG. LOADER"  
*15 PCLEAR 1  
20 POKE 65345,44  
30 POKE 65346,255  
40 POKE 65347,4  
*50 CLEAR 25,8000  
60 DATA 182,255,64,182,255,65,42,251,182,255,64,  
183,255,66,32,243  
70 FOR I = 0 TO 15  
*80 READ B: POKE 8000 + I,B  
90 NEXT I  
100 END
```

*Notes:

Do not enter line 15 unless you have an Extended Color BASIC system.

If you have a 4K system, you must use the value 4000 in place of 8000 in lines 50 and 80.

Step 14 — Execute the assembly language program by entering the command, EXEC 8000 (or 4000).

Step 15 — Observe the oscilloscope display. You should observe the "echoed," or reproduced, sine wave pictured in the bottom of Fig. 6-28. Notice that you can actually see the points on the waveform that were sampled and output to the DAC. We have pictured the sine-wave input to the ADC at the top of Fig. 6-28 for comparison purposes. You can actually measure how much time is required for the input-to-output echo by measuring the horizontal displacement be-

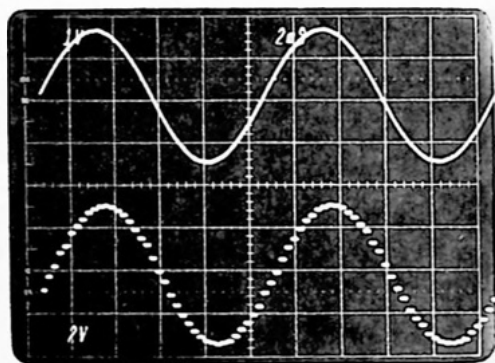


Figure 6-28
A 100-Hz sine-wave input to the ADC0804LCN (top) and its "echoed" output from the NE5018 (bottom).

tween the input and output waveforms shown in Fig. 6-28. We found the conversion time to be approximately 300–400 microseconds. The time delay is a function of the time it takes to execute the assembly language program and the conversion time required by the external ADC and DAC.

Step 16—Increase the frequency of the generator until the echoed sine wave is not reproducible. We have found that the maximum reproducible frequency is around 500 Hz. After this, the echoed sine wave is quite distorted. Our results at 1000 Hz are pictured in Fig. 6-29.

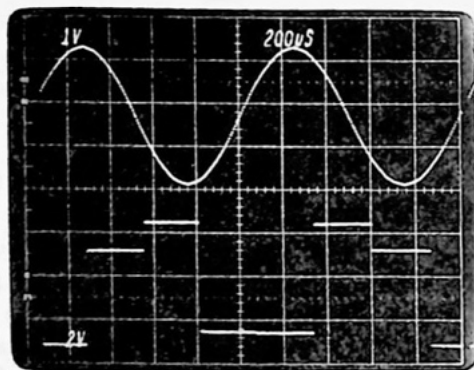


Figure 6-29
A 1000-Hz sine-wave input to the ADC0804LCN (top) and its "echoed" output from the NE5018 (bottom).

Discussion

The assembly language program uses a polling routine to read the ADC values and write them to the DAC to provide the input-to-output echo. As you can see, even the speed of the assembly language program cannot sample the input waveform fast enough to reproduce the input waveform above 500 Hz.

A Number Systems and Computer Arithmetic

NUMBER SYSTEMS

Digital computers are made up of a series of logic gates integrated onto "chips." These gates are composed of standard electronic devices such as resistors, capacitors, diodes, and transistors. Each of these devices can be made to function like a switch; that is, to represent an "on" or "off" state. A resistor is either conducting or not conducting. A capacitor is charged or not charged. The diode or transistor is either conducting or not conducting. Thus, all of these devices can be used to represent one of two states—on or off. Digital computers, therefore, ultimately use a two-state system to represent numbers. Binary is a number system ideally suited for digital computers since it comprises only two digits, 1 and 0, which can be used to represent on and off states. Even though a computer operates on binary information, it is very cumbersome to you and me. Therefore, it is easier for us to use a higher-order number system or computer language when communicating with the computer. Two such number systems commonly used with microcomputers are octal (base 8) and hexadecimal (base 16).

Decimal (Base 10)

This is the system familiar to all of us. It utilizes the digits 0 through 9. Remember how you learned to read decimal numbers? You started with the decimal point. To the left of that point, each position represented a positive power of 10 from 10^0 on up. To the right of that point, each position represented a negative power of 10 from 10^{-1} on

down. Positional weights in the decimal system are arranged as follows:

Positional Weights $> \dots [10^2] [10^1] [10^0] \cdot [10^{-1}] [10^{-2}] \dots$

To interpret a number you simply:

1. Determine the positional weight of each digit.
2. Multiply the digit times its positional weight.
3. Add all of the above products.

Decimal Interpretation:

Interpret the number $(537)_{10}$.

$$\begin{aligned}(537)_{10} &= 5 \quad 3 \quad 7 \\ &= (5 \times 10^2) + (3 \times 10^1) + (7 \times 10^0) \\ &= 500 \quad + \quad 30 \quad + \quad 7\end{aligned}$$

Think about it. Isn't this what you do mentally to determine the "weight" of a number?

Binary (Base 2)

This is the number system used by the electronics of all digital computers. It uses the digits 0 and 1. The idea here is the same as with decimal numbers except that powers of 2 are used rather than 10 for the digit weights. Positional weights in the binary number system are arranged as follows:

Positional Weights $> \dots [2^2] [2^1] [2^0] \cdot [2^{-1}] [2^{-2}] \dots$

To interpret a binary number the procedure is the same:

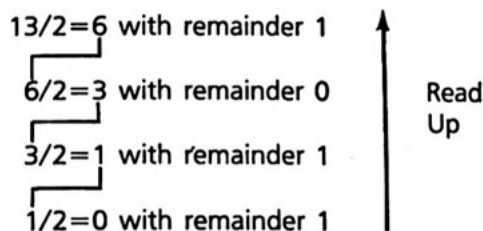
1. Determine the positional weight of each digit.
2. Multiply the digit by its positional weight.
3. Add the above products.

Binary $\rightarrow$ Decimal

Convert 1101_2 to decimal.

$$\begin{aligned}(1101)_2 &= 1 \quad 1 \quad 0 \quad 1 \\ &= (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) \\ &= 8 \quad + \quad 4 \quad + \quad 0 \quad + \quad 1 \\ &= (13)_{10}\end{aligned}$$

You now have a feeling for the weight of the binary number 1101 since you have expressed it as a decimal number—something you are familiar with. Now suppose you wish to express the decimal 13 as a binary number. The procedure is as follows:



Therefore, $(13)_{10} = (1101)_2$

To summarize the process, you see that the binary number is obtained by repetitive division by the base number, 2 in this case. If you need some brushing-up on binary numbers, study the preceding examples and work the related problems at the end of this Appendix until you feel comfortable with the process.

Octal (Base 8)

Now that you know the decimal and binary systems, octal will be easy. Octal uses eight digits, 0 through 7. The idea of positional weights is the same as in decimal and binary except that now you use powers of 8 as follows:

Positional Weights > . . . $[8^2]$ $[8^1]$ $[8^0]$ • $[8^{-1}]$ $[8^{-2}]$. . .

The interpretation procedures are the same as you used for the binary system.

Octal→Decimal

Interpret the number $(257)_8$.

$$\begin{aligned}
 (257)_8 &= 2 \ 5 \ 7 \ . \\
 &= (2 \times 8^2) + (5 \times 8^1) + (7 \times 8^0) \\
 &= 128 + 40 + 7 \\
 &= (175)_{10}
 \end{aligned}$$

Decimal→Octal

Express $(175)_{10}$ as an octal number:

$$\begin{array}{rcl} 175/8 & = & 21 \text{ with remainder } 7 \\ \hline 21/8 & = & 2 \text{ with remainder } 5 \\ \hline 2/8 & = & 0 \text{ with remainder } 2 \end{array} \quad \begin{array}{c} \uparrow \\ \text{Read} \\ \text{Up} \end{array}$$

Therefore, $(175)_{10} = (257)_8$ which checks with the above example. As before, work the problems at the end of this Appendix if you need practice.

Hexadecimal (Base 16)

The hexadecimal numbering system, or "hex," uses the 10 digits 0 through 9, and the alphabetical characters A, B, C, D, E, and F. It might seem awkward to let the letters A through F represent numbers, but keep in mind that numbers are just symbols for quantities. In this case:

$$A=(10), B=(11), C=(12), D=(13), E=(14), \text{ and } F=(15).$$

Keeping this in mind, the interpretation processes are the same. The positional weights for hex (base 16) are as follows:

$$\text{Positional Weights} > \dots [16^2] [16^1] [16^0] \bullet [16^{-1}] [16^{-2}] \dots$$

Hex→Decimal

Interpret the number $(9CF)_{16}$.

$$\begin{aligned} (9CF)_{16} &= 9 \quad C \quad F \\ &= (9 \times 16^2) + (C \times 16) + (F \times 1) \\ &= (9 \times 256) + (12 \times 16) + (15 \times 1) \\ &= (2511)_{10} \end{aligned}$$

Note the quantity you can represent with just 3 digits using hex.

Decimal→Hex

Express $(2511)_{10}$ as a hex quantity:

$$\begin{array}{rcl}
 2511/16 & = & 156 \text{ with remainder } 15=F \\
 \swarrow & & \uparrow \\
 156/16 & = & 9 \text{ with remainder } 12=C \\
 \swarrow & & \uparrow \\
 9/16 & = & 0 \text{ with remainder } 9
 \end{array}
 \quad
 \begin{array}{c}
 \uparrow \\
 \text{Read} \\
 \text{Up}
 \end{array}$$

Thus, $(2511)_{10} = (9CF)_{16}$

As we said before, you might "talk" to the computer using octal or hex; however, the computer must translate this to binary. Therefore, it might be beneficial if you also could make this translation to better understand the operation of the digital computer.

Octal $\rightarrow$ Binary

Translate $(257)_8$ to binary:

$$\begin{array}{ccc}
 \begin{array}{c} 2 \\ \swarrow \searrow \\ 010 \end{array} &
 \begin{array}{c} 5 \\ \swarrow \searrow \\ 101 \end{array} &
 \begin{array}{c} 7 \\ \swarrow \searrow \\ 111 \end{array}
 \end{array}
 \quad
 (257)_8 = (010101111)_2$$

Note: You simply expressed each octal digit as a 3-bit number.

Binary $\rightarrow$ Octal

Translate $(010101111)_2$ to octal:

$$\begin{array}{ccc}
 \begin{array}{c} 010 \\ \swarrow \searrow \\ 2 \end{array} &
 \begin{array}{c} 101 \\ \swarrow \searrow \\ 5 \end{array} &
 \begin{array}{c} 111 \\ \swarrow \searrow \\ 7 \end{array}
 \end{array}
 \quad
 (010101111)_2 = (257)_8$$

The process is reversed; i.e., divide the binary number into groups of three digits and translate to the octal digit.

Hex $\rightarrow$ Binary

Translate $(9CF)_{16}$ to binary:

$$\begin{array}{ccc}
 \begin{array}{c} 9 \\ \swarrow \searrow \\ 1001 \end{array} &
 \begin{array}{c} C \\ \swarrow \searrow \\ 1100 \end{array} &
 \begin{array}{c} F \\ \swarrow \searrow \\ 1111 \end{array}
 \end{array}
 \quad
 (9CF)_{16} = (100111001111)_2$$

Binary → Hex

Translate $(100111001111)_2$ to hex:

$$\begin{array}{ccc} \begin{array}{c} 1001 \\ \swarrow \searrow \\ 9 \end{array} & \begin{array}{c} 1100 \\ \swarrow \searrow \\ C \end{array} & \begin{array}{c} 1111 \\ \swarrow \searrow \\ F \end{array} \end{array} \quad (100111001111)_2 = (9CF)_{16}$$

The process is similar to the octal translation except that you break the binary number into groups of four digits.

Bits, Bytes, and Nibbles

A binary digit (1 or 0) is referred to as a "bit." Information in microprocessors and all digital computers is typically represented by a series of bits. A string of 4 bits is referred to as a "nibble" and an 8-bit string is called a "byte." You will see that memory *addresses* used by the 6809 are 16 bits = 4 nibbles = 2 bytes. *Data* within the 6809 are represented using 8 bits = 2 nibbles = 1 byte.

Bits, Bytes, and Nibbles

	4 bits:	1011 = 1 nibble
6809 Data —	8 bits:	1001 1011 = 1 byte
6809 Addresses —	16 bits:	1101 0100 1001 1011 = 2 bytes

A "word" is not as easily defined. Its definition is largely a function of the way we use it. For example, an *address word* for the 6809 is 16 bits while a *data word* is 8 bits in length. The PDP-11 minicomputer has a word length of 16 bits, which means that it represents information as 2 bytes. Some larger computers have word lengths of 24 bits, 32 bits, 64 bits, etc. Note that all of the above words are multiples of 8 bits (1 byte). The lesson here is to be very careful of how you use this "word."

Binary Coded Decimal (BCD)

Binary coded decimal (BCD) is another way in which numbers can be represented in digital computers and digital systems, particularly digital instrumentation. This number system was developed because it is easy to convert between BCD and decimal. When converting from decimal to BCD, *each* decimal digit is represented by its own 4-bit binary equivalent.

Decimal $\rightarrow$ BCD

To convert 256_{10} to BCD you simply express each decimal digit as a 4-bit binary number.

$$2 = 0010$$

$$5 = 0101$$

$$6 = 0110$$

Therefore, $256_{10} = 0010\ 0101\ 0110_{\text{BCD}}$.

The reverse procedure is used for converting from BCD to decimal.

BCD $\rightarrow$ Decimal

To convert $1001\ 0101_{\text{BCD}}$ to decimal, you simply divide the BCD number into groups of four bits starting at the right, then convert each group of four bits to decimal.

$$\text{1st group of four bits: } 0101 = 5_{10}$$

$$\text{2nd group of four bits: } 1001 = 9_{10}$$

Therefore, $1001\ 0101_{\text{BCD}} = 95_{10}$.

Convert 110111_{BCD} to decimal:

$$\text{1st group of four bits: } 0111 = 7_{10}$$

$$\text{2nd group of four bits: } 0011 = 3_{10}$$

Therefore $110111_{\text{BCD}} = 37_{10}$.

Note: You had to add the leading zeros in the 2nd group of four bits.

Caution: BCD and binary are not the same. With BCD, any group of four bits which represents a decimal number larger than nine is considered *invalid*. For example, $1101 = 13_{10}$ is an invalid BCD number. However, in the binary system this bit configuration is valid.

As mentioned earlier, many digital instruments use the BCD format for data being transmitted and received. The 6809 has an instruction that will allow you to perform arithmetic operations with BCD num-

bers. This instruction is the decimal adjust accumulator (DAA) instruction. With this instruction, the 6809 can work with BCD data directly without using special BCD-to-binary conversion routines.

DIGITAL COMPUTER ARITHMETIC

Digital computers utilize the binary number system and, therefore, must add, subtract, multiply, and divide using this system.

Addition

Suppose that you wish to add two binary numbers. Just as you did in decimal, starting with the rightmost column, you simply add each column, generating and using the carries as you go. To determine the sum of a column, count the number of 1s in the column.

Binary Addition

Add 10_2 to 11_2 :

carry generated	1	
by previous column	11 ₂	
	+ 10 ₂	
	+-----	
	101 ₂	

└ Column contains odd number of 1s and no pairs of 1s.

└ Column contains even number of 1s and one pair of 1s.

└ Column contains odd number of 1s and no pairs of 1s.

If the number of 1s is odd, the sum will be 1 with a carry generated for each *pair* of 1s. If the number of 1s is even, the sum will be 0 with a carry generated for each *pair* of 1s in the column that was added. Study the above examples.

Check your results by converting to decimal:

$$\begin{array}{r}
 11_2 = 3_{10} \\
 10_2 = 2_{10} \\
 \hline
 101_2 = 5_{10}
 \end{array}$$

Add 1010_2 to 1110_2 :

carry generated	111	
by previous column	1110_2	
	$+ 1010_2$	
	11000_2	

Column contains even number of 1s with no pairs.

Column contains even number of 1s with one pair.

Column contains even number of 1s with one pair.

Column contains odd number of 1s with one pair.

Column contains odd number of 1s with no pairs.

Check:

$$\begin{array}{r}
 1110_2 = 14_{10} \\
 + 1010_2 = 10_{10} \\
 \hline
 11000_2 = 24_{10}
 \end{array}$$

Add $0010\ 1100_2$ to $0100\ 1010_2$:

Check:

$$\begin{array}{r}
 0010\ 1100_2 = 44_{10} \\
 + 0100\ 1010_2 = 74_{10} \\
 \hline
 0111\ 0110_2 = 118_{10}
 \end{array}$$

NEGATIVE NUMBERS AND TWOS COMPLEMENT

Microprocessors utilize twos complement to represent negative numbers. Let us examine some definitions before we use this idea.

Definition: The *ones complement* of a binary word is obtained by inverting all of the bits within the word. (The ones complement of a word, A , is symbolized by $\bar{A}$.)

Ones Complement

The ones complement of $1101\ 1011_2$ is $0010\ 0100_2$.

The ones complement of $0000\ 0111_2$ is $1111\ 1000_2$.

The ones complement of $1010\ 0000_2$ is $0101\ 1111_2$.

Definition: The *twos complement* of a binary word is obtained by taking the ones complement and adding 1 to it.

Twos Complement

Find the twos complement of each binary word above:

The twos complement of $1101\ 1011_2 =$

$$\begin{array}{r} \text{ones complement} + 1 = 0010\ 0100_2 \\ + 1_2 \\ \hline 0010\ 0101_2 \end{array}$$

The twos complement of $0000\ 0111_2 =$

$$\begin{array}{r} \text{ones complement} + 1 = 1111\ 1000_2 \\ + 1_2 \\ \hline 1111\ 1001_2 \end{array}$$

The twos complement of $1010\ 0000_2 =$

$$\begin{array}{r} \text{ones complement} + 1 = 0101\ 1111_2 \\ + 1_2 \\ \hline 0110\ 0000_2 \end{array}$$

The 6809 uses twos complements to represent negative numbers. Given any positive number, the twos complement of that number will be its negative, within the range of bits allowed.

Given any 8-bit data word, $D_7D_6D_5D_4D_3D_2D_1D_0$, we will designate the eighth bit (D_7) to be the sign bit. When this bit is 1, the 6809 will consider the word to be negative. When $D_7 = 0$, the 6809 will recognize the word as positive. Using this scheme and the twos-complement negative number system, it can be shown that the largest possible positive number using an 8-bit data word is $0111\ 1111_2 = +127_{10}$. The maximum negative quantity would be $1000\ 0000_2 = -128_{10}$.

Negative Numbers

What is the 8-bit binary representation for -5_{10} ? The 8-bit word for $+5_{10}$ is $0000\ 0101$; therefore, the 8-bit word for -5_{10} is the twos complement of this. Hence, $-5_{10} = 1111\ 1011_2$.

What is the 8-bit binary representation for -127_{10} ? It is $+127_{10} =$

Logic Operations

$$\begin{array}{r} 0101\ 1100_2 \\ \text{AND } 1110\ 1011_2 \\ \hline 0100\ 1000_2 \end{array}$$

$$\begin{array}{r} 0101\ 1100_2 \\ \text{OR } 1110\ 1011_2 \\ \hline 1111\ 1111_2 \end{array}$$

$$\begin{array}{r} 0101\ 1100_2 \\ \text{XOR } 1110\ 1011_2 \\ \hline 1011\ 0111_2 \end{array}$$

Multiplication and Division

To multiply or divide with the 6800, you must successively add or subtract, respectively. This is one of the drawbacks of the first generation processors such as the 6800. This process consumes time and memory space. The second and third generation chips such as the 6801, 6803, and 6809 all contain internal hardware multiply functions.

REVIEW QUESTIONS

1. To what decimal numbers do the following binary numbers correspond?
 - a. 1101 _____
 - b. 1111 1111 _____
 - c. 1101 0011 _____
 - d. 1111 1111 1111 1111 _____
2. To what hexadecimal numbers do the binary numbers in Question 1 correspond?
 - a. _____
 - b. _____
 - c. _____
 - d. _____
3. To what binary numbers do the following hexadecimal numbers correspond?
 - a. 00FC _____
 - b. 0100 _____
 - c. 01CA _____
 - d. ED5B _____
 - e. ABCD _____
4. To what decimal numbers do the hexadecimal numbers in Question 3 correspond?
 - a. _____
 - b. _____
 - c. _____
 - d. _____
 - e. _____
5. $(0011\ 0101_2) + (0100\ 1110_2) = \text{_____}_{16}$.
6. The ones complement of 0100 1010 is _____.

7. The two's complement of the number in question 6 is _____.
8. Using two's-complement code, what is the 8-bit binary representation for -16_{10} ?

9. Using two's-complement code, what decimal number does 1011 0110 represent?

10. $(0000\ 0101_2) - (0000\ 0011_2) = \text{_____}_2$.
11. $(1010\ 1100_2) \text{ XOR } (0111\ 0010_2) = \text{_____}_2$.
12. Convert the following decimal numbers to binary and BCD: (a) 7 (b) 15 (c) 25 (d) 109

ANSWERS

1. a. 13_{10}
b. 255_{10}
c. 211_{10}
d. $65,535_{10}$
2. a. D_{16}
b. FF_{16}
c. $D3_{16}$
d. $FFFF_{16}$
3. a. $0000\ 0000\ 1111\ 1100_2$
b. $0000\ 0001\ 0000\ 0000_2$
c. $0000\ 0001\ 1100\ 1010_2$
d. $1110\ 1101\ 0101\ 1011_2$
e. $1010\ 1011\ 1100\ 1101_2$
4. a. 252_{10}
b. 256_{10}
c. 458_{10}
d. $60,763_{10}$
e. $43,981_{10}$
5. 83_{16}
6. $1011\ 0101_2$
7. $1011\ 0110_2$
8. $1111\ 0000_2$
9. -74_{10}
10. $0000\ 0010_2$
11. $1101\ 1110_2$

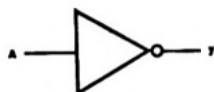
12. (a) 0111_2
 0111_{BCD}
 (b) 1111_2
 $0001\ 0101_{\text{BCD}}$
 (c) 11001_2
 $0010\ 0101_{\text{BCD}}$
 (d) 1101101_2
 $0001\ 0000\ 1001_{\text{BCD}}$

B Digital Review

BASIC LOGIC GATES

Recall the basic logic gates used in digital electronics. The following summary includes the gate symbol, the Boolean expression for the gate, and the truth table for the gate.

Inverter



A	y
0	1
1	0

$$y = \text{NOT } A = \overline{A}$$

OR Gate



AB	y
00	0
01	1
10	1
11	1

$$y = A \text{ OR } B = A + B$$

AND Gate



AB	y
00	0
01	0
10	0
11	1

$$y = A \text{ AND } B = A \cdot B$$

Exclusive OR Gate



AB	y
00	0
01	1
10	1
11	0

$$y = A \text{ Exclusive-OR } B = A \oplus B$$

NOR Gate



AB	y
00	1
01	0
10	0
11	0

$$y = \text{NOT } (A \text{ OR } B) = \overline{A+B}$$

NAND Gate



AB	y
00	1
01	1
10	1
11	0

$$y = \text{NOT } (A \text{ AND } B) = \overline{A \cdot B}$$

Exclusive NOR Gate

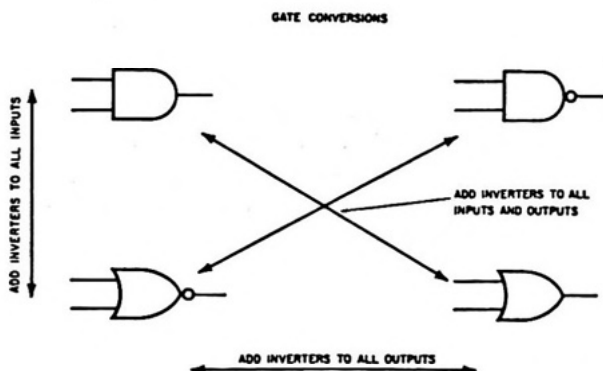


AB	y
00	1
01	0
10	0
11	1

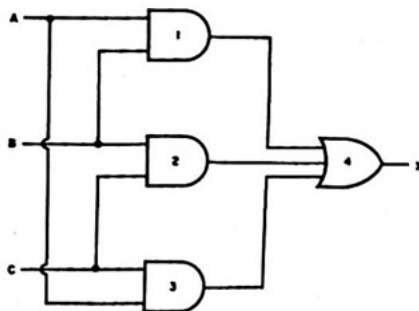
$$y = \text{NOT (A Exclusive-OR B)} = \overline{A \oplus B}$$

Gate Conversions

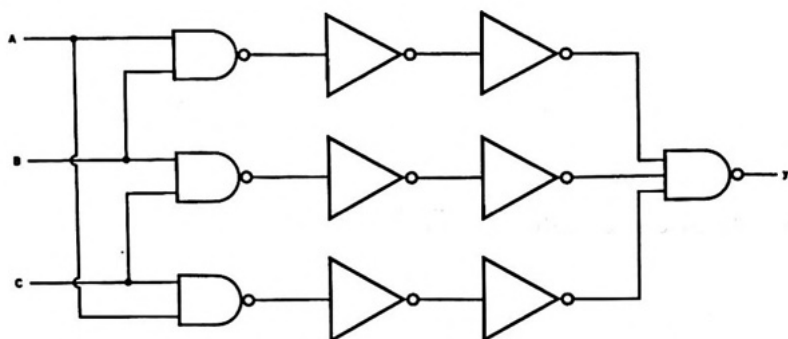
If we want to express one gate in terms of others the following conversion chart diagrams might be handy:



Convert the Following to NAND Logic:



By using the above conversion diagram, we get:

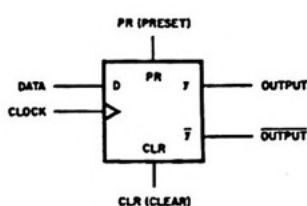


Note: The double inversions will cancel each other.

FLIP-FLOPS

A flip-flop is simply an electronic device that is used to store a binary digit. It will "latch" (hold) in one of two states (0 or 1) depending on what it sees at its input *when clocked*. There are two basic types we will be concerned with—the D flip-flop and the JK flip-flop.

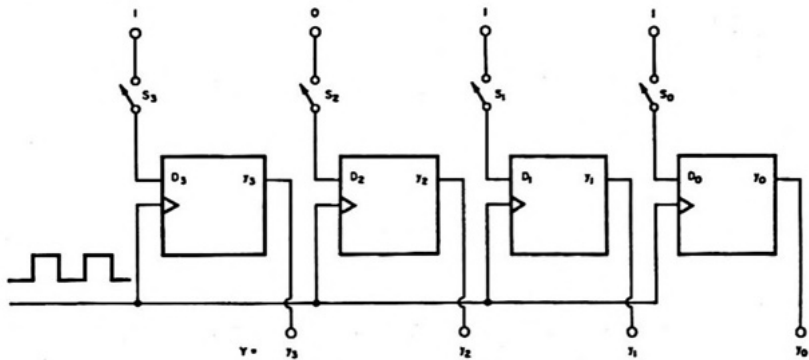
D Flip-Flop



CLK	D	y
0	0	Previous State
0	1	Previous State
1	0	0
1	1	1

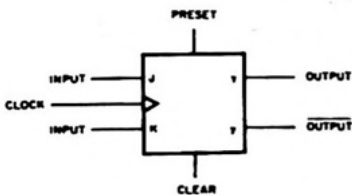
Note that the inputs are D (data line) and CLK (clock). The outputs are y and $\bar{y}$, where $\bar{y}$ is simply NOT y. By observing the truth table, you can see that binary data on the data line is transferred to the output when a clock pulse is provided. The outputs at y and $\bar{y}$ will remain latched until another clock pulse is provided. PR (preset) and CLR (clear) functions are also provided. A high at PR implies $y=1$ while a high at CLR implies $y=0$. No clocking is needed to preset or clear the latch. D flip-flops are used most commonly as storage registers.

4-Bit Storage Register



If in the beginning the four switches are open, the output is left floating and will be a random 4-bit number. After the switches are closed and a clock pulse provided, the output word will be $Y = y_3y_2y_1y_0 = 1011$. If you now would open the four switches, the output would remain at 1011 regardless of the amount of clocking. You can change the output by changing the input data, closing the switches, and clocking the circuit.

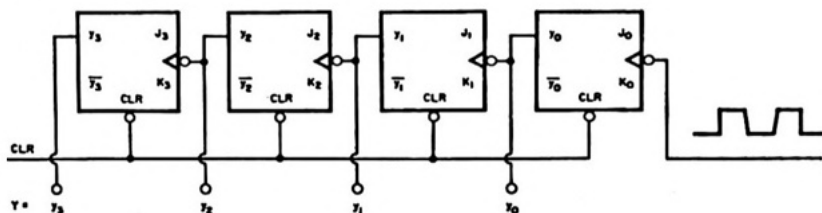
JK Flip-Flop



J	K	y
0	0	Last State (Latch)
0	1	0 (Reset)
1	0	1 (Set)
1	1	Last State (Toggle)

Here, the J and K inputs determine what the latch does when it is clocked. When J and K are both low, the clock pulse has no effect. When J and K are both high, the output will "toggle" (change) with each clock pulse. This idea of toggling is demonstrated in the following example.

4-Bit Counter



First note that the JK inputs are not connected (floating). In this floating mode, the flip-flop toggles the same as if $J = 1$ and $K = 1$. The "bubble" at the clock input indicates that the flip-flop will actuate when the clock goes in a negative direction, i.e., from 1 to 0. The CLR is also "bubbled"; therefore, you will begin by making CLR low $\rightarrow Y = y_3y_2y_1y_0 = 0000$.

1st negative clock $\rightarrow y_0 = 1 \rightarrow Y = y_3y_2y_1y_0 = 0001$

2nd negative clock $\rightarrow y_0 = 0 \rightarrow y_1 = 1 \rightarrow Y = 0010$

Note in the above case, y_0 went from a high to a low condition. Therefore, $y_1 = 1$ since y_0 provides the clock signal for the second gate.

3rd negative clock $\rightarrow y_0 = 1 \rightarrow y_1 = 1 \rightarrow Y = 0011$

Note here that y_0 toggled but y_1 remained the same since its clock (y_0) went high. Again, the flip-flop will only toggle when its clock goes low.

4th negative clock $\rightarrow y_0 = 0 \rightarrow y_1 = 0 \rightarrow y_2 = 1 \rightarrow Y = 0010$

Here,

1. y_0 toggled.
2. y_1 toggled since its clock (y_0) went low.
3. y_2 toggles since its clock (y_1) went low.

Following this procedure,

5th negative clock $\rightarrow Y = 0101$

6th negative clock $\rightarrow Y = 0110$

7th negative clock $\rightarrow Y = 0111$

8th negative clock $\rightarrow Y = 1000$

9th negative clock → Y=1001
10th negative clock → Y=1010
11th negative clock → Y=1011
12th negative clock → Y=1100
13th negative clock → Y=1101
14th negative clock → Y=1110
15th negative clock → Y=1111
16th negative clock → Y=0000 (Reset)

The circuit in this example is a binary counter. JK flip-flops are commonly used in this type of counter.

C Data Sheets

Complete data sheets for the devices listed in this Appendix should be obtained directly from the respective manufacturer:

MC6809E & MC6821	Motorola Semiconductors 3501 Ed Bluestein Boulevard Austin, TX 78721
NE5018	Signetics Corp. P. O. Box 9052 Sunnyvale, CA 94086
ADC0801 — ADC0804	National Semiconductor Corp. 2900 Semiconductor Drive Santa Clara, CA 95051

Only the first page of each data sheet is shown in this Appendix as a representative example of the information available. We urge you to obtain the latest edition of each data sheet directly from the manufacturer, as this information is subject to change and it is updated frequently.

**MOTOROLA****SEMICONDUCTORS**

3501 ED BLUESTEIN BLVD. AUSTIN TEXAS 78721

8-BIT MICROPROCESSING UNIT

The MC6809E is a revolutionary high performance 8-bit microprocessor which supports modern programming techniques such as position independence, reentrancy, and modular programming.

This third-generation addition to the M6800 family has major architectural improvements which include additional registers, instructions and addressing modes.

The basic instructions of any computer are greatly enhanced by the presence of powerful addressing modes. The MC6809E has the most complete set of addressing modes available on any 8-bit microprocessor today.

The MC6809E has hardware and software features which make it an ideal processor for higher level language execution or standard controller applications. External clock inputs are provided to allow synchronization with peripherals, systems or other MPUs.

MC6800 COMPATIBLE

- Hardware — Interfaces with All M6800 Peripherals
- Software — Upward Source Code Compatible Instruction Set and Addressing Modes

ARCHITECTURAL FEATURES

- Two 16-bit Index Registers
- Two 16-bit Indexable Stack Pointers
- Two 8-bit Accumulators can be Concatenated to Form One 16-Bit Accumulator
- Direct Page Register Allows Direct Addressing Throughout Memory

HARDWARE FEATURES

- External Clock Inputs, E and Q, Allow Synchronization
- TSC Input Controls Internal Bus Buffers
- LIC Indicates Opcode Fetch
- AVMA Allows Efficient Use of Common Resources in A Multiprocessor System
- BUSY is a Status Line for Multiprocessing
- Fast Interrupt Request Input Stacks Only Condition Code Register and Program Counter
- Interrupt Acknowledge Output Allows Vectoring By Devices
- SYNC Acknowledge Output Allows for Synchronization to External Event
- Single Bus-Cycle RESET
- Single 5-Volt Supply Operation
- NMI Inhibited After RESET Until After First Load of Stack Pointer
- Early Address Valid Allows Use With Slower Memories
- Early Write-Data for Dynamic Memories

SOFTWARE FEATURES

- 10 Addressing Modes
 - M6800 Upward Compatible Addressing Modes
 - Direct Addressing Anywhere in Memory Map
 - Long Relative Branches
 - Program Counter Relative
 - True Indirect Addressing
 - Expanded Indexed Addressing:
 - 0, 5, 8, or 16-bit Constant Offsets
 - 8, or 16-bit Accumulator Offsets
 - Auto-Increment/Decrement by 1 or 2
- Improved Stack Manipulation
- 1464 Instruction with Unique Addressing Modes
- 8 x 8 Unsigned Multiply
- 16-bit Arithmetic
- Transfer/Exchange All Registers
- Push/Pull Any Registers or Any Set of Registers
- Load Effective Address

MC6809E

(1.0 MHz)

MC68A09E

(1.5 MHz)

MC68B09E

(2.0 MHz)

H MOS

(HIGH-DENSITY N-CHANNEL, SILICON-GATE)

8-BIT**MICROPROCESSING UNIT**

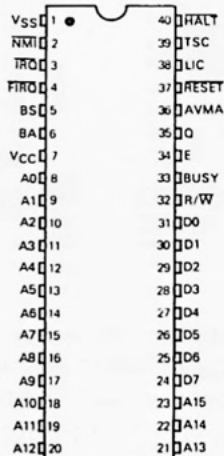
L SUFFIX
CERAMIC PACKAGE
CASE 715



P SUFFIX
PLASTIC PACKAGE
CASE 711



S SUFFIX
CERDIP PACKAGE
CASE 734

FIGURE 1 — PIN ASSIGNMENT

**MOTOROLA****SEMICONDUCTORS**

3501 ED BLUESTEIN BLVD. AUSTIN, TEXAS 78721

PERIPHERAL INTERFACE ADAPTER (PIA)

The MC6821 Peripheral Interface Adapter provides the universal means of interfacing peripheral equipment to the M6800 family of microprocessors. This device is capable of interfacing the MPU to peripherals through two 8-bit bidirectional peripheral data buses and four control lines. No external logic is required for interfacing to most peripheral devices.

The functional configuration of the PIA is programmed by the MPU during system initialization. Each of the peripheral data lines can be programmed to act as an input or output, and each of the four control/interrupt lines may be programmed for one of several control modes. This allows a high degree of flexibility in the overall operation of the interface.

- 8-Bit Bidirectional Data Bus for Communication with the MPU
- Two Bidirectional 8-Bit Buses for Interface to Peripherals
- Two Programmable Control Registers
- Two Programmable Data Direction Registers
- Four Individually-Controlled Interrupt Input Lines, Two Usable as Peripheral Control Outputs
- Handshake Control Logic for Input and Output Peripheral Operation
- High-Impedance Three-State and Direct Transistor Drive Peripheral Lines
- Program Controlled Interrupt and Interrupt Disable Capability
- CMOS Drive Capability on Side A Peripheral Lines
- Two TTL Drive Capability on All A and B Side Buffers
- TTL-Compatible
- Static Operation

MAXIMUM RATINGS

Characteristics	Symbol	Value	Unit
Supply Voltage	V_{CC}	-0.3 to +7.0	V
Input Voltage	V_{in}	-0.3 to +7.0	V
Operating Temperature Range MC6821, MC68A21, MC68B21 MC6821C, MC68A21C, MC68B21C	T_A	T_L to T_H 0 to 70 -40 to +85	°C
Storage Temperature Range	T_{stg}	-55 to +150	°C

THERMAL CHARACTERISTICS

Characteristic	Symbol	Value	Unit
Thermal Resistance			
Ceramic	θ_{JA}	50	°C/W
Plastic		100	
CerDip		60	

This device contains circuitry to protect the inputs against damage due to high static voltages or electric fields; however, it is advised that normal precautions be taken to avoid application of any voltage higher than maximum-rated voltages to this high-impedance circuit. Reliability of operation is enhanced if unused inputs are tied to an appropriate logic voltage (i.e., either V_{SS} or V_{CC}).

MC6821

(1.0 MHz)

MC68A21

(1.5 MHz)

MC68B21

(2.0 MHz)

MOS(N-CHANNEL, SILICON-GATE,
DEPLETION LOAD)**PERIPHERAL INTERFACE
ADAPTER**L SUFFIX
CERAMIC PACKAGE
CASE 715S SUFFIX
CERDIP PACKAGE
CASE 724P SUFFIX
PLASTIC PACKAGE
CASE 711**PIN ASSIGNMENT**

V_{SS}	1	40	CA1
PA0	2	39	CA2
PA1	3	38	IRQA
PA2	4	37	IROB
PA3	5	36	IR50
PA4	6	35	IR51
PA5	7	34	RESET
PA6	8	33	IO0
PA7	9	32	IO1
PB0	10	31	IO2
PB1	11	30	IO3
PB2	12	29	IO4
PB3	13	28	IO5
PB4	14	27	IO6
PB5	15	26	IO7
PB6	16	25	E
PB7	17	24	CS1
CB1	18	23	CS2
CB2	19	22	CS0
V_{CC}	20	21	IR/W

ADC0801, ADC0802, ADC0803, ADC0804 8-Bit μ P Compatible A/D Converters

General Description

The ADC0801, ADC0802, ADC0803, ADC0804 are CMOS 8-bit, successive approximation A/D converters which use a modified potentiometric ladder—similar to the 256R products. They are designed to meet the NSC MICROBUS™ standard to allow operation with the 8080A control bus, and TRI-STATE® output latches directly drive the data bus. These A/Ds appear like memory locations or I/O ports to the microprocessor and no interfacing logic is needed.

A new differential analog voltage input allows increasing the common-mode rejection and offsetting the analog zero input voltage value. In addition, the voltage reference input can be adjusted to allow encoding any smaller analog voltage span to the full 8 bits of resolution.

Features

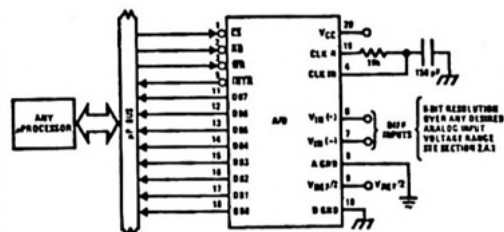
- MICROBUS (8080A) compatible—no interfacing logic needed
- Easy interface to all microprocessors, or operates "stand alone"

- Differential analog voltage inputs
- Logic inputs and outputs meet T²L voltage level specifications
- Works with 2.5V (LM336) voltage reference
- On-chip clock generator
- 0V to 5V analog input voltage range with single 5V supply
- No zero adjust required
- 0.3" standard width 20-pin DIP package

Key Specifications

- | | |
|--|--|
| ■ Resolution | 8 bits |
| ■ Total error | $\pm 1/4$ LSB, $\pm 1/2$ LSB and ± 1 LSB |
| ■ Conversion time | 100 μ s |
| ■ Access time | 135 ns |
| ■ Single supply | 5 V _{DC} |
| ■ Operates ratiometrically or with 5 V _{DC} , 2.5 V _{DC} , or analog span adjusted voltage reference | |

Typical Applications



Connection Diagrams



Index

A

- ADC0804LCN 8-bit successive approximation ADC, 122-123
- ADC0809LCN 8-bit successive approximation ADC, 123
- Address decoding, 55-59
 - absolute, 55-56
 - nonabsolute, 56-59
- Addressing modes, 6809E, 37
- Alphanumeric character codes, 17-18
- Analog input ports, 23-24
- Analog-to-digital converters, 108-126
 - ADC interfacing, 118-122
 - ADC0804LCN, 122-123
 - ADC0809LCN, 123
 - Color computer internal ADC, 114-118
 - ICL7109, 123-126
 - operating fundamentals, 108-114
 - dual-slope integration method, 111-114
 - successive approximation method, 109-111
- AND gate, 186
- AND/OR/XOR logic, 181-182
- Architecture, 6809E microprocessor, 30-33
 - accumulators, 31-32
 - arithmetic logic unit, 31-32
 - condition code register, 32
 - direct page register, 32

Architecture—cont

- index/stack pointer registers, 32
- program counter register, 32
- Audio cassette port, 24

B

- Binary addition, 178-179
- Binary (base 2) number system, 172-173
- Binary coded decimal (BCD) number system, 176-178
- Binary division, 182
- Binary multiplication, 182
- Binary subtraction, 181
- Bits, bytes, and nibbles, 176
- Bus control interface lines, 6809E, 40-43
- Bus status interface lines, 6809E, 40

C

- Cassette port, 24
- Character codes, 17-18
- Clock frequency, 6809E, 38
- Color BASIC interpreter, 12-15
- Condition code register, 32
- Connector, parallel expansion, 20-21
- Converters
 - analog-to-digital, 108-126
 - ADC interfacing, 118-122
 - ADC0804LCN, 122-123
 - ADC0809LCN, 123

Converters—cont

analog-to-digital

Color Computer internal ADC, 114-118

ICL7109, 123-126

operating fundamentals, 108-114

digital-to-analog, 90-108

Color Computer internal DAC, 95-100

DAC interfacing, 101-104

MC1408-8, 104

MC3410, 104, 108

NE5018, 104

operating fundamentals, 90-95

D

Data/address interface lines, 6809E, 39

Decimal (base 10) number system, 171-172

Decoders

74LS42, 57, 60

74LS138, 57, 60

74LS139, 57, 61

74LS154, 57-59

Decoding, address, 55-59

absolute, 55-56

nonabsolute, 56-59

D flip-flop, 188-189

Digital computer arithmetic, 178-182

addition, 178-179

AND/OR/XOR logic, 181-182

division, 182

multiplication, 182

negative numbers, 179-181

ones complement, 179

subtraction, 181

twos complement, 179-181

Digital logic gates, 185-188

AND gate, 186

exclusive NOR gate, 187

exclusive OR gate, 186

gate conversions, 187-188

inverter (NOT gate), 185

NAND gate, 186

NOR gate, 186

OR gate, 185

Digital-to-analog converters, 90-108

Color Computer internal DAC, 95-100

Digital-to-analog—cont

Color Computer internal DAC

positive voltage ramp, program for, 96

sine waveform, programs for, 97, 99-100

square waveform, program for, 97

triangular waveform, program for, 96-97

DAC interfacing, 101-104

MC1408-8, 104

MC3410, 104, 108

NE5018, 104

operating fundamentals, 90-95

step size, 92

voltage resolution, 94

Direct page register, 32

Dual-slope integration method of A/D conversion, 111-114

Dynamic RAM, 14

E

Exclusive NOR gate, 187

Exclusive OR gate, 186

Expansion connector

breadboard, 130-136

pin assignments, 70

signals, 69-73

address decode, 71-72

control, 72-73

data/address, 71

power/clock, 71

Expansion ports, 20-24

analog input, 23-24

audio cassette, 24

joystick interface, 23-24

parallel, 20-21

serial RS-232-C, 22-23

Experiments

1: Data Input and the PIA Interface, 136-140

2: Data Output, 140-143

3: Internal Digital-to-Analog Conversion, 143-151

4: DAC Interfacing, 151-158

5: Internal Analog-to-Digital Conversion, 158-163

6: ADC Interfacing, 163-169

Extended Color BASIC, 13-14

F

- Flip-flops, 188-191
 - D flip-flop, 188-189
 - JK flip-flop, 189-191
- Full decoding, 55-56

G

- Graphics characters, 17-19

H

- Handshaking, 79
 - input, 80
 - output, 81
- Hexadecimal (base 16) number system, 174-176

I

- ICL7109 12-bit dual-slope ADC, 123-126
- Index/stack pointer registers, 32
- Initialization, 6821 PIA, 65-67
- Input handshaking, 80
- Instruction set, 6809E, 34-36
- Interface lines, 6809E, 33, 37-43
 - bus control, 40-43
 - bus status, 40
 - data/address, 39
 - power/clock, 37-38
- Interfacing
 - microcomputer, 47-67
 - address decoding, 55-59
 - absolute, 55-56
 - nonabsolute, 56-59
 - general I/O concepts, 48-55
 - input port, 48-52
 - I/O port, 54
 - output port, 48, 52-54
 - parallel expansion connector, 69-87
 - connector pin assignments, 70
 - connector signals, 69-73
 - address decode, 71-72
 - control, 72-73
 - data/address, 71
 - power/clock, 71
 - I/O device control, 75-87
 - interrupt I/O control, 83-87
 - programmed I/O control, 76-83
 - PIA interface, 73-75
- Internal registers, 6821 PIA, 62-65
 - control registers (CRA and CRB), 62, 64

- Internal registers—cont
 - data direction registers (DDRA and DDRB), 62-63
 - data registers (DRA and DRB), 62
- Interrupts, 6809E
 - maskable, 40-41
 - nonmaskable, 40-43
- I/O instructions, 6809E, 43-46
 - INPUT, 46
 - PEEK, 43-45
 - timing diagram, 45
 - POKE, 43-46
 - timing diagram, 46
 - PRINT, 46

J

- JK flip-flop, 189-191
- Joystick interface ports, 23-24

L

- Latches, 52-55
 - 74LS75, 53
 - 74LS373, 53
- Logic, AND/OR/XOR, 181-182

M

- Maskable interrupts, 6809E, 40-41
- MC1408-8 8-bit DAC, 104
- MC3410 10-bit DAC, 104, 108
- Memory
 - and memory control, 12-15
 - map, 14-15
 - mapped I/O, 16
 - nonvolatile, 12
 - random access, 12-15
 - read-only, 12-15
 - read/write, 12-13
 - stack, 32
 - text screen, 16-17
 - volatile, 14
- Microcomputer interfacing, 47-67
 - address decoding, 55-59
 - absolute, 55-56
 - nonabsolute, 56-59
 - general I/O concepts, 48-55
 - input port, 48-52
 - I/O port, 54
 - output port, 48, 52-54
- Modem, 22

N

- NAND gate, 186
- NE5018 8-bit DAC, 104
- Negative numbers, 179-181
- Nonabsolute decoding, 56-59
- Nonmaskable interrupts, 6809E, 40-43
- Nonvolatile memory, 12
- NOR gate, 186
- NOT gate (inverter), 185
- Number systems, 171-178
 - binary (base 2), 172-173
 - binary coded decimal (BCD), 176-178
 - bits, bytes, and nibbles, 176
 - decimal (base 10), 171-172
 - hexadecimal (base 16), 174-176
 - octal (base 8), 173-174

O

- Octal (base 8) number system, 173-174
- Ones complement, 179
- OR gate, 185
- Output handshaking, 81

P

- Parallel expansion connector interfacing, 69-87
 - connector pin assignments, 70
 - connector signals, 69-73
 - address decode, 71-72
 - control, 72-73
 - data/address, 71
 - power/clock, 71
 - I/O device control, 75-87
 - interrupt I/O control, 83-87
 - programmed I/O control, 76-83
 - PIA interface, 73-75
- Parallel expansion port, 20-21
- Partial decoding, 56-59
- PEEK instruction, 6809E, 43-45
 - timing diagram, 45
- Peripheral interface adapter (PIA), 59, 61-67
 - functional description, 59, 61
 - initialization, 65-67
 - internal registers, 62-65
 - control registers (CRA and CRB), 62, 64
 - data direction registers (DDRA and DDRB), 62-63

Peripheral interface adapter—cont internal registers

- data registers (DRA and DRB), 62
- Pin assignments, expansion connector, 70
- POKE instruction, 6809E, 43-46
 - timing diagram, 46
- Polling, 76
- Power/clock interface lines, 6809E, 37-38
- Program counter register, 32

R

- Random access memory (RAM), 12-15
 - dynamic, 14
- Read-only memory (ROM), 12-15
- Read/write memory, 12-13
- Registers, 6809E microprocessor, 30-33
 - accumulators, 31-32
 - condition code, 32
 - direct page, 32
 - index/stack pointer, 32
 - program counter, 32
- RS-232-C serial expansion port, 22-23

S

- Serial RS-232-C port, 22-23
- 74LS42 1-of-10 decoder, 57, 60
- 74LS75 4-bit latch, 53
- 74LS125 quad 3-state buffer, 50
- 74LS138 1-of-8 decoder, 57, 60
- 74LS139 1-of-4 decoder, 57, 61
- 74LS154 1-of-16 decoder, 57-59
- 74LS240 octal 3-state buffer, 50-51
- 74LS241 octal 3-state buffer, 50-51
- 74LS244 octal 3-state buffer, 50-51
- 74LS373 8-bit latch, 53
- Signals, parallel expansion connector, 69-73
 - address decode, 71-72
 - control, 72-73
 - data/address, 71
 - power/clock, 71
- 6809E microprocessor, 11-12, 27-46
 - addressing modes, 37
 - architecture, 30-33
 - accumulators, 31-32
 - arithmetic logic unit, 31-32
 - condition code register, 32
 - direct page register, 32
 - index/stack pointer registers, 32

6809E microprocessor—cont
architecture
 program counter register, 32
 clock frequency, 38
 general concepts and evolution, 28-30
 instruction set, 34-36
 interface lines, 33, 37-43
 bus control, 40-43
 bus status, 40
 data/address, 39
 power/clock, 37-38
 interrupts, 40-43
 maskable, 40-41
 nonmaskable, 40-43
 I/O instructions, 43-46
 INPUT, 46
 PEEK, 43-45
 POKE, 43-46
 PRINT, 46
 6821 peripheral interface adapter (PIA),
 59, 61-67
 functional description, 59, 61
 initialization, 65-67
 internal registers, 62-65
 control registers (CRA and CRB), 62,
 64
 data direction registers (DDRA and
 DDRB), 62-63
 data registers (DRA and DRB), 62
 Status flags, 6809E, 32
 Successive approximation method of
 A/D conversion, 109-111

T

Text screen memory, 16-17
 Three-state buffers, 48-55
 74LS125, 50

Three-state buffers—cont
 74LS240, 50-51
 74LS241, 50-51
 74LS244, 50-51
 Timing diagram
 PEEK instruction, 45
 POKE instruction, 46
 TRS-80 Color Computer
 circuit, 10-20
 input/output, 15-20
 memory and memory control, 12-15
 memory map, 14-15
 6809E microprocessor, 11-12, 27-46
 video display, 15-20
 expansion ports, 20-24
 analog input, 23-24
 audio cassette, 24
 joystick interface, 23-24
 parallel, 20-21
 serial RS-232-C, 22-23
 internal ADC, 114-118
 internal DAC, 95-100
 positive voltage ramp, program for,
 96
 sine waveform, programs for, 97,
 99-100
 square waveform, program for, 97
 triangular waveform, program for,
 96-97
 Twos complement, 179-181

V

Video display circuit, 15-20
 Video display generator (VDG), 15
 Volatile memory, 14



READER SERVICE CARD

To better serve you, the reader, please take a moment to fill out this card, or a copy of it, for us. Not only will you be kept up to date on the Blacksburg Series books, but as an extra bonus, we will randomly select five cards every month, from all of the cards sent to us during the previous month. The names that are drawn will win, absolutely free, a book from the Blacksburg Continuing Education Series. Therefore, make sure to indicate your choice in the space provided below. For a complete listing of all the books to choose from, refer to the inside front cover of this book. Please, one card per person. Give everyone a chance.

In order to find out who has won a book in your area, call (703) 953-1861 anytime during the night or weekend. When you do call, an answering machine will let you know the monthly winners. Too good to be true? Just give us a call. Good luck.

If I win, please send me a copy of:

I understand that this book will be sent to me absolutely free, if my card is selected.

For our information, how about telling us a little about yourself. We are interested in your occupation, how and where you normally purchase books and the books that you would like to see in the Blacksburg Series. We are also interested in finding authors for the series, so if you have a book idea, write to The Blacksburg Group, Inc., P.O. Box 242, Blacksburg, VA 24060 and ask for an Author Packet. We are also interested in TRS-80, APPLE, OSI and PET BASIC programs.

My occupation is _____

I buy books through/from _____

Would you buy books through the mail? _____

I'd like to see a book about _____

Name _____

Address _____

City _____

State _____ Zip _____

MAIL TO: BOOKS, BOX 715, BLACKSBURG, VA 24060
!!!!!!PLEASE PRINT!!!!!!

The Blacksburg Group

According to Business Week magazine (Technology July 6, 1976) large scale integrated circuits or LSI "chips" are creating a second industrial revolution that will quickly involve us all. The speed of the developments in this area is breathtaking and it becomes more and more difficult to keep up with the rapid advances that are being made. It is also becoming difficult for newcomers to "get on board."

It has been our objective, as The Blacksburg Group, to develop timely and effective educational materials that will permit students, engineers, scientists, technicians and others to quickly learn how to use new technologies and electronic techniques. We continue to do this through several means, textbooks, short courses, seminars and through the development of special electronic devices and training aids.

Our group members make their home in Blacksburg, found in the Appalachian Mountains of southwestern Virginia. While we didn't actively start our group collaboration until the Spring of 1974, members of our group have been involved in digital electronics, minicomputers and microcomputers for some time.

Some of our past experiences and on-going efforts include the following:

-The design and development of what is considered to be the first popular hobbyist computer. The Mark-B was featured in Radio-Electronics magazine in 1974. We have also designed several 8080-based computers, including the MMD-1 system. Our most recent computer is an 8085-based computer for educational use, and for use in small controllers.

-The Blacksburg Continuing Education Series™ covers subjects ranging from basic electronics through microcomputers, operational amplifiers, and active filters. Test experiments and examples have been provided in each book. We are strong believers in the use of detailed experiments and examples to reinforce basic concepts. This series originally started as our Bugbook series and many titles are now being translated into Chinese, Japanese, German and Italian.

-We have pioneered the use of small, self-contained computers in hands-on courses for micro-computer users. Many of our designs have evolved into commercial products that are marketed by E&L Instruments and PACCOM, and are available from Group Technology, Ltd., Check, VA 24072.

-Our short courses and seminar programs have been presented throughout the world. Programs are offered by The Blacksburg Group, and by the Virginia Polytechnic Institute Extension Division. Each series of courses provides hands-on experience with real computers and electronic devices. Courses and seminars are provided on a regular basis, and are also provided for groups, companies and schools at a site of their choosing. We are strong believers in practical laboratory exercises, so much time is spent working with electronic equipment, computers and circuits.

Additional information may be obtained from Dr. Chris Titus, the Blacksburg Group, Inc. (703) 951-9030 or from Dr. Linda Leffel, Virginia Tech Continuing Education Center (703) 961-5241.

Our group members are Mr. David G. Larsen, who is on the faculty of the Department of Chemistry at Virginia Tech, and Drs. Jon Titus and Chris Titus who work full-time with The Blacksburg Group, all of Blacksburg, VA.

TRS-80® Color Computer Interfacing, With Experiments

- Provides the reader with a basic understanding of the inner structure and operation of the TRS-80® Color Computer, its major LSI components, and its expansion capabilities.
- Explains the various features of the 6809E microprocessor—the central processing unit of the TRS-80® Color Computer.
- Shows how the three fundamental interfacing functions—decoding, three-state buffering, and latching—can be performed using digital logic design techniques.
- Describes how to use a programmable interface adapter (PIA) to develop a simple parallel input/output interface for the TRS-80® Color Computer expansion connector.
- Covers digital-to-analog (D/A) and analog-to-digital (A/D) converters and shows how these devices can be connected to the TRS-80® Color Computer to monitor and control external events.
- Presents six practical experiments in the final chapter designed to demonstrate the material presented in the preceding chapters.
- Contains three appendixes of reference material covering number systems and computer arithmetic, a review of digital logic circuits, and manufacturers' data sheets on the 6809E CPU, the 6821 PIA, the NE5018 D/A converter, and the ADC0804LCN A/D converter.



Andrew C. Staugaard, Jr., is an experienced engineer/educator in microelectronics. Currently, he is an Associate Professor of Electrical Engineering at Jamestown Community College, a State University of New York campus. In addition, he is an Adjunct Professor for Rochester Institute of Technology. In 1981, he received the Chancellor's Award for Excellence in Teaching from the State University of New York.

Prior to teaching, Professor Staugaard was employed as a quality control engineer in microelectronics processing by the Bendix Corporation, Kansas City Division. He is the author of three additional SAMS books: *How to Program and Interface the 6800; 6801, 68701, and 6803 Microcomputer Programming & Interfacing*; and *6809 Microcomputer Programming & Interfacing, With Experiments*.

HOWARD W. SAMS & CO., INC.

4300 West 62nd Street, Indianapolis, Indiana 46268 USA

\$14.95/21893

ISBN: 0-672-21893-3